SugarLoafPLoP 2002 Proceedings

Editors
Rosana T. Vaccare Braga
Joseph W. Yoder

Organized by:
COPPE/UFRJ - Brazil
ICMC/USP - Brazil
UFC - Brazil
UIUC - USA

Underwritten by the Hillside Group
Supported by SBC (Brazilian Computer Society)

Publisher: ICMC/USP

Editors: Rosana T. V. Braga and Joseph W. Yoder

Latin American Conference on Pattern Languages of Programming SugarLoafPLoP 2002 (2.: 2002 : Itaipava, RJ)
Proceedings… / Editors Rosana T. Vaccare Braga, Joseph W. Yoder. -- São Carlos, SP: ICMC/USP, 2002.
326 p.
ISBN 85-87837-07-9

1. Padrões de Software. 2. Linguagens de Padrões. I. Braga, Rosana T. Vaccare, ed. II. Yoder, Joseph W., ed. III. Título.

Program Comittee

Conference Co-Chairs

Claudia M. L. Werner (COPPE / UFRJ, BR)
Rossana M. Castro Andrade (DC / UFC, BR)

Program Co-Chairs

Rosana T. Vaccare Braga (ICMC-USP, BR)
Joseph W. Yoder (University of Illinois/The Refactory, Inc., US)

Local Organization

Leonardo G. P. Murta (COPPE/UFRJ, BR)
Marcio de Oliveira Barros (COPPE/UFRJ, BR)

Shepherds

Federico Balaguer - University of Illinois at Urbana-Champaign, USA
Fernão Stella de Rodrigues Germano - ICMC/Universidade de São Paulo, Brasil
Gustavo Rossi - Universidad Nacional de La Plata, Argentina
Jerffeson Teixeira de Souza - University of Ottawa, Canada
Jorge Ortega Arjona - University College London, UK
Jugurta Lisboa Filho - Universidade Federal de Viçosa-MG, Brasil
Márcio de Oliveira Barros - COPPE/Universidade Federal do Rio de Janeiro , Brasil
Robert Hanmer - Software Technology Center - Bell Labs, USA
Rossana Maria de Castro Andrade - DC/Universidade Federal do Ceará - Brasil

Referees

Adenilso da Silva Simão - ICMC/Universidade de São Paulo
Antonio Castelo Filho - ICMC/ Universidade de São Paulo
Antonio Valerio Netto - ICMC/ Universidade de São Paulo
Claudia Lima Werner - COPPE/ Universidade Federal do Rio de Janeiro
Ernesto Massaroppi - DEM/Universidade de São Paulo
Fernando de Carvalho Gomes - DC/ Universidade Federal do Ceará
Fernão Stella de Rodrigues Germano- ICMC/ Universidade de São Paulo
Julio Wilson Ribeiro - DC/ Universidade Federal do Ceará
Maria Istela Cagnin- ICMC/ Universidade de São Paulo
Paulo César Masiero- ICMC/ Universidade de São Paulo
Rosana T. Vaccare Braga- ICMC/ Universidade de São Paulo
Rosangela A. D. Penteado - DC/Universidade Federal de São Carlos
Rossana Maria de Castro Andrade - DC/ Universidade Federal do Ceará
Willie Dresler Leiva - ICMC/Universidade de São Paulo

Table of Contents

Foreword

Software developers have long observed that certain themes recur and endure across different applications and systems. The emerging interest in patterns represents an effort to catalog and communicate these themes and motives to provide handbooks of proven solutions to common problems.

SugarLoafPLoP brings together researchers and practitioners whose interests span a remarkably broad range of topics, who share an interest in exploring the power of the pattern form. SugarLoafPLoP invites you to add your expertise to the growing corpus of patterns.

SugarLoafPLoP focuses on improving the expression of patterns. Here you have the opportunity to refine and extend your patterns with help from knowledgeable and sympathetic fellow patterns enthusiasts.

PLoP Conferences usually restrict paper submissions to works that propose patterns. However, for some of them, like Chilli PLoP, hot topics involving patterns are acceptable. Following this trend, this year SugarloafPLoP has introduced two special sessions: "Software Pattern Applications" (SPA) and "Writing Patterns" (WP).

These proceedings are the result of the Conference. Besides improving their papers during the shepherding process, authors have gained insights and constructive criticism during the workshops so that, after the Conference, they have evolved their papers to the version presented in these proceedings. As this is a Latin American Conference, we allow papers to be written in English, Spanish, or Portuguese. In these proceedings we have eleven English papers and seven Portuguese papers.

The efforts of many people contributed to the SugarloafPLoP Conference. We thank all authors that submitted papers, making possible the realization of this event. We thank the shepherds that have dedicated their time to help us, as well as the referees that revised the special session papers. We specially thank the Conference co-chairs Claudia and Rossana for their excellent and hard work in the organization of this event. We thank Leonardo and Marcio that helped in the local arrangements and in the Web site maintenance. Finally, we thank the cooperation of the Brazilian Computer Society (SBC) and the sponsorship of CNPq, FAPERJ, and the Hillside Group.

Rosana T. Vaccare Braga (rtvb@icmc.usp.br)
Joseph W. Yoder (joeyoder@joeyoder.com)
Program co-chairs

November, 2002

Writers' Workshops

Each workshop session was about 1 hour and 15 minutes in length. There was a special 30 minute session at the beginning of the conference so that each workshop group could be introduced and work out logistics, such as how much they wanted non authors to participate and in which order the papers would be presented.

The Workshop Process

The writers workshop format has proven to be useful in past PLoP conferences, so it was followed by each group. Although this format may seem unfamiliar, it has shown to be useful for developing an environment where patterns authors can share their ideas.

Introduction/Reading: Moderator introduces the Author and the Author reads a selection from the paper. This is the last we hear of the author til the end. (allow 5 minutes)

Summary: One of the workshop participants summarizes the paper.(5 minutes)

Positive Feedback: Moderator asks for things people liked about the patterns. The comments can be about presentation or content, and at the discretion of the moderator comments about presentation and content can be intermingled, or done separately. (Allow 15 minutes)

Constructive Criticism:Moderator asks for ways in which the paper can be improved, both in content and presentation. (Allow 20-40 minutes)

Positive Closure: Moderator asks particpants for a final closure, in which they reinforce the positive aspects of the pattern. (Allow 5 minutes)

Author Feedback: The author asks for clarification on comments made during the session. The Author should pick a few of the most important points. (or ones which were made by the most people.) Further clarification can be had during off line discussions. (Allow 10 minutes)

Closing: The workshop participants thank the author.

DORS: Database Query Optimizer with Rule Based Search Engine

Carlo Giovano S. Pires[1]
cgiovano@atlantico.com.br
Instituto Atlântico

Javam C. Machado
javam@ufc.br
Federal University of Ceará

Abstract

The database query optimizer is a very important and complex module in database management systems. It receives a query optimization request with a query tree as a parameter and return an optimized execution plan. The query optimization problem is NP-Hard; therefore, there are many proposals of heuristics and techniques for optimization strategies. There are also several data models (e.g object-oriented, relational, object-relational and semi-structured/XML) suitable to store information for different kinds of applications. Several optimization frameworks were proposed with the aim of making easier to build optimizers and reuse design decisions. However, they are tied to some specific language and hard to integrate with other database modules. We propose a design pattern to help the design and construction of a database optimizer. So far, we do not have knowledge about similar work.

1. Context

Different types of applications should use a suitable data model. For instance, commercial systems work well with relational data models, CAD/CAM systems need a more expressive data model as the object-oriented, and the Internet with XML applications work well with semi-structured data models. Different data models and different kinds of applications require specific implementation of the query optimizer. The database developer should design optimizers able to support different kinds of databases, data models and applications. Some examples of database and application kinds are object-oriented databases, multidatabases, and parallel databases. Object-oriented databases use a model that requires the optimizer to support different data types, use of indexes for class hierarquies and method execution. Multidatabase is a front-end application that integrates different kinds of database. A multidatabase optimizer should support different data models, distributed data and data transfer issues.

The large adoption of Web-based applications in the Internet has increased the number of users and the number of database query requests. Thus, scalability is important for a database server. Parallel database architecture is a good alternative for improving scalability and response time. However, parallel query optimization is a very complex problem. The optimizer for parallel databases should balance the trade-offs of memory use, data partitioning, data transfer, synchronization of operations and a very large search space. Different kinds of parallel database architectures must also be analyzed.

A database query processing module (Figure 1) provides the retrieval of information using a high-level query language such SQL [ISO96] or OQL [CB97]. The query processing module is usually composed of three other modules. The first module is the query parser. The parser converts a query, submitted by a database user and written in a high-level language, into an algebraic operators expression. Next, the optimization module receives the expression and builds a good execution plan. The plan determines the order of execution of the operators and select suitable algorithms for implementation of the operators. The plan is built with the

[1] The presentation of this work was sponsored by Instituto Atlântico (www.atlantico.com.br)

aim of retrieving the result of the query with high performance. Finally, the query plan is executed by the execution engine module that deliver the result for the user. The query optimization module, or optimizer, is the key module for query processing design.

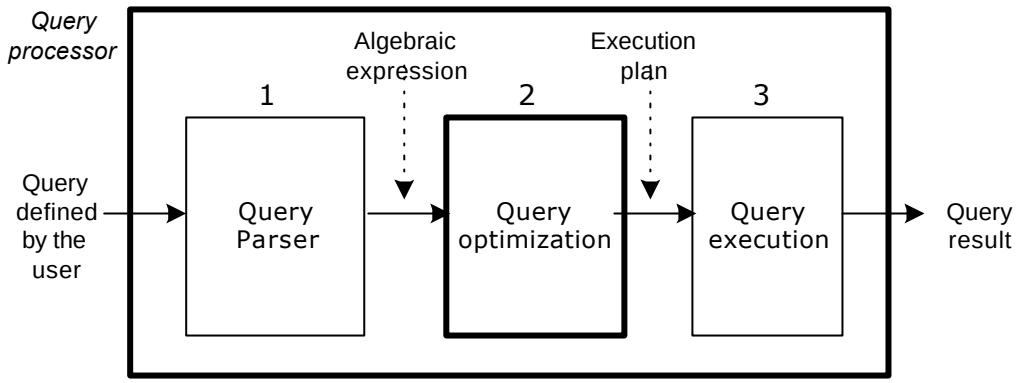


Figure 1 – Database query processor

The optimization process is usually broken into different phases. This approach simplifies the optimization problem using each phase to optimize a specific aspect of the query. The aspects may be, for instance, the logical optimization and the order of operations, the algorithms for implementation of the operations and the allocation of them into the nodes of a parallel or distributed system. Each aspect requires specific algebraic operations, transformations and specific optimization algorithms.

Database developers and researchers of optimization techniques may use the DORS (*Database Query Optimizer with Rule Based Search Engine*) pattern to help them to build extensible optimizers.

2. Problem

How to design an extensible database query optimizer that supports different data models, new algebra proposals and search strategies?

3. Forces

- *Maintainability and Prototyping*: The optimizer should support different data models. The database developer should be able to change the specific implementation related to a specific data model transparently to other aspects (e.g, optimization algorithms and design of optimization phases and their ordering). For instance, the developer may change the transformation of operators related to a data model and add transformations of operators of a new data model without changing the optimization. The developer may even add new operations and transformations for a data model and keep the other components implementation.

- *Flexibility:* The database developer should be able to change the optimization algorithm, configure optmization algorithms for each phase, add and experiment new proposals without changing the other components. The optimizer may be configured according to a query context or according to a database user configuration. For instance, a database may support queries written in object query language, or semi-

tructured languages and, in run time, choose the specific algebraic transformations. Some databases allows the user to select the type of optimization algorithms, as cost-based or heuristics-based algorithms.

- *Scalability*: The optimizer should be scalable to support a large number of simultaneous requests. In the last years, the growth of Internet applications has opened the access of information for millions of simultaneous users.

4. Solution

Decouple the optimization process/procedure from each of its phases by creating a hierarchy of search strategies (SearchStrategy). It also decouples each phase from the set of transformation rules (*Rule*) that a search strategy can apply. Instances of *SearchStrategy* are parameterized with a given set of instances of Rule that it can apply over a query expression.

A *Rule* is composed of a header, a promise value, an applicability condition and a transformation code. The header defines a pattern matching description of the target expression. The applicability condition uses properties of the expression and the optimization context to determine if the transformation code is executed or not. The transformation code rewrites the expression, modifies the optimization context and estimates values for the properties of the new expression (eg. ordering of the result and execution cost of the expression). The promise is used to order the application of the rules.

An instance of *Optimizer* can have more than one optimization phase. Each phase is implemented as an instance of *SearchStrategy*. The configuration of one *Optimizer* and its *SearchStrategies* can be changed at run-time. When an optimizer gives a query expression to a *SearchStrategy*, the *SearchStrategy* applies the appropriate set of rules to the expression and stores the results on a repository of optimized expression (*SearchSpace*). Finally the optimizer compose the optimal plan retrieving optimized expressions from the repository.

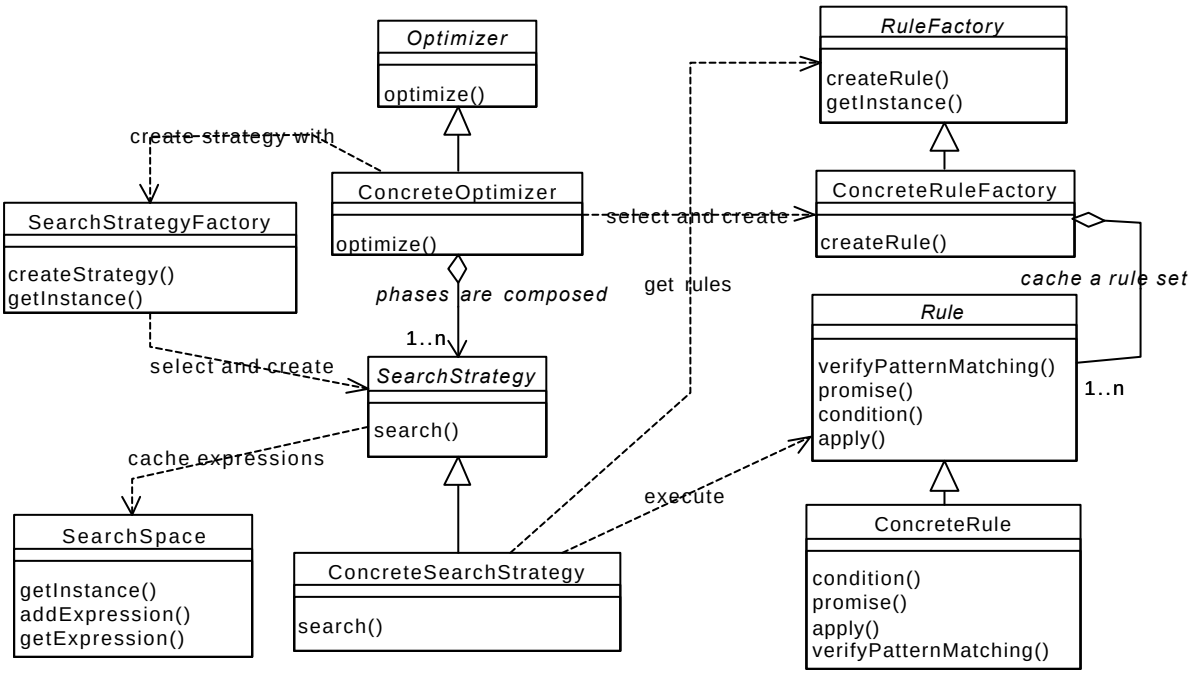


Figure 2 – Structure of DORS Pattern

5. Structure

Figure 2 presents the structure of DORS pattern using an UML class diagram. A detailed description of each component is presented in section *Participants.*

6. Participants

- **Optimizer**
  - provides a simple interface for the optimization subsystem of the query processor.
  - defines the method *optimize* that receive, as input, a query expression (usually represented as a tree). The output of the *optimize* method is an expression that represents an execution plan. The execution plan determines the order, processor allocation and algorithms for the query operations.

- **ConcreteOptimizer**
  - implements the *Optimizer* interface.
  - coordinates the optimization process. The *ConcreteOptimizer* defines the optimization phases, coordinates rule sets and search strategies. It creates and uses one or several types of *SearchStrategy* and apply them in some order, according to the optimization phases adopted in the optimization model.
  - delegates the implementation of the search algorithm for optimization to some *ConcreteStrategy*. The *ConcreteOptimizer* may work with more than one strategy and rule factory for different optimization phases.
  - combines dynamically different types of rule sets, search strategies and cost models.

- **SearchStrategyFactory**
  - encapsulates the creation of search strategies.
  - has a method named *createSearchStrategy* that receives a parameter identifying the phase/order/task of optimization.

- **SearchStrategy**
  - defines an interface for expanding the search space and searching for good expressions.
  - has a method named *search* that receives the target expression and the factory of rules as parameters.
  - hides one or more specifics types of search strategies within an optimizer for one or more phases.

- **ConcreteSearchStrategy**

  - implements the services for query optimization search strategies as dynamic programming, exhaustive search, bottom-up search, or some strategy else.
  - collaborates with rules to execute transformation in the query expression, expanding the search space and exploiting optimization possibilities.
  - uses the *RuleFactory* to create a list of rules.
  - process each rule received from the factory against the expression.
  - adds the expressions generated by the rules to the search space.
  - determines how to apply the rules to each term in the expression.

- **SearchSpace**

  - provides a data structure for storage of the expressions generated by the optimization process.
  - provides methods to store and find a expression in the search space.

- **Rule**

  - encapsulates a transformation of a term within an expression.
  - provides a mechanism based on pattern matching for verifying if it is applicable to a given term.
  - provides a method named *promise* to return a value to help the search strategy to give priority to the application of rules.
  - provides a method to verify some condition of application. After the *Pattern Matching* and before the execution of the rule, this method is verified. The rule is executed only if the method returns true.

- **ConcreteRule**

  - implements a specific transformation rule for some algebraic operator under some condition.
  - defines a pattern description of the expression it should be applied.
  - may usually work as a logical rule (mapping logical algebraic operators into logical algebraic operators and reordering them), a physical rule (mapping a logical operator into a physical operator), or an enforcer rule (use to enforce required properties, as the order of the query result, or the partition across processing nodes in a parallel system).
  - estimates the expression cost. The cost is estimated using database statistics. Physical rules are tied to physical operators, therefore they are suitable for defining cost functions and applying them.

- **RuleFactory**
  - defines the abstract interface for a *ConcreteRuleFactory*.
  - represents a rule set (or rule module). A rule set groups rules that should be applied in a specific context or optimization phase. A rule set may be defined for a logical algebra, a physical algebra, or different proposal for logical and physical algebras.

- **ConcreteRuleFactory**
  - creates the rule objects for some algebra according to the expression.
  - uses the pattern matching mechanism to select the rule for the target expression.
  - uses a generic pattern matching mechanism that allows a *ConcreteRuleFactory* compare an expression with the rules and discover which are applicable to the expression. This mechanism uses the information about the top node in the expression to execute compare the expression with the pattern description in the rule. The node information may be the algebraic operator in the node and details about the parameters of the operator.

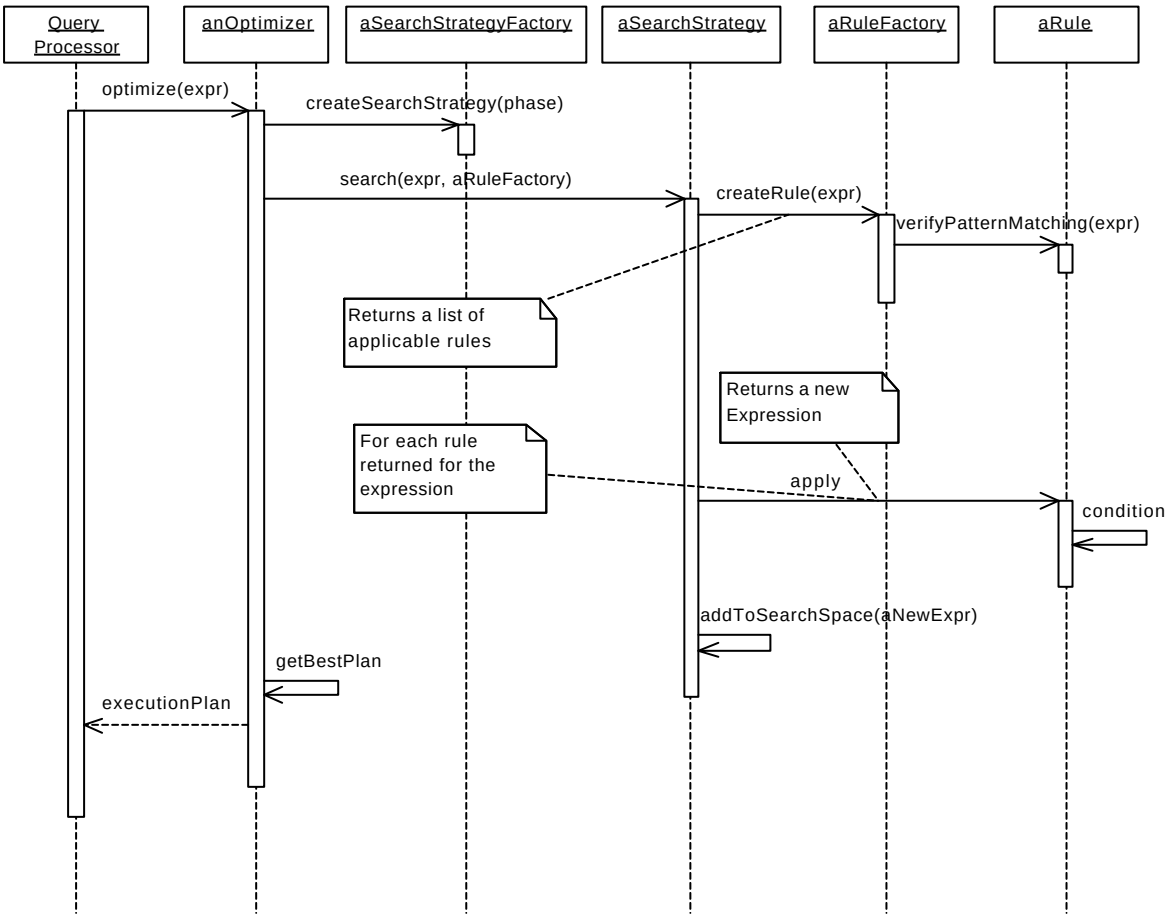


Figure 3 – Dynamics of DORS

7. Dynamics

Figure 3 shows, using a sequence diagram [FMSK00], a typical scenario of query optimization with the DORS pattern. The query processor starts the optimization sending an *optimize* message to the optimizer. Then, the optimizer uses a factory to create a concrete search strategy according to the optimization phase. The optimizer uses the *search* method of the concrete search strategy to start the tranformations of the target expression. The *search* method recieves the expression as a parameter and it is configured with some rule factory. The method uses the rule factory to create rules (calling the *createRule* method with the query expression as a parameter). The *createRule* method iterates over a pool of rules, according to the order of promise, and chooses a rule using the *verifyPatternMatching* method. All rules that matches the pattern for the expression in the optimization context are added, in order of promise, to a list. This list is returned to the search strategy. The search strategy iterates over the returned list and runs each rule using the *apply* method. This method verifies the condition of execution before transforming the expression. If the *condition* method returns true, then the rule is executed and a new expression is generated. The cost of the new expression is estimated and the expression is finally returned to the search strategy. The search strategy adds each new expression generated from each rule to the search space. The transformations are done for one term of the expression. Each specific search strategy has its own way of exploring the other terms and sub-expressions. The search strategy type also defines the criteria to add the generated expressions to the search space. After the transformation process is finished, the optimizer gets the best plan from the search space and returns it to the database execution engine as the selected execution plan.

8. Consequences

- *Maintainability and Prototyping*. The rule abstraction allows the easy adoption of new algebraic operations. New operators may be used due to the adoption of a new data model or due to the support for new characteristics in a database. The search engine abstraction allows the adoption of new optimization algorithms or prototyping of new techniques.

- *Flexibility*. The database system has the flexibility to use different implementations of the optimizer. The implementations may differ according to query contexts and languages, or according to different architecture of the database server (a parallel server for instance). *Rule* and *RuleFactory* abstractions allows the optimizer to work with several data model (e.g. object-oriented, relational, semi-structured) and architectures (e.g. sequential, parallel, distributed). Supporting new data models requires defining and adding new rule sets.

- *Scalability*. The *Flyweight* pattern used for rule factories and rule objects provides the management of the large number of rule objects necessary to support the processing of a huge number of query requests, improving the scalability of database servers. The multi-phased implementation provided by the *SearchStrategy* and *SearchStrategyFactory*, and the cost model encapsulation into rules provides flexibility for use of parallel machines as database servers. Parallel data server provides high degree of scalability. The pipelining of *SearchEngine* tasks allows the use of parallelism within the optimization phases.

- *Encapsulation.* The rule based approach with *RuleFactory*, *ConcreteRuleFactory*, *Rule* and *ConcreteRule* classes provides the encapsulation of data model/algebra and cost estimation.

- *Dynamically combines phases, search strategies and rule sets.* Usually database optimization algorithms divides the problem in different phases to decrease the problem complexity. For instance, an optimizer may execute a logical optimization phase to execute structural transformations in the query expression, and then, execute a physical optimization phase to choose algorithms suitable to implement the operators in the expressions generated in the first phase. The DORS pattern allows each phase to be modelled with a search strategy abstraction, so one may even use different strategies to each phase. The optimizer configure *SearchStrategy* with a *RuleFactory.* The *RuleFactory* interface lets a *SearchStrategy* to use a rule set transparently. The *SearchStrategy* works with *Rules* returned from the *RuleFactory* and apply them without being tied to the algebraic transformation.

- *Performance*: The execution time of a query is the critical component to response time. However, the optimization time is also important for the final result. The complete search space for a query expression is huge. The optimizer uses heuristics and pruning techniques in the search strategies and rules to reduce the search space. The rational pruning of the search space is the key point to keep optimization time small. However, decoupling optimizer components to gain flexibility may introduce some overhead in the optimization process. This overhead may be in memory use due to extra classes as factories or may be in execution time due to the indirection introduced with abstract classes and polimorphism. In standart database servers, these issues may not be critical to due the availability of memory and processor resources, but in low-resource devices as PDA's, the issues might be critical. In this case, the database developer may consider the trade-offs between flexibility and performance as stated in the *Variants* section.

9. Implementation

- The query processing module may pass a list of search strategies types in the constructor of the *Optimizer* to configure the algorithms for different optimization phases, or even, pass a search strategy factory that creates a strategy according to a phase parameter.

- A *RuleFactory* implements the *Flyweight* pattern [GHJV95], creating an instance of each rule type and adding it to a pool in order of promise (the *promise* method of the *Rule* type). When the search strategy ask for a rule, the factory iterates over the pool (in the promise order) and applies the *verifiesPatternMatching* method against the expression passed as parameter of the factory method. If the method returns true, the instance is added to a return list. After the process had walked through the entire pool, the list is returned. Actually, the *RuleFactory* returns a subset of the pool.

- The *SearchSpace* supports a fixed number of instances (greater than one). The pool of instances is used to organize the expressions into groups. It also may be implemented

as a MEMO structure, using the *Memoization* technique [Gra95]. This technique lets the optimizer ask to a MEMO structure if an expression was generated for a given rule (transformation). Thus, it avoids a rule being applied two times for the same expression.

- A common way of implementing optimizers is to divide the optimization tasks in phases. Usually, one phase runs a complete set o rules and after generating all possible expressions, it starts the next phase using the expressions produced. Another interesting technique is to build a pipeline of tasks where as soon as one expression is produced it is passed to the next tasks. This may be implemented just passing a *SearchStrategy* instance to its predecessor search strategy. The first one calls the second as soon as a rule produces a list of expressions. This process should use just abstract interfaces for the search strategies, letting the optimizer configure the order of them.

- The *Expression* class should be a tree data structure that stores in each node a representation of a database query operation. These operations may be, for instance, a data read from the disk, a join of two data streams, the formatting of data for the final result. Besides the operation description, each node stores attributes for the sub-expression delimited from the node to the leafs of the tree. These attributes may be, for example, the cost of execution of the sub-expression or the order of the delivered intermediate, or final, result. The result of the optimization process is also an expression named plan. The plan is evaluated by the execution engine by executing the operations from the leafs towards the root node.

10. Related Patterns

- *Strategy* [GHJV95]:
  - The *Strategy* pattern is used to let the *ConcreteOptimizer* (the client) to configure the *SearchStrategy* with some type of *RuleFactory*. Thus, the *SearchStrategy* may execute transformations (execution of a *Rule*) according to the *ConcreteOptimizer* choice.

- *Façade* [GHJV95]:
  - *ConcreteOptimizer* works as a *Façade* to coordinate the optimization phases, search strategies and rule factories.

- *Abstract Factory* [GHJV95]:
  - *SearchStrategyFactory* and *RuleFactory* are *Abstract Factories*. They use concrete implementations to build families of search strategies and rules.

- *Singleton* [GHJV95]:
  - *SearchSpace* is a singleton. It provides a single point of access to store and retrieve generated expressions in a memory cache.
  - *Rule* is a singleton. It provides a single instance of each rule type.

- *Flyweight* [GHJV95]:
  - The *RuleFactory* uses this pattern to improve performance in the management of several fine-grained rule objects during the optimization process.

11. Sample Code

The *Fortaleza XML Data Server* (FoX) is a project with the aim of providing fast storing and retrieval operations for XML [W3C] data. One key area of the project is to build a flexible optimizer to evaluate techniques of optimization for this new data type. The project basically investigates two alternatives. The first one uses an object-oriented data model to store XML over Lambda-DB [FSRM00]. This implies that the database query parser should convert a *XQuery* [W3C] query into an object-query tree and pass it to the optimizer. The second alternative stores the XML data using the GOA data manager [MS94] and uses an algebra suitable for *XQuery*. In both solutions, the project adopt a two-phase optimization model with a Top-Down search strategy for the two-phases.

Figure 4 presents the structure of the first approach (object-oriented data-model). This solution uses the object-oriented monoid algebra [Feg97]. It has the *LogicalMonoidRuleFactory* for the logical and first phase and the *PhysicalMonoidRuleFactory* for the physical phase. The solution uses a configuration file to declare the algebra used.

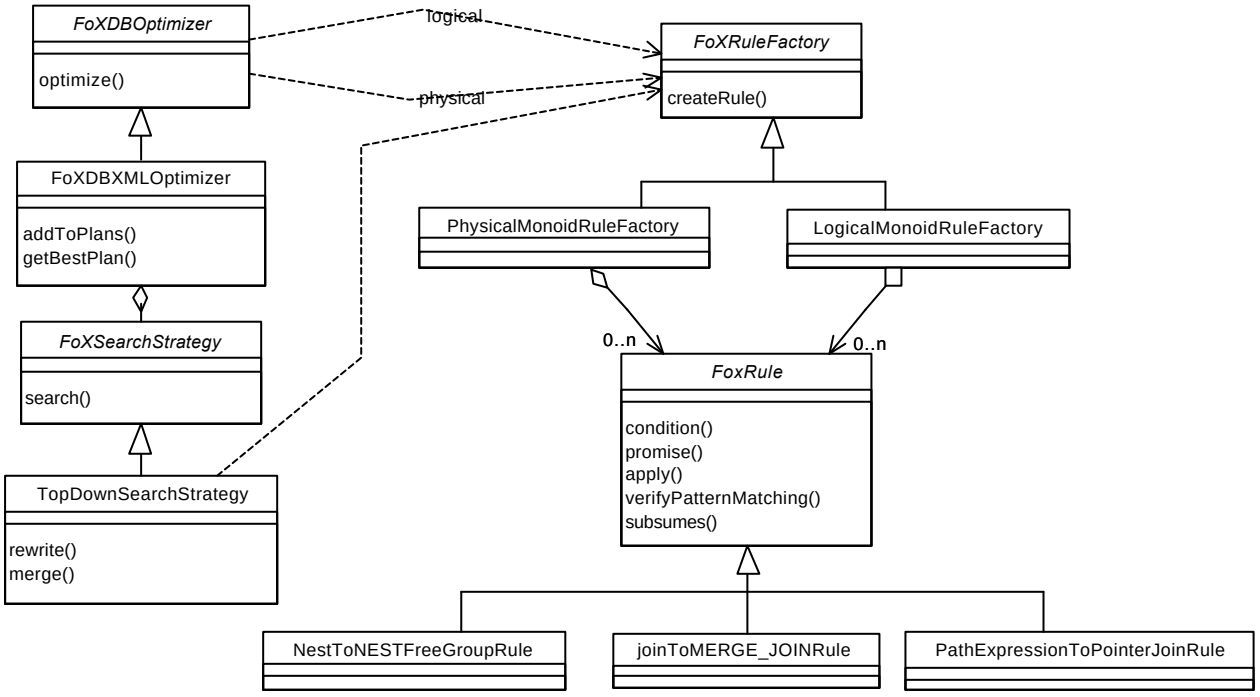


Figure 4 – Design of the database query optimizer of the FoX project

The diagram shows just three rules (there are more than twenty). The rule *PathExpressionToPointerJoin* is from the logical set and the other two are physical rules.

The class *FoXDBXMLOptimizer* uses the *FoxSearchStrategyFactory* to create search engines for each phase. The optimizer stores one instance of a search engine for each phase

into a private variable named *strategies*. In this implementation, the optimizer uses only one search strategy type for the two phases and two rule sets. However, the *FoxSearchStrategyFactory* reads a configuration file that determines a strategy for each phase. The *FoXDBXMLOptimizer* does not apply a pipelined processing, rather, it process the entire logical rule set for the Monoid algebra and, for each new generated expression, it process the second phase to select physical operators. The physical operators are algorithms selected to efficiently implement the logical operators. Each new physical expression generated in the second phase is added to the list of execution plans. In the end of the method, the optimizer executes the *getBestPlan* method that, based on the cost of expressions, chooses the best one. This plan is returned to the execution engine of FoX Database.

```
public class FoXDBXMLOptimizer extends FoXDBOptimizer {

private List strategies;

public FoXDBXMLOptimizer(){
        strategies = new ArrayList();
        FoXSearchStrategyFactory sFac = FoXSearchStrategyFactory.getInstance();
        strategies.add(sFac.createStrategy("phase1"));
        strategies.add(sFac.createStrategy("phase2"));
}

public Expression optimize(Expression expr){
       List plans;
       FoXSearchStrategy search1 = (FoXSearchStrategy) strategies.get(0);
       FoXSearchStrategy search2 = (FoXSearchStrategy) strategies.get(1);

       FoxRuleFactory logicalMonoidRuleFactory = new LogicalMonoidRuleFactory();
       FoxRuleFactory physicalMonoidRuleFactory = new PhysicalMonoidRuleFactory();

       List logicalExprs = search1.search(expr, logicalMonoidRuleFactory);

       for(int i=0; i < logicalExprs.size(); i++){
              Expression logicalExpr = (Expression)logicalExprs.get(i);
              List phyExpr
                  = search2.search(logicalExpr, physicalMonoidRuleFactory);
              addToPlans(plans, phyExpr);
       }

       return getBestPlan(plans);
}

...

}
```

The *TopDownSearchStrategy* extends the *FoXSearchStrategy* abstract class and provides an implementation for its methods. It receives in the constructor a *FoXRuleFactory* and stores the instance into a private variable named *ruleFactory*. This implementation uses an auxiliar method named *rewrite* in the *TopDownSearchStrategy* class. The *rewrite* method does not belong to the pattern and it is used in this specific implementation to provide the recursive calls necessary to implement a top-down search strategy The search method calls the *rewrite* method. This kind of strategy optimize first the parameters of a term in the expression and then, with the optimized sub-expressions, optimize the term. During this process, the strategy passes attributes that represents required properties that helps the lower levels of recursive optimization to provide new expression according to a context. This context may be, for

example, some specific order, a required data partition for parallel a database, or even, lower and upper costs for pruning plans.

```
public class TopDownSearchStrategy extends FoxSearchStrategy{

protected RuleFactory ruleFactory;

public TopDownSearchEngine(FoxRuleFactory rf){
       ruleFactory = rf;
}

public void search(Expression expr){
    Attributes atrbs = new Attributes();
    rewrite(expr, atrbs, ruleFactory);
}

...

}
```

The first action of the *rewrite* method from the *TopDownSearchStrategy* class is to verify if the expression being optimized is not already stored in the search space. If the expression is already optimized, the *SearchSpace* returns a list of expressions. This list stores all the plans generated for the expression. The search strategy uses the list to get the plans stored in the search space and abort the rest of the rewrite process.

If there is no expression list for the expression, the rewrite method starts the generation of new optimized expressions. The first action is to ask to the *ruleFactory*, that may be a *LogicalMonoidRuleFactory* or *PhysicalMonoidRuleFactory*, to create a list of rules applicable to the expression. For each rule in the returned list, the method gets the parameter of the expression and optimize them first, applying recursively the *rewrite* method. Before calling the method recursively, it asks the rule for expected attributes. If the rule needs some properties to be produced in the lower levels of optimization, it returns a valid instance of attributes. The expected attributes are passed in the recursive call. Note that the recursive calls end when the method reaches the leafs terms because they do not have parameters. Each list returned in the recursive calls is added to a list of optimized parameters. For simplicity of the presentation of the method, the examples shows a method named *combine* that builds a list of lists of parameters, representing all the combinations of optimized parameters.

Each rule is applied to every combination of optimized parameter and the result of the *apply* method of the rule is merged with the list of rewrote expressions. This process builds a list with the union of the results of the application of the rules for the expression. Finally, these expressions are added to the search space and returned as the result of the method *rewrite* from the *TopDownSearchStrategy* class.

```
public class TopDownSearchStrategy extends FoxSearchStrategy{

...

public Expression rewrite(Expression expr, Attributes atrbs){

List rewroteExpressions;
List optimizedExpressions = SeachSpace.getInstance().getExpression(expr, atrbs);

if(optimizedExpressions != null){
    return optimizedExpressions;
}
```

```
List rules = ruleFactory.createRule(expr);

for(int i = 0; i < rules.size(); i++){
       Rule rule = (Rule) rules.get(i);
       Attributes expectedAttrbs = rule.getExpectedAttribs();
       if(expectedAttrbs !=null) attribs = expectedAttrbs;

       List optimizedParams = new ArrayList();
       List params = expression.getParams();
       for(int p=0; p<params.size(); p++){
              optimizedParams.add(rewrite((Expression)params.get(p), attribs));
       }

       List combinedParams = combine(optimizedParams);
       for(int p=0; p < combinedParams.size(); p ++){
              List optimizedCombinedParams = (List) combinedParams.get(p);
              List transformedExprs = rule.apply(optimizedCombinedParams);
              merge(rewroteExpressions, transformedExprs);
       }
}

SeachSpace.getInstance().addExpression(rewroteExpressions);

return rewroteExpressions;

}

...

}
```

The *condition* method of the *joinToMERGE_JOINRule* class receives, as its argument, the list of parameters that belong to the expression being optimized. The method uses the parameters to verify if the rule should be applied to the expression. The method gets the second, the third and the forth parameters, that corresponds, respectively, to the left expression (x), right expression (y) and predicate of the join operation. The MERGE-JOIN algorithm expects the data stream in the left expression to be ordered according to the attributes of the join predicate. Thus, the condition method uses another auxiliar method, named *subsumes*, to verify this condition. If the left expression is ordered according to the predicate, then the MERGE-JOIN algorithm may be used and the method returns true. Otherwise, it returns false.

```
public class joinToMERGE_JOINRule extends FoxRule{

...

private boolean condition(List params){
       List params = expr.getParams();
       Expression x = (Expression)params.get(1);
       Expression pred = (Expression)params.get(3);

       if(subsumes(pred, x)){
              return true;
       }
       else{
              return false;
       }
}
```

```
...
}
```

The apply method of the class *joinToMERGE_JOINRule* transforms (rewrite) the logical term denoting a *join* operator into a physical term denoting a *MERGE_JOIN* algorithm to implement the join of two streams. First, the method verifies the condition of the applicability of the rule. If the *condition* method returns true, then the transformation begins. The method gets the second and third parameters of the expression, that corresponds, respectively, to the left plan (x) and right plan (y) of the join operation and use them to estimates the cost of the new expression. It creates an instance of the *Attribute* class. This class is a simple data structure used to store properties of the plan. The properties are the ordering, size and cost of the plan. The cost of the new expression is estimated according to a cost model that states that the cost of a MERGE-JOIN algorithm is the sum of the size of the left and right plan. This value is doubled to accumulate the cost of the sub-expressions. Finally a new expression is created identifying a MERGE_JOIN term with the parameters and the new attributes. The expression is added to the return list and the list is returned as the result of the *apply* method. The return value is a list because a rule may create alternate plans with a loop.

```
public class joinToMERGE_JOINRule extends FoxRule{

...

public List apply(List params){

       List returnList = new ArrayList();

       if(condition(params)){

              Expression x = (Expression)params.get(1);
              Expression y = (Expression)params.get(2);

              Attributes atts = new Attributes();
              atts.setSize(x.getAttributes().size() * y.getAttributes().size());
              atts.setCost(2*(x.getAttributes().size()
                                                  * y.getAttributes().size()));
              atts.setOrder(x.getAttributes().order());
              Expression newExpr = new Expression(Expression.MERGE_JOIN, params,
                                                                        atts);
              returnList.add(newExpr);
       }
       return returnList;

}

...

}
```

12. Variants

- A *Factory* class may be introduced to create the concrete types of *RuleFactory* according to some system configuration. This implementation allows the use one single optimizer implementation for different data models.

- Changes in optimizers usually occurs in data and cost models, forcing changes in rules and rule factories. Thus, providing interfaces for *ConcreteRule* and *ConcreteRuleFactory*

is very important for optimizer flexibility. For optimizers with controlled and established search strategies and phases, the database developer may avoid using *SearchStrategyFactory* and *SearchStrategy.* In this case, one may use the optimizer bound to specific types of *ConcreteSearchStrategy*.

13. Known uses

In the following examples, the FoX project uses the complete DORS pattern. The frameworks Cascades [Gra95], Columbia [SMB01] and OPTGEN [Feg97] use the main abstractions (*Rule*, *RuleFactory*, *SearchStrategy,* and *SearchSpace*) in their implementations with similar structure and behavior. See bellow more details of each one of them.

- Cascades [Gra95] and Columbia [SMB01] are frameworks with top-down search engine that provides a set of classes to build optimizers. The framework uses the *DBI-Optimizer* interface hiding the concrete implementation of the optimizer. It provides a *Rule* class with methods that perform transformations, ordering of rules and conditions of use. The phases are implemented by the *Task* class, but the framework provides only the top-down approach for search strategies. The *Task* objects may use pipelining of processing. It uses the MEMO approach to implement the search space. The pattern matching mechanism is provided with the *Binding* class. *Guidance* objects are used to divide rules into rule sets.

- The OPTGEN [Feg97] framework provides a rule language (OPTL [Feg97]) for optimizer specification. The language is an extension of C++. With the OPTL language, the developer may declare rules with pattern matching, conditions (named guards) and the transformation of expressions. The language provides the declaration of rule modules (*rule factories*). The rules are processed in each module in the order they are declared. The language allows the declaration of several logical modules and one physical module. The framework uses top-down search engine with MEMO search space. The result of the OPTL program compilation is the C++ source code of the optimizer. The source code uses C++ classes to implement the specification declared in the OPTL program.

- In [PM01] and [Pir01] the OPTGEN framework wad modified to add the support for optimization for parallel object-oriented databases. The modified framework used a three-phase optimization model with top-down search engine for the first two phases and a simple substitution search engine for the third phase. It also used in the first phase a sequential logical algebra, and in the second and third phases it used two different physical parallel algebras. The rules for parallel algebras were built according to an analysis pattern.

- The FoX Data Server Project in development at Federal University of Ceará uses the DORS pattern to design and build a flexible optimizer for XML data.

Acknowledgements: We would like to thank our shepherd Federico Balaguer and our writers workshop group (F. Montero, Joe Yoder, Marcos D´Ornellas, Savitha Muthanna) for their valuable suggestions and comments on this paper.

References

[CB97] R. Cattel, D. Barry, editors. *The Object Database Standard: ODMG 93*. Morgan Kaufman, 1997.

[Feg97] L. Fegaras. *The OPTGEN Optimizer Generator. Department of Computer Science and Engineering*. The University of Texas at Arlington, http://ranger.uta.edu/~fegaras/optimizer, 1997.

[FSRM00] L. Fegaras, C. Srinivasan, A. Rajendran, D. Maier. *Lambda-DB: An ODMG-Based Object-Oriented DBMS*. ACM SIGMOD International Conference on Management of Data, Dallas, Texas, 2000.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Gra95] G. Graefe. *The Cascades Framework for Query Optimization*. Bulletin of the IEEE Technical Committee on Data Engineering, 18(3), Pages 19-29, September 1995

[ISO96] ISO/IEC JTC1/SC21 N10489, ISO//IEC 9075, Part 2, Committee Draft (CD), *Database Language SQL - Part 2: SQL/Foundation,* ftp://speckle.ncsl.nist.gov/isowg3/dbl/BASEdocs/cd-found.pdf, July, 1996.

[MS94] M.L.Q. Mattoso, J.M. Souza, *GOA: Um Servidor de Objetos Persistentes para Sistemas de Banco de Dados Orientados a Objetos* (in portuguese), XX Conferência Latinoamericana de Informática, Mexico City, Mexico, September 1994.

[Pir01] C. G. Pires. *Optimizer Development with Query Rewrite for Parallel Object-Oriented Database Management Systems* (in portuguese). Master's thesis, Federal University of Ceará, September, 2001.

[PM01] C. G. Pires, J. C. Machado. *Applying Rules for Partitioned Parallelism in OODBMS within an Optimizer Generator Framework.* XVI Brazilian Symposium of Database, SBC, Rio de Janeiro, 2001.

[SMB01] L. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H. Wu, B. Vance. *Exploiting Upper and Lower Bounds In Top-Down Query Optimization*, IDEAS 2001*,* Grenoble, France*, 2001.*

The AbstractOptimizer[1]

Savitha Muthanna,
savitha_muthanna@agilent.com
Agilent Technologies Inc.
24001 E. Mission Avenue,
Liberty Lake WA 95019-9599
U.S.A

Abstract

Several times there is the need to further optimize aspects of a design that might result in better performance for the program or application. In the realm of real-time embedded systems, optimizing the number of tasks executing in the system would be a major optimization. This paper presents a design pattern, the AbstractOptimizer, which provides a way to achieve this optimization in a simple and elegant manner.

1. Motivation

Test and Measurement instrument boxes, present useful measurement results to the user, after performing a series of measurements on a device under test. Examples are embedded real-time instruments that test the Radio Frequency (RF) performance of a cellular phone. These are devices that can be extremely complicated and can use a complicated combination of hardware and firmware to simulate a real-time cellular network.

Consider the User Interface of such an instrument, illustrated in Figure 1. This will have a CRT displaying several measurements. In Figure 1, 'Waveform Quality Measurement' is a viewing area or a Screen. 'Rho', 'Frequency Error', 'TimeError', 'Continuous' etc. give the user different pieces of information that are useful to her. 'Rho', 'Frequency Error' and 'Time Error' describe the results of measurements. 'Continuous', for example, describes a setting that applies to the 'Waveform Quality' measurement.

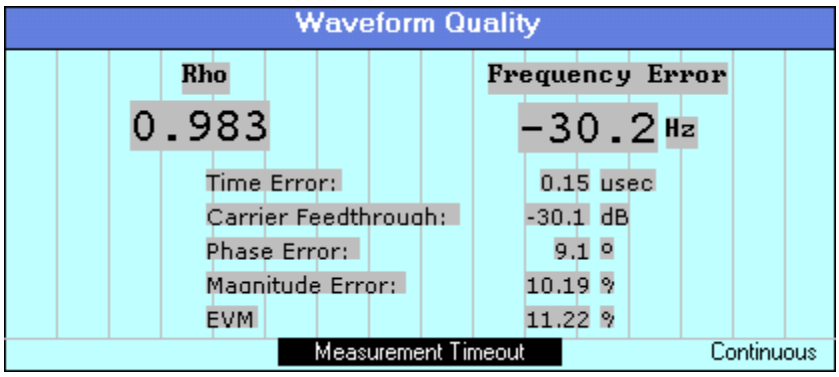


Figure 1: Instrument User Interface

Figure 2 illustrates an object-oriented design of such a system.

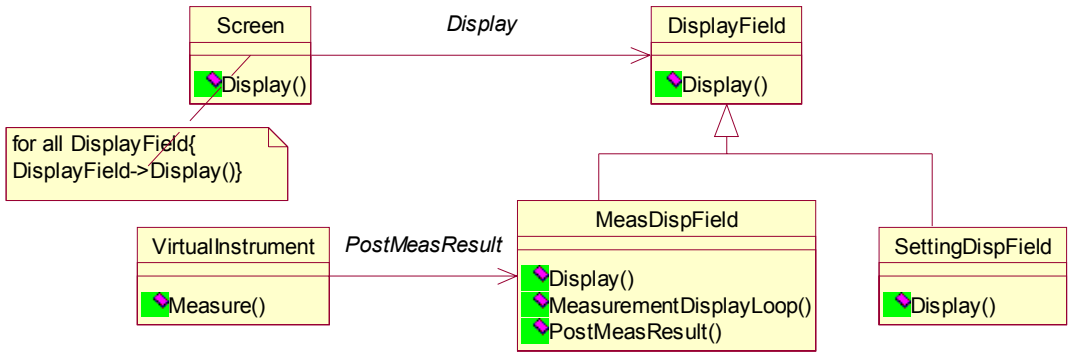


Figure 2: Classes and relationships.

The Screen is a class containing one or more DisplayFields. Each DisplayField will display one measurement result (MeasDispField), e.g.: 'Frequency Error' in Figure 1. A DisplayField can also represent a setting that the instrument is set to (SettingDispField), e.g.: 'Continuous'.

Each MeasDispField is connected to a VirtualInstrument(VI) which is an abstraction of a small piece of functionality, one of the mini instruments in this box. A VI computes a set of measurements results. Each MeasDispField represents **one** measurement result sent by **a** VI. Also, each MeasDispField is executed by an independent thread and encapsulates a fairly complicated measurement display loop that continually looks for the most recent measurement result posted by the VI and displays this result at a given display rate. The MeasDispField is pivotal to this problem context. The VI periodically posts measurement results at a much higher rate than the MeasDispField can display it.

Applying this design to implement the Screen described in Figure 1, we will have the following: The Screen will be represented by a Screen instance. The "Rho", "Frequency Error" etc. will be represented by MeasDispField instances. Each one of these MeasDispFields is associated with a measurement result associated with a VI instance.

So, here's the problem: Of the MeasDispFields displayed on a Screen, some of them could be results associated with the same VI. Also, each MeasDispField has one task associated with it. The purpose of this task is to allow the MeasDispField to display one real-time measurement result. The problem that needs to be solved in this case is, to optimize the design of the Screen and DisplayFields to reduce the number of tasks running to display results on a single screen.

Also, if this test and measurement box is a platform then it will be capable of supporting various incarnations. E.g., it needs to be capable of supporting various wireless protocols in each incarnation to test various types of cellular phones. Therefore, once the Screen and DisplayFields classes have been designed, multiple instances of Screen and DisplayField objects will need to be instantiated for the several measurements for each incarnation of this box, for each wireless protocol. This pattern must allow several Screen and DisplayField objects to be configured and instantiated without any knowledge of the details of this optimization.

The AbstractOptimizer pattern in this context allows the number of tasks associated with a Screen to be optimized, with a clean design and prevents any redundancy of critical code.

This paper introduces the concept of the "application developer" to further highlight the advantages of using this pattern (The Consequences section discusses the advantages in more detail). The term "application developer" refers to the individual(s) responsible for creating the DisplayField and Screen objects. The application developer may or may not be the same person as the designer of the DisplayField and Screen classes. The idea here is that, once the DisplayField and Screen classes are designed, they would reside in a library and the "application developer" would create any number of DisplayField and Screen objects to fit their application.

2. Context

You are building real-time embedded systems, and are doing a major optimization of the number of tasks executing in the system.

3. Problem

How can we design a set of reusable classes for real-time embedded systems that improve performance by optimizing the number of tasks associated with the system and protect the application developer from knowledge about the optimization details?

4. Forces

- **Improved performance:** It is critical that optimization of a real-time embedded systems result in better performance.

- **Design Abstraction:** The application developer needs to be shielded from any knowledge of this optimization. In other words, this design should allow various instances of Screens and DisplayFields to be created for every application, without having to know about the existence of the AbstractOptimizer or having to create instances of the Abstract Optimizer. The application developer is therefore completely shielded from any knowledge or use of the AbstractOptimizer.
- **Reusability:** Design Reusability across various incarnations of the test instrument is a necessity. Allowing optimization code to be re-used in a library-like manner and be untouched for various incarnations of the test instrument can provide for this reusability.
- **Eliminate redundancy:** This may be obvious, but in redesigning to optimize areas of the design and move functionality around, ensure that the code is not redundant.

5. Solution

In the context of the problem described above, the solution of this problem is to introduce a set of classes to that allow the DisplayFields to be grouped based on whether they are associated with the same VirtualInstrument or not. This will allow all the DisplayFields associated with the same VirtualInstrument to be executed by one task and still have a real-time display of the associated measurement result. Figure 3 describes the solution.

6. Structure

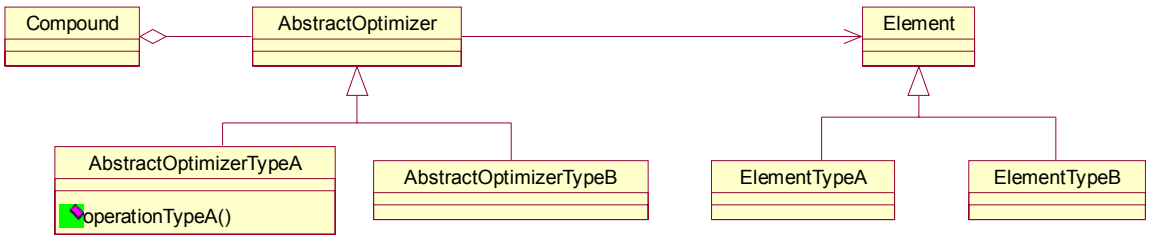


Figure 3: *Classes and relationships in the solution*

7. Participants

- **Compound**: The Compound class contains one or more instances of the AbstractOptimizer. The Compound, in its constructor, actually creates one or more instances of the AbstractOptimizer. It is passed the list of Elements (which have already been created) into its constructor. It then maps the Elements to each AbstractOptimizer by using properties of Elements that allow them to be grouped into exclusive sets. Before this pattern is applied or used, the Compound contains one or more instances of the Elements instead of one or more instances of the AbstractOptimizer. In problem context described earlier, the 'Screen' class is the Compound.

- **AbstractOptimizer**: The AbstractOptimizer is an abstract base class, that represents common functionality shared by all optimizers. It will contain one or more instances of Elements. It provides an interface to not only encapsulate functionality common among Elements but also offers significant optimization. It allows mutually exclusive groupings of Elements, with optimization in mind.
- **AbstractOptimizerTypeA**: The AbstractOptimizerTypeA is a type of AbstractOptimizer that encapsulates functionality, previously (before applying this pattern) encapsulated by ElementTypeA. Therefore, the behaviors exhibited by one or more Elements (which existed as part of ElementTypeA) will now be encapsulated inside AbstractOptimizerTypeA, Similarly for AbstractOptimizerTypeB. For e.g., 'operationTypeA()' is a method belonging to the 'AbstractOptimizerTypeA' class. Before the application of this pattern it would have been part of the ElementTypeA class. It further specializes in encapsulating behaviors that are common to one or more Elements.
- **Element**: Every AbstractOptimizer contains one or more Element classes. The Elements are generally groupable into exclusive sets of common attributes/behaviors. In the problem context described earlier, the 'DisplayField' is the Element.

8. Collaborations

Figure 4 below, describes one set of collaborations, between the different classes in this pattern. The Compound class requests that "OperationTypeA" be performed on all its constituent AbstractOptimizers. Each AbstractOptimizer, executed by one task, requests its constituent Element classes to perform "Operation TypeA".

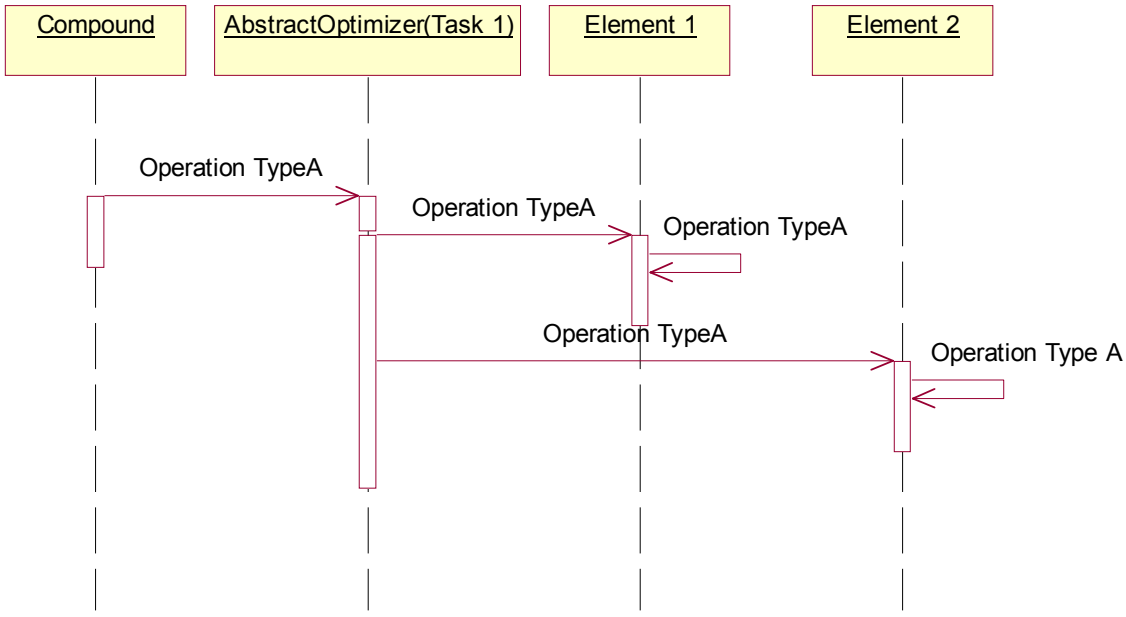


Figure 4: *Dynamic interactions of the pattern objects*

Please see the section on implementation, for depth and detail on the collaborations that are introduced upon application of this pattern on the problem described in the motivation section.

9. Consequences

- This pattern will cause a definite improvement in performance of the system that is being optimized, by reducing the number of tasks associated with the system. An added cost in terms of memory consumption however, is that a few more instances of a class are added to the system if that is a consideration.
- It completely shields the application developer from the knowledge of its existence, because it is the Compound's responsibility to actually create the AbstractOptimizer. Say, that once designed, the set of classes described above reside in a library and it is now the responsibility of the application developer to instantiate Compounds and Elements, that are required for their application and configure them correctly. The application developer will not know of the existence of the AbstractOptimizer hierarchy of classes, since it is the Compound's responsibility to create the AbstractOptimizer and map its constituent Elements to each AbstractOptimizer by gleaning the required information from the Elements.
- This pattern is very useful where the Element and Compound classes need to be designed and then reside in a library. The application developer only needs to instantiate the Compound and the Element classes based on the requirements of her system. This will make automatic optimization most desirable. This is especially true, if there are a large number of Compound and Element classes to create, which makes the task of manually grouping together Elements forbidding. However, it is easily applicable and useful in situations where a library is not used to house the Compounds and Elements. The objective of optimization by using this pattern is achievable whether a library-model is used or not.
- Optimization is the primary purpose of using this pattern. There is always a cost if a design is over-optimized. Therefore the designer needs to evaluate the need for optimization before applying this pattern to their context. For example, at one end of the spectrum, if it happened in the above example that, every MeasDispField was associated with a different VI, there may not be any need for optimization and the Compound may as well contain a list of Elements. On the other end of the spectrum, every MeasDispField on the Screen could be associated with the same VI. If we then employ our pattern, we have one Compound that includes one AbstractOptimizer which in turn contains all the Elements.

10. Implementation

An important aspect in the implementation of this pattern, is the identification and definition of the AbstractOptimizer and the determination of its constituent Element classes. The following points are meant to serve as a guideline in going about this task:

- Identify what the "common functionality" is that will allow objects to be grouped together and optimized. Let us call this "groupable property".
- Next, design the AbstractOptimizer hierarchy for your context.

- Instantiate the AbstractOptimizer during the construction of the Compound class. The construction of the AbstractOptimizer will utilize this "groupable property" to know how to group various Elements and associate them with one AbstractOptimizer.
- Redesign the old Element classes to move the common behavior of these objects to the Abstract Optimizer, which will now apply this behavior on a group of Elements rather than one Element.

Applying this pattern to the context previously discussed (in the Motivation section), the solution will look like the set of classes in Figure 5.

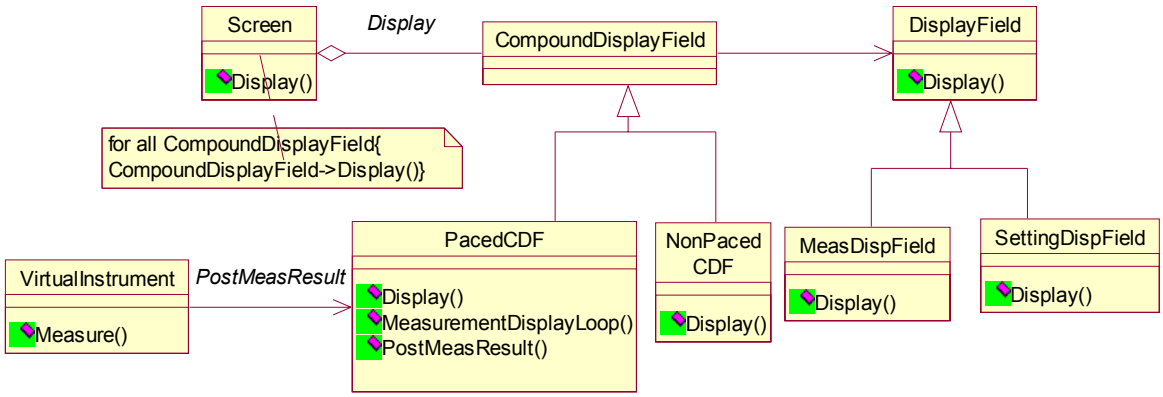


Figure 5: *Design Pattern applied on the example problem*

The Screen class (Compound) now contains one or more CompoundDisplayFields (AbstractOptimizers). The CompoundDisplayField provides an abstract interface to not only encapsulate functionality common among DisplayFields (Elements) but also offers significant optimization. It allows for mutually exclusive groupings of DisplayFields and its purpose is to achieve optimization.

Each CompoundDisplayField object is associated with or more DisplayFields. Each CompoundDisplayField can be of two types, Paced CompoundDisplayField (PacedCDF) and Non-Paced CompoundDisplayField (NonPacedCDF), to accommodate the optimizations associated with the two variations of DisplayFields. The PacedCDF will now contain the 'MeasurementDisplayLoop()' operation, which paces every so often looking for posted measurement results sent by a Virtual Instrument. This operation was previously contained by the MeasDispField class. The PacedCDF therefore contains one or more MeasDispField objects. The NonPacedCDFs on the other hand contains the SettingDispField objects.

Let us examine the collaborations introduced by this re-designed set of classes, upon application of this pattern. The collaborations between the different classes work as described in Figure 6 below. The Screen class requests all its constituent PacedCDFs to display themselves. Each PacedCDF, associated with one task, wakes up, subscribes to a measurement result from the VI and waits for a measurement result. When it receives a

measurement result, it then asks its constituent MeasDispFields to display themselves. Each MeasDispField then extracts the relevant portion of the measurement result and writes the pixels on the physical display.

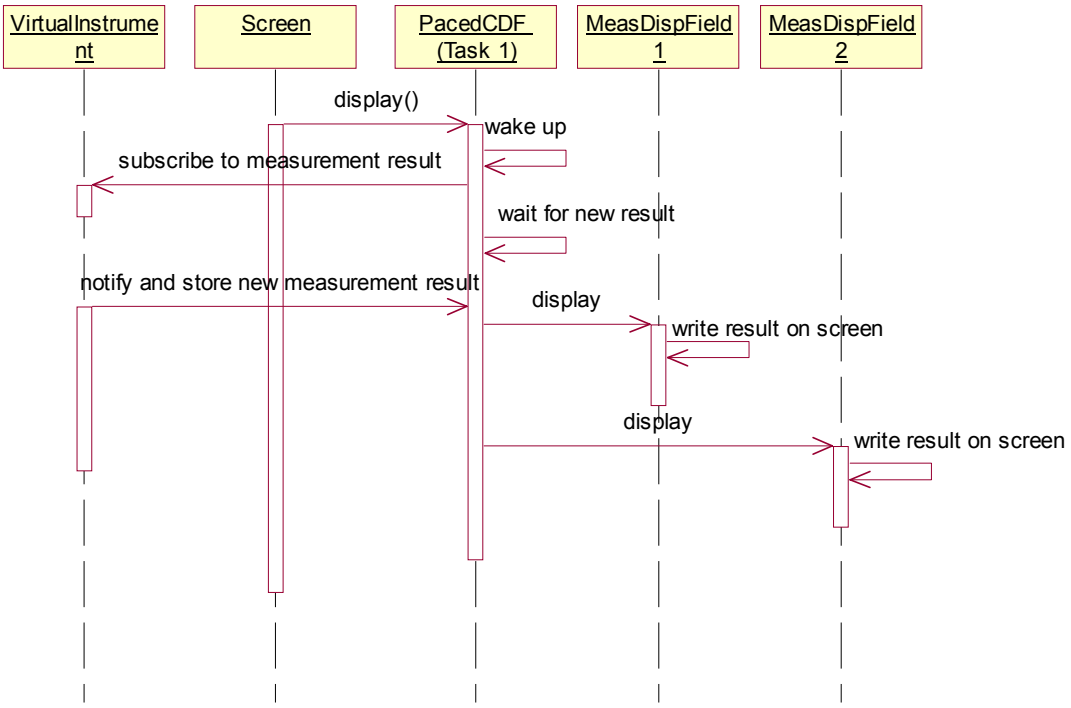


Figure 6: *Dynamic interactions of the classes in the solution after application of the pattern.*

Figure 7 illustrates how the collaborations worked before application of the pattern, highlighting the performance boost introduced by the pattern. The Screen class requests that all its constituent MeasDispFields, each of them associated with its own individual task to display itself. Each task wakes up, subscribes to results from a VI, waits for a measurement result to be sent by the VI. The VI then notifies the MeasDispField of the new measurement result and also stores this result. The MeasDispField then displays the measurement result on the physical display.

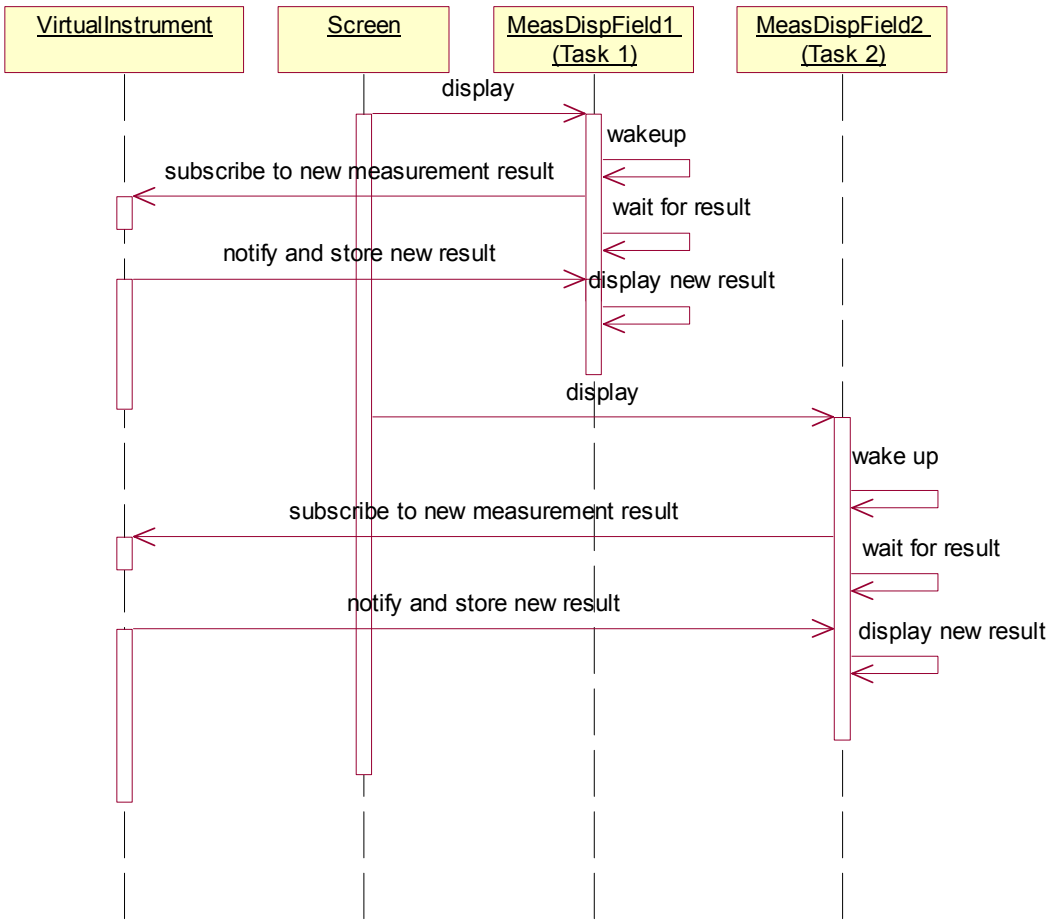


Figure 7: *Dynamic interactions of the classes before the application of the pattern*

11. Sample Code:

Here is some sample code, in C++, that illustrates the creation of the CompoundDisplayField objects inside the Screen constructor.

CompoundDisplayField objects (PacedCDF and NonPacedCDF objects) are created as follows: Each DisplayField is examined by determining if it is a MeasDispField and if so, all MeasDispFields associated with the same VI will be associated with one PacedCDF. Each DisplayField that is not a MeasDispField (perhaps a 'SettingDispField') will be associated with a Non-PacedCDF.

The list of DisplayFields associated with a Screen is passed to Screen at construction time. Each DisplayField will contain attributes that are all programmed into it (during its construction) that determine whether the DisplayField is a MeasDispField or

not. It will also contain a pointer to the VI that it is associated with. Obviously the DisplayFields are created before the Screens are constructed.

```
// Screen class interface
Class Screen{
Public:
   Screen(DisplayField *);
   ~Screen();
   void Display();

Private:
   AddDisplayFieldToCDF(DisplayField *);
   AddDisplayFieldToNonPacedCDF(DisplayField *);

   CompoundDisplayField *compound_df[MAX_NUM];
}

// The Screen class Constructor.
Screen::Screen(DisplayField  *list_of_displayField)
   {
   DisplayField *temp = list_of_displayField;
   Int32 num_of_CompoundDisplayFields;

   // Construct the associated CompoundDisplayField objects.

   /*
    NOTE: Before applying this pattern, the list of DisplayField
    pointers would simply have been stored away.
   */

   while (temp != NULL)
      {
      if (temp->typeQ() ==  MEASDISPFIELD)
         {
         // If a CompoundDisplayField  that is associated
         // with this VI does not exist, then create a new
         // PacedCDF.
         if (!CompoundDisplayFieldExists(temp))
            {
            // Create new PacedCDF
            compound_df[num_of_CompoundDisplayFields] =
                                       new PacedCDF();

            num_of_CompoundDisplayFields++;
            }
         else
            {
            // Add this DisplayField to the list of
            // DisplayFields associated with an existing
            // CompoundDisplayField.
            AddDisplayFieldToCDF(temp);
```

```
            }
         }
   else
      // Add this DisplayField to the list of NonPacedCDFs.
      {
      AddDisplayFieldToNonPacedCDF(temp)
      }

   // Get the next DisplayField.
   temp++;
   }
```

Next, sample code has been included below to illustrate a simplified version of the "MeasurementDisplayLoop()" mechanism of the PacedCDF, after application of this pattern.

The Screen class can be asked to "display" the Screen. Upon receiving the request to display, Screen will request all its associated CompoundDisplayFields to display themselves in turn.

```
Void Screen::display()
   {
   for (index = 0; index < MAX_NUM; index ++)
       {
        if (compound_df[index] != NULL)
           {
           compound_df->display();
           }
       }
   }
```

When the CompoundDisplayField is asked to display, if the CompoundDisplayField is a PacedCDF, it will have a task associated with it, that is blocked pending a message that will be woken up if asked to display. Before applying this pattern, the MeasDisplayLoop was executed by one task associated with each MeasDispField. After applying the pattern, only the PacedCDF has one task that executes the MeasDisplayLoop and simply asks its constituent MeasDisplayFields to display themselves.

The following piece of code illustrates the definition of the PacedCDF class and the mechanism of displaying the PacedCDF and while doing so, all the MeasDispFields associated with this PacedCDF.

```
// PacedCDF inherits from CompoundDisplayField.
Class PacedCDF::public CompoundDisplayField
   {
   public:
     PacedCDF();
     ~PacedCDF();
```

```
      virtual void display();

   private:
      const int MAX_NUM_OF_DISPLAY_FIELDS = 10;
      void executeMeasDisplayLoop();
      void measDisplayLoop();
       void initializeLoop();
       void startTimer();
       void waitForEvent(semaphore);
       void notifyResult(MeasurementResult *);

       // A pointer to the associated VI.
       VirtualInstrument *VI;
       MeasDisplayField
              *list_of_display_fields[MAX_NUM_OF_DISPLAY_FIELDS];

       // To contain the results from the VI.
       MeasurementResult *result;

      // The state of the PacedCDF, i.e., whether displayed
      // or not.
      DisplayState state;

      // The task associated with this Paced CDF, uses this
      // queue to //communicate new display requests.
      MsgFifoQueue mesg_queue;

      // This semaphore indicates the presence of a new
      // measurement //result posted by the VI.
      Semaphore vi_result_arrived_semaphore;

}
```

The following piece of code illustrates the workings of the critical "measurementDisplayLoop". First, the 'display()' function of the 'PacedCDF' class wakes up the task, by sending a message indicating that the PacedCDF now needs to be displayed. The 'executeMeasDisplayLoop()' function is then executed. This function receives the message on the queue, and kicks off the execution of the 'measurementDisplayLoop()' function by the task.

```
void PacedCDF::display()
   {
   // Set the state of this PacedCDF to "displayed".
   State = DISPLAYED;

   // Send a message on a queue to wake up the task associated
   // with this PacedCDF.
   msgFifoQSend(msg_queue, this, OSI_WAIT_FOREVER);
   }

// The PacedCDF task is waiting for a message to arrive.
```

```
// When it does, it calls the 'measDisplayLoop' function.
void PacedCDF::executeMeasDisplayLoop()
   {
   msgFifoQReceive(msg_queue, this, OSI_WAIT_FOREVER);

   this->measurementDisplayLoop();
   }
```

The 'measurementDisplayLoop()' function does some initialization, starts a timer to control the display rate of the measurement result. It then subscribes to the VI for a new measurement result and waits for the result. The VI then executes the 'notifyResult()' function to notify of the arrival of a new result and stores this result. The PacedCDF now requests all its constituent MeasDispFields to extract their portion of the measurement result and display it on the Screen.

```
// The critical "MeasurementDisplayLoop" function that subscribes
// to the measurement result of interest and waits for the result
// from the VI.
void PacedCDF::measurementDisplayLoop()
   {

   // do some initial setup
   initializeLoop();
   while (state == DISPLAYED)
      {
      // kick off a timer, to ensure that display happens
      // every so often.
      startTimer();

      // Subscribe to the measurement result from the VI.
      VI->subscribeToMeasurement();

      // Wait for the result from the VI.
      waitForEvent(vi_result_arrived_semaphore);

      // At this point we have the result from the VI. The VI is
      //a concurrent task that has deposited the results, in a
      // member variable of this class and then has released
      // the "vi_result_arrived_semaphore". Now request all the
      // associated MeasDispFields to display the result.
      for(index=0; index <  NUM_OF_DISPLAY_FIELDS; index++)
         {
         list_of_display_fields->display();
         }
      }
   }
```

The following piece of code describes the 'notifyResult()' function which is executed by the VI, to notify the task executing the 'measurementDisplayLoop()' that there is a new result available and to store this result.

```
void PacedCDF::notifyResult(MeasurementResult *result)
   {
   storeResult(result);
   signalEvent(vi_result_arrived_semaphore);
   }
```

The following piece of code describes the 'display()' function of the MeasDisplayField class that simply writes the measurement result on the physical display.

```
void MeasDisplayField::display(MeasurementResult *result)
   {
   videoDisplay(result[this_measurement_index]);
   }
```

12. Related Patterns:

Observer: The PacedCDF can obtain periodic measurement results to display from the VI, using the Observer pattern. It can use a subcriber-notify model to susbscribe to measurement results and periodically obtain results that need to be displayed by the task associated with the PacedCDFs.

State: The State pattern can be associated with the PacedCDFs. The behavior of the PacedCDF object is dependent whether it is displayed or not.

AbstractFactory: Creation of the DisplayFields and Screens associated with each incarnation of a test and instrument box, can use the AbstractFactory pattern to create instances of it.

13. Known Uses:

Used in the design of one of the sub-systems inside the User Interface of the Agilent 8960 Series 10 for Wireless Manufacturing.

14. Acknowledgements:

Many thanks are due to my shepherd Robert Hanmer for his thought-provoking comments that helped to refine this paper. I am also deeply grateful to Agilent Technologies for providing the time and resources required to author and present this paper at the SugarLoaf PLOP 2002. I would like to thank Joseph Yoder, Program Co-chair, SugarLoaf PLOP 2002, who repeatedly reviewed and helped revamp this paper. I would also like to thank the participants of my Writer's workshop group "hushy" at SugarLoafPLOP 2002, Carlo Giovano S. Pires, Francisco Montero and Marcos Cordeiro

d'Ornellas for their comments and criticisms that have helped get this paper into a much better shape. Finally I would like to thank the organizers of the SugarLoaf PLOP 2002 for making this a fun and rewarding experience for me.

15. References:

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading MA:Addison-Wesley, 1995.

[2] James O. Coplien and Douglas C. Schmidt(Editors).*Pattern Languages of Program Design. Addison-Wesley (Software Patterns Series), 1995.*

[3] Martin Fowler, Scott Kendall. *UML Distilled: A brief guide to the standard Object Modelling language*, Second Edition, Addison-Wesley, 2000.

[4] The Agilent 8960 Series 10 for Wireless Manufacturing. http://we.home.agilent.com/cgi-bin/bvpub/agilent/Product/cp_Product.jsp?NAV_ID=-11874.536881883.00&LANGUAGE_CODE=eng&COUNTRY_CODE=ZZ

[5] David Stepner, Nagarajan Rajan, David Hui . *Embedded Application Design Using a Real-Time OS*. Proceedings of the 38th ACM/IEEE conference on Design automation conference. June 1999.

Um *Design Pattern* para Configuração de Arquiteturas de Software

Jonivan Coutinho Lisbôa (jlisboa@ic.uff.br)
Sérgio Teixeira de Carvalho (sergiotc@ic.uff.br)
Orlando Gomes Loques Filho (loques@ic.uff.br)

Universidade Federal Fluminense – Instituto de Computação
Rua Passo da Pátria 156 – Bloco E – 3º. Andar
Boa Viagem 24210-240 Niterói-RJ - Brasil

Resumo

*Este trabalho apresenta a descrição de um design pattern, chamado **Architecture Configurator**, que modela o processo de implantação da configuração de um sistema de software. Os patterns são meios utilizados para documentar situações recorrentes em desenvolvimento de software. Sua utilização no estudo de implantação de arquiteturas visa facilitar o estudo comportamental de um sistema, sem que o projetista precise ater-se a detalhes de implementação. Além disso, o uso de patterns possibilita obter alguns requisitos desejados na implementação de arquiteturas, como separação de interesses, reutilização de componentes, facilidade de manutenção e extensão do sistema, entre outros.*

Palavras-chave : design patterns*, arquiteturas de software, configuração de software.*

Abstract

*This work presents the description of a design pattern called **Architecture Configurator**, that models the establishment of software system configuration. Patterns are used to document recurrent situations in software development. The use of patterns for studying architecture establishment aims to make easy the catch of system behavior with no need by the designer to rely on implementation details. Moreover, using patterns makes possible to acquire some desired features in architecture implementation, like separation of interests, component reuse, maintenance and extension easiness, and more.*

Keywords : design patterns, software architecture, software configuration.

1. Introdução

A concepção de um sistema de software parte da definição de uma arquitetura, que o descreve em termos dos componentes que o integram – os módulos e conectores – e das ligações feitas entre eles, através de pontos de interação específicos – as portas.

A descrição de uma arquitetura pode ser realizada de maneira formal, através de uma linguagem de descrição arquitetural (ADL – *Architecture Description Language*). O produto da descrição arquitetural de um sistema é a sua **configuração**, ou seja, a estrutura topológica da aplicação. Na configuração, estão definidos os pontos de interação de cada módulo e cada conector, e também a maneira como os componentes irão interagir entre si – as

ligações entre eles. A configuração é abstrata, e deve ser implementada mediante a criação das instâncias dos componentes, e a realização das ligações especificadas.

Quando bem definido, o projeto de uma arquitetura pode fornecer um nível de abstração que permite a análise do comportamento do sistema como um todo, sem a necessidade de se conhecer detalhes de implementação. Para conseguir isso, pode ser útil o reaproveitamento de experiências anteriores na implantação de arquiteturas de software. Isso é possível através da utilização de *patterns*, que são meios de se documentar situações recorrentes em desenvolvimento de software.

Este artigo apresenta uma descrição do *design pattern* ***Architecture Configurator***, que fornece uma base para a implementação de configurações arquiteturais. Para isso, fundamenta-se nos mecanismos de interceptação, encaminhamento e manipulação de requisições realizadas entre os componentes do sistema, e também na interligação entre eles [Car01].

Em linhas gerais, o ***Architecture Configurator*** privilegia a reutilização de software, através da separação de interesses (requisitos funcionais e não-funcionais). Podem ser entendidos como requisitos funcionais aqueles que descrevem a funcionalidade do sistema, ou seja, representam os serviços oferecidos pelo mesmo. Já os requisitos não-funcionais envolvem propriedades desejadas para o sistema (por exemplo, confiabilidade, tempo de resposta, capacidade de sincronização) [Som00]. Na configuração, os requisitos funcionais podem ser encapsulados nos módulos (chamados também de módulos funcionais), enquanto que os requisitos não-funcionais podem ser encapsulados pelos conectores.

O ***Architecture Configurator*** foi proposto quando se observaram alguns padrões de recorrência encontrados tanto na implementação de arquiteturas quanto na implementação dos conectores [Car01]. Independentemente de sua funcionalidade, os conectores interceptam as requisições de serviços e respostas, examinam e manipulam tais requisições, e encaminham-nas a seus respectivos destinos. Por sua vez, a configuração arquitetural possui como pontos de recorrência os seguintes fatos: módulos e conectores possuem portas; as portas utilizadas na interação entre dois módulos devem ter o mesmo tipo; um módulo interage com outro através de um ou mais conectores; um conector pode ser conectado diretamente a outro conector.

Na configuração, os conectores podem ser encadeados, formando uma rede, com o objetivo básico de interceptar as requisições e respostas vindas de módulos funcionais. A característica de interceptação dos conectores e o encadeamento dos mesmos, feito durante a implantação da configuração, formam a base para a implementação de arquiteturas, e, por conseguinte, do ***Architecture Configurator***.

2. Descrição

A descrição de *patterns* deve seguir o formato Contexto-Problema-Solução, apresentando o contexto no qual o problema deve ser tratado, descrevendo o problema em si, e apresentando a solução empregada. Existem vários formatos padronizados de descrição, nos quais podem ocorrer diferenças quanto aos elementos apresentados. Segundo os *patterns* *Mandatory Elements Present* e *Optional Elements When Helpful* [MRB97], existem alguns elementos considerados obrigatórios, enquanto outros são opcionais. Os obrigatórios são: Nome, Contexto, Problema, Forças e Solução. Alguns opcionais : Consequências, *Patterns* Relacionados, Usos Conhecidos, dentre outros.

Para a descrição a seguir, foi utilizada uma combinação de elementos de dois formatos: o formato canônico, ou formato de Alexander [App00] e o padrão GoF, de *Gang of*

Four, referência aos autores do primeiro catálogo a ter aceitação como uma forma padronizada para descrição de *patterns* [GHJ95]. Do primeiro foram tomados os elementos Problema, Contexto, Forças, Solução e Exemplo, e do segundo, os elementos Objetivo (ou Intento), Estrutura e Participantes, Colaborações e Consequências. Os elementos Nome, *Patterns* Relacionados e Usos Conhecidos são comuns aos dois formatos de descrição.

2.1. Nome

Architecture Configurator.

2.2. Objetivo

O *Architecture Configurator* modela o processo de implantação da configuração de um sistema, obtida mediante a descrição arquitetural do mesmo.

2.3. Contexto

A arquitetura de uma aplicação envolve módulos e um ou mais conectores. Os módulos interagem uns com os outros, requerendo e/ou fornecendo serviços, e os conectores intermediam essa interação. A interação entre módulos e conectores ocorre através de pontos específicos: as portas. Esses três elementos (módulos, conectores e portas) formam a base de uma arquitetura de software.

Neste contexto, um módulo refere-se a um processo, objeto, *procedure*, ou qualquer pedaço de código ou dados identificável. Um conector possui semelhança com um módulo, pois também possui interfaces e serviços definidos, e sua funcionalidade também é representada por classes, métodos e procedimentos; porém, funcionalmente comporta-se como um interceptador e encaminhador de requisições, pois seus métodos são específicos para tais fins. As portas podem ser de dois tipos: portas de entrada, que representam serviços oferecidos, e portas de saída, que representam invocações de métodos.

Genericamente, um sistema de software pode ser composto por módulos que oferecem algum tipo de serviço (servidores) e módulos que utilizam serviços oferecidos (clientes). Então, os sistemas podem ser descritos como uma aplicação cliente-servidor, com clientes invocando métodos dos servidores, para realizar alguma tarefa. A figura 1 ilustra uma possível arquitetura para uma aplicação cliente-servidor simples, na qual os módulos cliente e servidor têm a interação entre si intermediada por um conector.

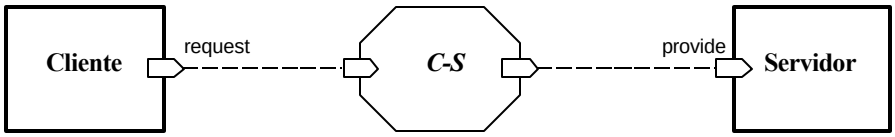


Figura 1 Arquitetura de um sistema cliente-servidor simples. Os módulos Cliente e Servidor têm sua interação intermediada pelo conector C-S. Observem-se as portas *request* e *provide* que indicam, respectivamente, uma invocação de método (porta de entrada) e um serviço oferecido (porta de saída).

A aplicação de uma arquitetura de software a um sistema supõe que a topologia do mesmo possa ser descrita de maneira formal, através de uma linguagem de descrição arquitetural (ADL – *Architectural Description Language*). Na descrição arquitetural do sistema estão presentes a descrição de cada módulo e cada conector, com suas respectivas

portas. Cada elemento da arquitetura (módulos, conectores e portas) é definido como um tipo, e são criadas instâncias de módulos e conectores. As ligações entre as instâncias também estão definidas de modo apropriado.

O código 1 mostra a definição da arquitetura ilustrada na figura 1, feita na ADL Babel [Mal96]. Nas linhas 2 e 3 são definidas as portas presentes na aplicação. Nas linhas de 5 a 11 são definidos os tipos para os módulos Servidor e Cliente, sendo declarada uma instância para cada um, nas linhas 7 e 11. Nas linhas de 13 a 16 é definido o conector, com sua instância. Na linha 18 as instâncias são criadas, e na linha 19 é especificada a ligação entre os módulos.

```
1     module ClienteServidor {
2           port Request;
3           port Provide;
4
5           module Servidor {
6                 inport Provide;
7           } servidor;
8
9           module Cliente {
10                outport Request;
11          } cliente;
12
13          connector C-S {
14                inport Request;
15                outport Provide;
16          } cs;
17
18          instantiate servidor, cliente, cs;
19          link cliente to servidor by cs;
20
21    } cliente_servidor;
22
23    start cliente_servidor;
```

Código 1 Descrição da arquitetura do sistema cliente-servidor simples em Babel.

A definição da arquitetura é abstrata, e torna-se concreta no momento em que são criadas as instâncias de módulos e conectores, e são realizadas as ligações apropriadas através de suas portas. A forma concreta da descrição arquitetural é chamada de **configuração** do sistema, e nada mais é do que a situação topológica do mesmo em relação aos objetos que o compõem. O processo de implantação de uma arquitetura é chamado de programação da configuração.

A utilização de uma descrição arquitetural possibilita ao desenvolvedor um certo nível de abstração, além de fornecer ao projeto do sistema a propriedade de separação de interesses funcionais de interesses não-funcionais. Como consequência disso, são obtidas maiores facilidades na reutilização de módulos, extensão e manutenção do sistema. Além disso, a própria arquitetura traz consigo algumas propriedades relativas à interação entre os componentes do sistema, feita através de conectores.

2.4. Problema

Na implantação da configuração, o mecanismo que concretiza a troca de mensagens entre as instâncias de módulos e conectores deve ser estabelecido, conforme

definido na descrição arquitetural do sistema. Contudo, as propriedades da arquitetura em questão devem permanecer inalteradas, independentemente da topologia estabelecida para o sistema ao qual a arquitetura é aplicada.

2.5. Forças

Uma solução para o problema da implementação de arquiteturas deve considerar o seguinte:

1) As propriedades da arquitetura, como reutilização de componentes e conectores, separação de requisitos e abstração, devem ser preservadas na sua implementação.
2) Os componentes e conectores devem ser independentes quanto à sua definição. Esta ortogonalidade permite que modificações em componentes não afetem conectores, e vice-versa, facilitando a separação de interesses;
3) Os módulos de um sistema não devem ter suas interfaces e comportamentos modificados com o objetivo de adicionar requisitos não-funcionais ao mesmo; se não for assim, fica comprometida a facilidade de reutilização de módulos com mesma funcionalidade para aplicações diferentes;
4) Os módulos e conectores devem preferencialmente ser coesos quanto aos seus requisitos; isso torna possível que o projetista de software se desprenda dos detalhes de implementação, podendo ter uma visão abstrata do comportamento do sistema;
5) A especificação da interface dos módulos deve tratar preferencialmente dos requisitos funcionais dos mesmos e deve ser claramente definida; com isso, ficam bastante claros quais serviços são oferecidos por módulos servidores, e quais serviços são requisitados por módulos clientes;
6) Os conectores podem ser utilizados para estender a arquitetura quanto a novos serviços não-funcionais;

2.6. Exemplo

Considere-se um sistema cliente-servidor simples, como, por exemplo, a aplicação Produtor-Consumidor com *buffer* limitado (PCB). Ela é composta de dois módulos clientes (`Produtor` e `Consumidor`) e um módulo servidor (`Buffer`). Os módulos possuem, cada um, sua funcionalidade intrínseca (requisitos funcionais), e interagem entre si da seguinte forma: o `Produtor` produz itens requerendo o serviço específico para armazenar itens no `Buffer` (*`put`*); o `Buffer` armazena/recupera itens, fornecendo os respectivos serviços (*`put`*/*`get`*); o `Consumidor` consome itens, requerendo o serviço específico para retirar itens do `Buffer` (*`get`*).

A aplicação de uma arquitetura de software a esse sistema poderia ser feita da seguinte maneira: cada módulo cliente (`Produtor` e `Consumidor`) seria representado contendo uma porta de saída, que representa a invocação de um serviço (*`put`* pelo `Produtor` e *`get`* pelo `Consumidor`). O módulo servidor (`Buffer`) seria representado contendo portas de entrada, indicando os serviços fornecidos (*`put`*, *`get`* e *`estado`*). Além disso, poderia existir um conector (`Guarda`) para intermediar a interação entre os módulos, tratando da sincronização de acesso ao `Buffer` por `Produtor` e `Consumidor`. `Guarda` retarda a execução de `Consumidor` se `Buffer` estiver vazio e retarda a execução de `Produtor` se `Buffer` estiver cheio. A capacidade de sincronização pode ser considerado um requisito não-funcional. A figura 2 ilustra essa possível arquitetura.

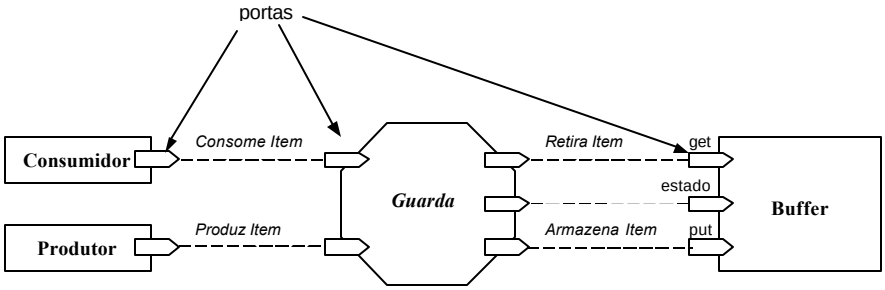


Figura 2 Arquitetura da aplicação Produtor-Consumidor com *buffer* limitado. Os componentes `Produtor`, `Consumidor` e `Buffer` agem como se o conector `Guarda` não existisse.

O conector intercepta, analisa e encaminha as requisições feitas por `Produtor` e `Consumidor` através de pontos de acesso específicos – as portas definidas nos módulos e nos próprios conectores. As portas representam serviços que poderão ser interceptados para o respectivo tratamento não-funcional. Tal interceptação é realizada de forma transparente em relação aos módulos envolvidos. Isto é, os módulos ignoram quaisquer mecanismos de intermediação existentes entre eles.

2.7. Solução

A implementação de uma arquitetura cliente-servidor pode ser feita segundo o diagrama apresentado na figura 3.

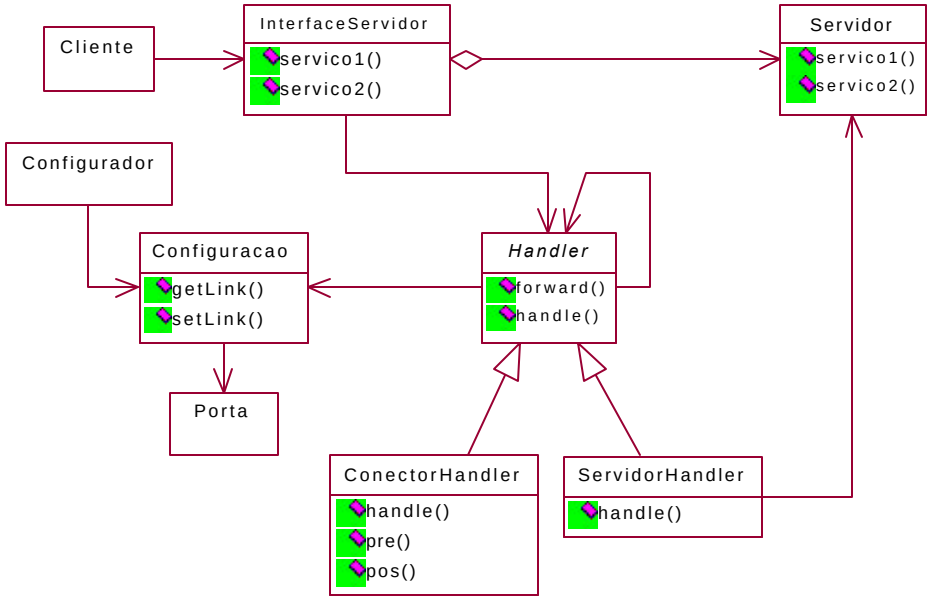


Figura 3 Diagrama de classes de uma possível solução para a arquitetura cliente-servidor utilizando *Architecture Configurator*

As classes `Cliente` e `Servidor` representam os módulos funcionais e a classe `ConectorHandler` representa o conector da arquitetura. A programação da configuração está representada pela classe `Configuracao`.

`Configurador` é uma classe que define, de um modo geral, um mecanismo que interpreta as instruções de uma ADL, e a partir da descrição interpretada, define os tipos para componentes, conectores e portas presentes na aplicação, e também procede com a instanciação e inteligação dos mesmos, para permitir a execução da aplicação. Para a realização de tais coisas, são invocados os serviços da classe `Configuracao`.

`Configuracao` recebe as solicitações de `Configurador` para realizar a configuração de componentes e conectores, e iniciá-los. Ela mantém duas categorias de informações: de descrição e de execução da arquitetura. Informações de descrição referem-se à arquitetura descrita através de uma ADL (código 1), enquanto as informações de execução relacionam-se às instâncias – referências – dos componentes e conectores durante o processo de execução da aplicação.

A classe abstrata `Handler` é responsável pelo encadeamento entre conectores e componentes, conforme a descrição arquitetural disponibilizada por `Configuracao`. `Handler` utiliza-se também das informações de execução da mesma classe, uma vez que necessita das referências aos objetos que representam componentes e conectores no espaço de execução da aplicação. No modelo, as classes `ConectorHandler` e `Buffer` são encadeadas por `Handler`. `ServidorHandler` representa a classe `Servidor` no encadeamento e possui uma referência a esta última.

As requisições invocadas por `Cliente` são interceptadas pela classe `InterfaceServidor`, que possui a mesma interface de `Servidor`. `InterfaceServidor` busca em `Configuracao` a referência ao conector `ConectorHandler` e invoca a operação *`handle()`* do mesmo.

Conforme a porta de entrada configurada no conector `ConectorHandler`, *`handle()`* invoca a operação adequada. Cada operação descrita no conector invoca *`forward()`*, responsável por dar seqüência ao encadeamento controlado por `Handler`. Na seqüência, `ServidorHandler` tem sua operação *`handle()`* solicitada, a qual encaminha a requisição original para `Servidor`, finalizando o encadeamento.

No exemplo de aplicação citado anteriormente (PCB), as classes `Produtor` e `Consumidor` têm o papel de `Cliente`, a classe `BufferReal` tem o papel de `Servidor`, a classe `InterfaceBuffer` tem o papel de `InterfaceServidor`, a classe `Guarda` tem o papel de `ConectorHandler` e a classe `Buffer` tem o papel de `ServidorHandler`.

A solução descrita apresenta algumas propriedades importantes, que conseguem absorver as forças apresentadas da seguinte maneira:

a) As funcionalidades de módulos e conectores estão separadas dos processos de interceptação e encaminhamento (realizados pelas classes `InterfaceServidor` e `Handler`, respectivamente), possibilitando a reutilização de componentes. Isso resolve parte da força 1.
b) A separação de requisitos é atendida, pois modificações funcionais ou de interface nos módulos não afetam os conectores, e vice-versa. Isso acontece porque são definidas classes distintas para módulos e conectores, nas quais as funcionalidades de cada módulo e conector são implementadas de forma bem definida e coesa – cabe ao conector, por exemplo, rotear as mensagens para os outros conectores e módulos

interligados. Além disso, a classe `Porta` mantém o mapeamento entre os pontos de interação dos componentes – as portas de entrada e saída – e as operações definidas nas classes e invocações a operações, independentemente das interfaces. Esses fatos resolvem parte da força 1, a força 2 e a força 4.

c) Os processos de interceptação e encaminhamento das requisições ocorrem de forma transparente, sem que os módulos do sistema tomem conhecimento disso. `InterfaceServidor` possui a mesma interface de `Servidor`, o que permite que `Cliente` utilize seus serviços como se estivessem lidando diretamente com ele. A solução, portanto, é independente da arquitetura apresentada. Tal independência pode ser estendida a qualquer tipo de arquitetura, oferecendo ao desenvolvedor um nível de abstração bastante elevado. A parte de abstração da força 1 é resolvida.

d) A definição de novos conectores, e sua inclusão apropriada no encadeamento mantido por `Handler`, torna possível a obtenção de novos requisitos não-funcionais, sem que seja preciso alterar a interface ou o comportamento dos módulos funcionais. Tal fato contribui para a extensão do sistema de uma maneira independente de sua topologia corrente. São atendidas as forças 3 e 6.

e) A classe `Configuracao` mantém informações de descrição e execução da arquitetura configurada. A descrição é composta basicamente pela definição das interfaces dos módulos e conectores e as ligações realizadas entre eles, enquanto que as informações de execução tratam de referências às instâncias dos objetos configurados. Nessa informação está contida a definição clara da interface dos módulos, e de como eles interagem entre si. Isso satisfaz a força 5.

2.8. Participantes

A figura 4 apresenta uma tabela, na qual é feito um resumo de todas as classes participantes no *Architecture Configurator*, suas responsabilidades e colaboradores.

Classes	Responsabilidades	Colaboradores
`Cliente`	- Utiliza a interface fornecida por Interface Servidor para requisitar um serviço particular;	`InterfaceServidor`
`Servidor`	- Implementa um ou mais serviços particulares;	-

`InterfaceServidor`	- Fornece a interface do servidor aos clientes;
	- Recupera a referência do conector interligado ao cliente no nível da configuração, ou do próprio servidor, caso não haja nenhum conector envolvido;
	- Repassa a solicitação do cliente ao conector recuperado, ou ao servidor, no caso de não haver conectores;
	- Retorna ao cliente resposta oriunda do servidor;
		`Servidor`
		`Configurador`
		`Conector Handler`

`Handler`	- Serve como classe abstrata base para o servidor e para os conectores; - Realiza o encadeamento de conectores e componentes a partir da configuração estabelecida;	`Configuracao`
`ConectorHandler`	- Implementa serviços relacionados à funcionalidade de um conector; - Invoca uma de suas operações correspondente à porta de entrada requisitada; - Requisita, junto ao próximo conector ou componente configurado, a operação correspondente a uma de suas portas de saída. Essa requisição é feita através da operação *forward()* da classe Handler.	`Configuracao`
`ServidorHandler`	- Encaminha ao servidor a requisição vinda originariamente do cliente;	`Servidor`
`Configurador`	- Define a arquitetura da aplicação a partir de uma ADL; - Recebe instruções a partir de uma determinada ADL e invoca serviços da classe Configuração para executá-las;	`Configuracao`
`Configuracao`	- Fornece a configuração estabelecida entre componentes e conectores; - Disponibiliza serviços para configurar componentes e conectores e iniciar a aplicação; - Mantém a descrição da arquitetura, bem como informações quanto à execução da mesma;	`Porta`
`Porta`	- Fornece as portas configuradas de conectores e componentes com suas respectivas assinaturas;	-

Figura 4 Tabela que resume as Classes, Responsabilidades e Colaboradores.

A figura 5 apresenta um possível diagrama de instâncias em tempo de execução para a implementação de *Architecture Configurator*.

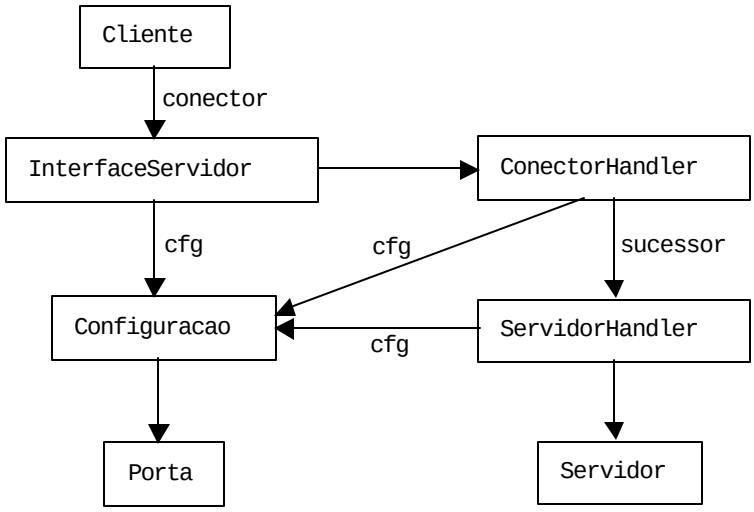


Figura 5 Um possível diagrama de objetos de *Architecture Configurator*.

2.9. Colaborações

Uma vez descritos tipos para portas, componentes e conectores através de uma ADL, torna-se necessário estabelecer, ainda no nível de descrição arquitetural, a estrutura do sistema de software. É a partir da descrição das interligações entre componentes e conectores que inicia-se a colaboração entre os participantes de *Architecture Configurator*.

A colaboração é composta basicamente de duas fases: (i) estabelecimento da configuração arquitetural; (ii) implementação da configuração arquitetural.

A primeira fase é iniciada pelo `Configurador` e descreve as interligações entre componentes e conectores. `Configurador` interpreta a descrição arquitetural, identificando as instâncias de componentes e conectores, e procedendo com sua instanciação. A referência às instâncias fica armazenada na classe `Configuracao`. Após isso, `Configurador` identifica as ligações previstas entre componentes e/ou conectores, e a cada instrução referente a uma ligação, é invocada na classe `Configuracao` a operação para realizar tal ligação. Ainda nesta fase, as instâncias de componentes e conectores são iniciados, através de instrução apropriada.

A segunda fase trata do mecanismo de interceptação e encaminhamento de requisições oriundas de componentes clientes. Este mecanismo compreende basicamente a organização dos componentes e conectores, realizada pela classe `Handler`, conforme a configuração estabelecida na primeira fase.

A primeira fase está retratada na figura 6. O diagrama de seqüência ilustra a interligação de `Cliente` a `ServidorHandler` através de `ConectorHandler` e a iniciação dos mesmos. Instâncias de componentes e conectores são mantidas pela classe `Configuracao`.

Note-se que `Servidor` não aparece no diagrama de seqüência da primeira fase, uma vez que é iniciado pelo objeto correspondente à classe `ServidorHandler`. Para cada classe que oferece serviços (`Servidor`) há uma classe respectiva associada (`ServidorHandler`), facilitando o processo de encadeamento de objetos realizado pela classe `Handler`. Esta solução é baseada no *pattern Object Recursion* apresentado em [HFR99].

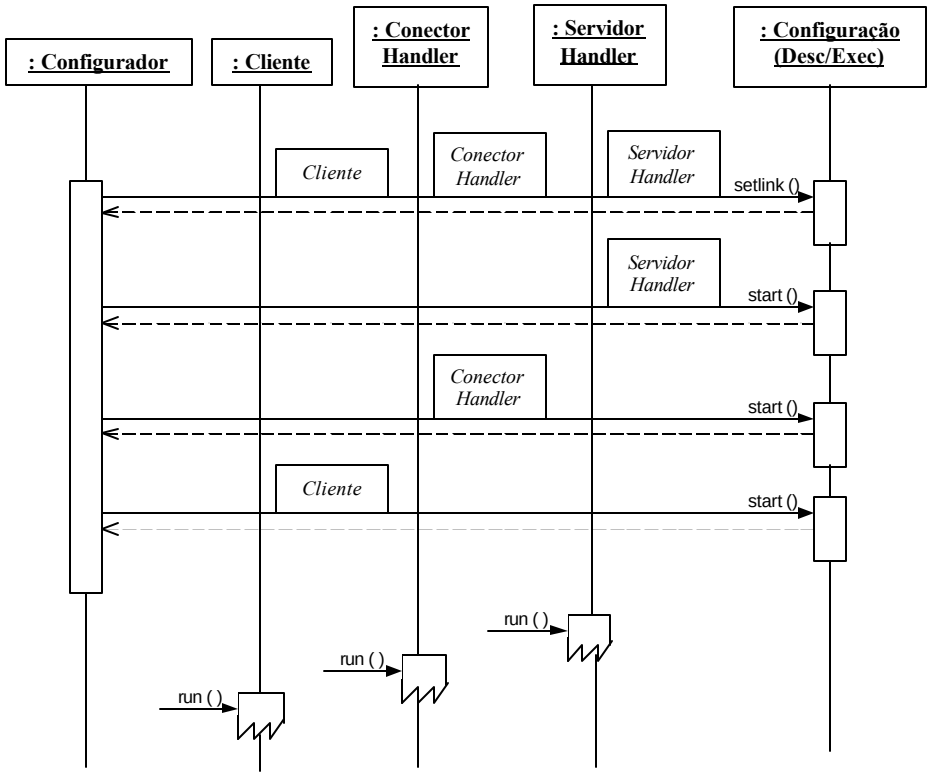


Figura 6 Diagrama de Seqüência (Colaboração). Primeira fase.

A iniciação de `Cliente` dará princípio à segunda fase da colaboração, retratada na figura 7. A seqüência de colaboração inicia-se a partir da invocação, por parte da instância de `Cliente`, a um serviço (*`servico1()`*, na figura) oferecido por um `Servidor`. Todas as invocações a partir do `Cliente` são interceptadas por um objeto `InterfaceServidor`. Este encaminha a invocação através da cadeia de conectores e componentes, representados por instâncias de `ConectorHandler` e `ServidorHandler`. Cada uma das instâncias de `ConectorHandler` representa, preferencialmente, um aspecto não-funcional, implementado por um conector. Ou seja, a cada conector configurado, deve estar associada uma instância de `ConectorHandler`.

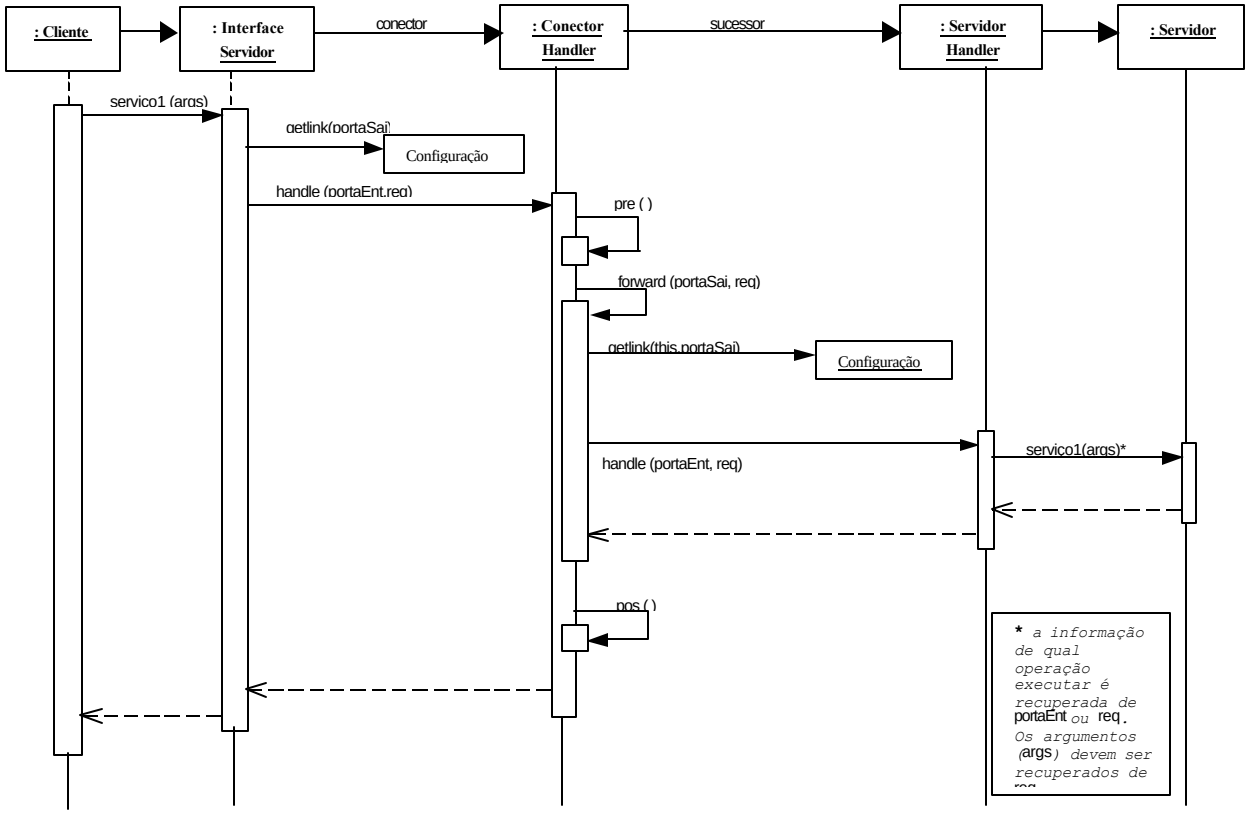


Figura 7 Diagrama de Seqüência (Colaboração). Segunda fase.

A invocação de um serviço corresponde a uma das portas de saída configuradas para o componente `Cliente`. A assinatura da porta é passada a `InterfaceServidor`, que, de posse disso, obtém em `Configuracao` a referência ao primeiro conector ligado a `Cliente`, e a porta de entrada ligada à sua porta de saída em questão. Feito isso, o método *`handle()`* de conector é invocado, com argumentos referentes à porta de entrada e à requisição realizada por `Cliente`.

O método *`handle()`* começa sua operação invocando o método *`pre()`* do respectivo conector, responsável pelos aspectos não-funcionais que devem ser executados antes do encaminhamento da requisição para o próximo conector/componente configurado. Após a execução de *`pre()`*, o fluxo deve seguir por alguma porta de saída do conector. Definida a porta de saída, a operação *`forward()`* do conector é invocada. Tal operação utiliza a classe `Configuracao` para obter a referência ao próximo componente/conector configurado, de modo análogo ao explicado anteriormente, e a requisição segue no encadeamento.

O final do encadeamento ocorre quando `ServidorHandler` é encontrado e tem sua operação *`handle()`* requisitada. Esta operação tem funcionamento análogo à de `ConectorHandler`, porém não invoca *`forward()`*, e sim concretiza a requisição junto a `Servidor`, invocando o método que implementa o serviço desejado.

No contexto das classes `InterfaceServidor` e `Handler`, portas de saída são representadas por invocações de operações, e portas de entrada são representadas por operações declaradas na interface de componentes e /ou conectores (assinaturas).

É importante ressaltar que podem ser criadas várias classes que implementam aspectos não-funcionais, e também várias instâncias de cada uma delas. Em outras palavras, seria possível ter classes `ConectorHandler1`, `ConectorHandler2`, `ConectorHandler3`, etc., cada qual com uma ou mais instâncias. A mesma observação vale para as classes `Cliente`, `Servidor` e `ServidorHandler`, tornando fácil também a separação entre requisitos funcionais do sistema. Deve ser levado em conta o fato de que cada classe `Servidor` criada deve ter associada a si uma classe `InterfaceServidor` específica.

2.10. Conseqüências

A utilização de *Architecture Configurator* traz as seguintes consequências:

a) ***conectores independentes***: cada conector pode ser construído levando-se em conta um aspecto não-funcional diferente e podem ser inter-relacionados sem que conheçam a configuração da aplicação.
b) ***transparência***: componentes requisitam e fornecem serviços transparentemente em relação aos conectores configurados.
c) ***responsabilidade flexível***: conectores têm amplo poder de manipular e analisar as informações por eles recebidas e podem encaminhá-las ou não a outro conector ou ao `Servidor`. A operação `forward()` busca o sucessor na seqüência de conectores e pode ser invocada ou não pelos mesmos, conforme os requisitos da aplicação.
d) ***conectores adaptáveis***: os conectores podem ser concebidos independentemente dos componentes os quais intermediarão. Uma vez implementados e configurados em uma determinada aplicação, não necessitam de alterações em decorrência de alguma modificação da interface de um dos componentes intermediados.
e) ***conectores podem ser usados para estender aplicações já existentes***: este *design pattern*, ao tornar possível a organização dos conectores, facilita a implementação de novos requerimentos que surgem na aplicação. Adaptações que seriam necessárias aos componentes para suprir tais requerimentos podem ser realizadas em novos conectores, os quais podem ser construídos e configurados independentemente de outros existentes.
f) ***conectores genéricos***: um determinado conector pode ser desenvolvido independentemente da quantidade de portas dos componentes que irá intermediar, e da diversidade de assinaturas das mesmas.

2.11. Patterns Relacionados

`InterfaceServidor` possui a mesma interface de `Servidor`, com o objetivo de controlar o acesso ao mesmo. O controle de acesso ao `Servidor` foi baseado no *design pattern Proxy* [GHJ95], sendo que `Servidor` corresponde a *RealSubject* de *Proxy* e `InterfaceServidor` corresponde à classe *Proxy* do mesmo *design pattern*. O `Servidor` desconhece a existência de `InterfaceServidor`.

A interligação dos conectores e componentes como desenhada na estrutura de *Architecture Configurator* segue o *design pattern Object Recursion* [HFR99], no qual sua classe abstrata `Handler` corresponde a `Handler` de *Architecture Configurator*, suas classes *Terminator* e *Recurser* correspondem respectivamente a `ServidorHandler` e

`ConectorHandler` e, finalmente, sua classe *Initiator* corresponde a `InterfaceServidor`. *Object Recursion* é utilizado no *Chain of Responsibility* de [GHJ95].

O *design pattern Component Configurator*, disponível em [SSR00], permite que uma aplicação configure seus componentes (*link* e *unlink*) em tempo de execução, sem necessidade de quaisquer modificações e/ou recompilações dos mesmos. O processo é feito separando-se a implementação dos componentes de suas operações de controle, tais como *init()*, *fini()*, *suspend()*, *resume()*. O *pattern* possui duas classes importantes: *Component Repository* e *Component Configurator*. Ambas classes têm funcionalidades semelhantes à `Configuracao` e `Configurador` de *Architecture Configurator*.

A diferença básica entre *Component Configurator* e *Architecture Configurator* relaciona-se aos aspectos arquiteturais. O primeiro contempla o espaço de endereços da aplicação, armazenando as referências dos componentes na classe *Component Repository*, sem ater-se à configuração arquitetural da mesma. *Architecture Configurator*, por sua vez, mantém a descrição da arquitetura e informações de execução da mesma. Assim, *Architecture Configurator* organiza os componentes e conectores de acordo com a descrição arquitetural mantida pela classe `Configuracao`.

Entretanto, *Architecture Configurator* pode ser empregado em conjunto com *Component Configurator*, com o objetivo de permitir a reconfiguração de componentes e conectores da aplicação em tempo de execução.

Interceptor [SSR00] é um *architectural pattern* que permite a adição de serviços a determinado *framework*, de forma transparente, tornando-o extensível. Este *pattern* relaciona-se ao *design pattern Architecture Configurator* no sentido de ambos permitirem a extensão da aplicação com serviços não previstos nos seus componentes básicos. Entretanto, *Interceptor* trata da extensão de *frameworks*, que são estruturas inacabadas que servem de base para o desenvolvimento de aplicações, enquanto que *Architecture Configurator* permite a extensão de arquiteturas mais genéricas, através da incorporação de conectores que podem encapsular serviços não-funcionais à aplicação.

Outro *design pattern* relacionado é *Facade* [GHJ95], o qual pode auxiliar na composição de `Configuracao` e `Porta`. No modelo da figura 4, `Configuracao` serve de *facade* para a funcionalidade de `Porta`, com o objetivo de distinguir as funções de `Porta` e `Configuracao`. Entretanto, uma terceira classe pode ser definida para servir como *facade* para `Configuracao` e `Porta` com o objetivo de tornar unificada a interface de acesso a ambas classes, simplificando a implementação de conectores e componentes.

Architecture Configurator pode ser empregado na implementação do *architectural pattern Reflection* [HFR99]. Este *pattern* separa a arquitetura de um sistema em dois níveis: nível base (*base level*) e meta-nível (*meta level*). O primeiro define a lógica da aplicação e o segundo mantém informações a respeito da própria estrutura e comportamento do sistema. O meta-nível consiste de metaobjetos (*metaobjects*) com uma interface que permite ao nível base acesso ao meta-nível. As classes `Cliente` e `Servidor` de *Architecture Configurator* podem compor o nível base, enquanto os conectores e suas respectivas portas podem compor o meta-nível, fazendo o papel de metaobjetos.

2.12. Usos Conhecidos

De um modo geral, os sistemas de software podem ser configurados como uma aplicação cliente-servidor, na qual existem módulos que oferecem serviços e módulos que requisitam serviços. Sendo assim, *Architecture Configurator* pode ser utilizado em qualquer

aplicação que tenha essa característica, bastando para isso a implementação do esquema de interpretação da descrição arquitetural (classe `Configurador`), estruturas de dados para armazenamento e recuperação da informação de configuração (classes `Configuracao` e `Porta`), e o esquema de interceptação e encaminhamento das requisições (classes `InterfaceServidor`, `Handler` e os *handlers* para conectores e servidor).

Um uso específico para o *Architecture Configurator* é o ambiente de suporte R-RIO [Lob99]. No R-RIO os componentes são objetos instanciados a partir de classes escritas em Java e conectores podem encapsular protocolos de comunicação e aspectos relacionados à interação [SLL99] entre os componentes. R-RIO permite a instanciação de componentes em um ambiente distribuído e mantém a configuração da aplicação implementada em gerentes localizados nos *hosts* do sistema distribuído. A configuração é alimentada por um interpretador central, ao qual recebe instruções baseadas na ADL Babel.

Os gerentes do R-RIO têm funcionalidade equivalente à classe `Configuracao` do *Architecture Configurator*, mantendo informações da descrição da configuração arquitetural, bem como da execução do sistema. O interpretador, por sua vez, tem associação com a classe `Configurador`.

O conceito de porta de entrada e saída com o respectivo mapeamento para operações, é usado pelo R-RIO, tanto para componentes quanto para conectores. Outro aspecto do R-RIO refere-se ao uso do mecanismo de reflexão estrutural da linguagem Java para gerar de forma automática um *proxy* para o componente `Servidor`, nos mesmos moldes da classe `InterfaceServidor` de *Architecture Configurator*.

Alguns conceitos colocados em *Architecture Configurator* aparecem em diversas ADLs e linguagens para definição de conectores. O conceito de conector aparece em ADLs como ACME [GMW97], Babel [Mal96], C2 [Med99], UniCon [SDZ96] e *Wright* [AG97]. Em todas elas existe a distinção entre o tipo de conector e a instância de conector.

Propriedades em relação às portas de componentes e conectores, representadas pela classe `Porta`, aparecem em UniCon na forma de listas chamadas Listas de Propriedades. Portas de componentes são denominadas *players* e portas de conectores são denominadas *roles* em UniCon.

O conceito de encaminhamento/roteamento de mensagens atribuído ao conector aparece na linguagem C2. O conector C2 tem a responsabilidade primária de realizar o roteamento e o *broadcast* de mensagens e, secundariamente a definição e implementação de políticas de filtragem de mensagens. Tais políticas têm paralelo com a interceptação e manipulação descritas em *Architecture Configurator*.

3. Conclusão

Este artigo apresentou o *design pattern Architecture Configurator*, proposto para servir de base à implementação da configuração de aplicações.

Arquiteturas definidas em alguma ADL podem ser implementadas através deste *design pattern*, uma vez que ele emprega características próprias dos conectores e características inerentes ao processo de configuração de componentes e conectores. *Architecture Configurator* faz a leitura da arquitetura e a implementa utilizando propriedades de conectores como a interceptação, manipulação e encaminhamento de mensagens ou requisições. As interconexões entre componentes e conectores são interpretadas conforme seus elos de ligação definidos pelas portas.

Cada componente ou conector participante da arquitetura pode ser implementado de forma autônoma e integrado de acordo com a configuração arquitetural.

A solução descrita em *Architecture Configurator* não apresenta novidades conceituais quanto às propriedades de configuração, de conectores ou mesmo de arquiteturas de software, mas reforça estes pontos, propondo um modelo-base sobre o qual a configuração entre componentes e conectores pode ser realizada.

4. Agradecimento

Os autores fazem um agradecimento à equipe de *shepherding* do SugarloafPLop 2002, em especial às professoras Rossana Maria de Castro Andrade e Rosana Teresinha Vaccare Braga, pela ajuda fundamental de orientação para que fosse possível atingir esta versão final deste artigo, e ao grupo de trabalho do Writers Workshop do SugarloafPlop 2002, pelas valiosas contribuições que ajudaram a enriquecer o conteúdo do trabalho apresentado.

5. Referências

[AG97] R. Allen, D. Garlan. *A Formal Basis for Architectural Connection*. ACM Transactions on Software Engineering and Methodology, vol. 6, no. 3, pág. 213-249, julho 1997.

[App00] B. Appleton. *Patterns and Software: Essential Concepts and Terminology*. Disponivel em http://www.enteract.com/~bradapp/docs/patterns-intro.html, 2000.

[Car01] Carvalho, S. *Um Design Pattern Para a Configuração de Arquiteturas de Software*. Dissertação de Mestrado. IC/UFF, Niterói-RJ, maio de 2001.

[GHJ95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GMW97] D. Garlan, R. Monroe, D. Wile. *Acme: An Architecture Description Interchange Language*. CASCON'97. Novembro, 1997.

[HFR99] N. Harrison, B. Foote, H. Rohnert. *Pattern Languages of Program Design 4*. Software Pattern Series. Addison-Wesley, 1999.

[Lob99] M. Lobosco. *Um Ambiente para Suporte à Construção e Evolução de Sistemas Distribuídos*. Dissertação de Mestrado. IC/UFF. Março, 1999.

[Mal96] V. V. Malucelli. *Babel – Construindo Aplicações por Evolução*. Dissertação de Mestrado. DEE / PUC-RJ. Fevereiro 1996.

[Med99] N. Medvidovic. *Architecture-Based Specification–Time Software Evolution*. Tese de Doutorado (PhD). Universidade da Califórnia, Irvine, 1999.

[MRB97] R. C. Martin, D. Riehle, F. Buschmann. *Pattern Languages of Programming Design 3*. Software Pattern Series, Addison-Wesley, 1997.

[SDZ96] M. Shaw, R. Deline, G. Zelesnik. *Abstractions and Implementations for Architectural Connections*. Third International Conference on Configurable Distributed Systems. Maio 1996.

[SLL99] A. Sztajnberg, M. Lobosco, O. Loques. *Configurando Protocolos de Interação na Abordagem R-RIO*. Simpósio Brasileiro de Engenharia de Software. Florianópolis, Santa Catarina. 1999.

[Som00] I. Sommerville. *Software Engineering*. 6th. Edition. Addison-Wesley, 2000.

[SSR00] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. Volume 2. John Wiley & Sons, 2000.

[Sun00] Sun Microsystems. *Java 2 Plataform, Standard Edition Documentation*. http://java.sun.com/products/jdk/1.3/docs/index.html. Maio 2000.

Padrões de Projeto para Estruturação de Aplicações Distribuídas Enterprise JavaBeans

Klissiomara Dias* and Paulo Borba†
Centro de Informática
Universidade Federal de Pernambuco

Resumo

Enterprise JavaBeans (EJB) auxilia o desenvolvimento de aplicações de negócio que lidam com aspectos como distribuição, persistência e transações. Aplicações dessa natureza, se desenvolvidas de forma ad hoc podem levar a sistemas cujo código mistura aspectos de negócio com aspectos não funcionais, podendo comprometer alguns fatores de qualidade, tais como reusabilidade e extensibilidade. Por este motivo, este artigo propõe dois padrões de projeto que auxiliam na estruturação de aplicações EJB, visando obter alguns benefıcios como reuso, modularidade, extensibilidade, independência de tecnologia (distribuição ou dados) e desempenho. Estes padrões auxiliam ainda na estruturação de aplicações EJB a partir de sistemas já existentes, sem EJB, minimizando o impacto das mudanças sobre as demais camadas da aplicação.

Abstract

Enterprise JavaBeans (EJB) technology provides support for the development of modern applications, taking into consideration aspects like distribution, persistence and transactions. Applications of this nature, if developed in ad hoc fashion can result in systems whose code mixes business and non-functional requirements (for example, distribution and persistence), being able to compromise some quality aspects, such as reusability and extensibility. This paper proposes two design patterns that aid in structuring EJB applications, in order to gain some benefits like reuse, modularity, extensibility, technology independence (i.e. distribution or data) and performance. In addition, they can assist in designing EJB applications from non-EJB already-existing systems, and thus softening the impact of changes on other application layers.

*Financiada pelo CNPQ. Email: kld2@cin.ufpe.br

†Parcialmente financiado pelo CNPq, vínculo 521994/96–9. Email: phmb@cin.ufpe.br

1 Introdução

O padrão arquitetural *Layer* (padrão em camadas) [5] é utilizado para a estruturação de aplicações complexas que lidam com diferentes requisitos, funcionais e não funcionais. Com a utilização desta arquitetura, é possível distribuir as classes que compõem o sistema em camadas bem definidas, de acordo com cada aspecto da aplicação (negócio, persistência, comunicação, etc.)

A divisão de um sistema em camadas permite obter aplicações modulares e reutilizáveis, uma vez que o código de diferentes aspectos (apresentação, comunicação, negócio e dados, por exemplo) não são misturados. Além disso, os elementos das diferentes camadas comunicam-se através de interfaces.

Em aplicações distribuídas, a comunicação entre objetos executando em diferentes máquinas é realizada através de mecanismos e protocolos de comunicação. Quando tais objetos manipulam aspectos de comunicação diretamente, a tendência é que suas funcionalidades sejam misturadas com as tarefas de comunicação. O mesmo acontece com aplicações que utilizam algum meio de armazenamento persistente. O desenvolvimento *ad hoc* de aplicações que utilizam alguma plataforma de persistência, para armazenamento e recuperação de seus objetos, leva a sistemas que misturam código de acesso a dados com o código de negócio da aplicação.

Enterprise JavaBeans (EJB) [14] é uma tecnologia que trata de aspectos como distribuição e persistência. Por este motivo, foi feita uma análise acerca da necessidade do uso de padrões existentes para a arquitetura EJB. Como resultado desta análise, surgiram os padrões apresentados neste artigo.

Desta forma, os padrões apresentados neste artigo são utilizados no contexto de aplicações EJB e estruturam classes e objetos que preenchem as camadas citadas acima. A apresentação dos padrões em um mesmo artigo visa facilitar sua compreensão, uma vez que estes estão relacionados:

- *Distributed Adapters Pattern with EJB* (DAP-EJB). O DAP-EJB corresponde à adaptação do padrão *Distributed Adapters Pattern* (DAP) [2], o qual foi primeiramente definido em *Progressive Development of Distributed Object-Oriented Applications* [1] e visa isolar o *middleware* da aplicação, tornando-a extensível para vários tipos de mecanismo de comunicação.

- *Persistent Data Collections with EJB* (PDC-EJB). O PDC-EJB corresponde à adaptação do padrão *Persistent Data Collections* (PDC) [11], o qual visa permitir reutilização da lógica de negócio para diferentes mecanismos de persistência.

2 DAP-EJB: Um Padrão para Distribuição com EJB

Objetivo

Fornecer uma estrutura para implementação de distribuição em um sistema com EJB, visando separação de conceitos e conseqüentemente fatores de qualidade como modularidade, extensibilidade e reusabilidade.

Contexto

O padrão DAP-EJB é utilizado no contexto de comunicação remota entre dois componentes, onde é desejável que tais componentes não estejam acoplados à tecnologia de distribuição.

Problema

Apesar de EJB prover interoperabilidade e transparência de localização para o acesso aos objetos remotos, os clientes de um *enterprise bean* ainda precisam fazer uso de interfaces e classes específicas da API de distribuição a fim de obter as referências remotas para os *beans*. Isto implica que a interface com o usuário acaba tendo código específico de EJB. O mesmo acontece com relação à camada de negócio. Como resultado, tanto a interface com o usuário quanto a camada de negócio ficam vulneráveis às modificações realizadas na camada de comunicação.

Forças

O DAP-EJB leva em consideração as seguintes forças:

- Um componente deve ser capaz de acessar serviços remotos fornecidos por outros componentes;
- Os componentes devem ser independentes do *middleware* da aplicação; isto permite, por exemplo, que o mesmo sistema possa utilizar diferentes *middleware* ao mesmo tempo ou, ainda, possa ser executado localmente.
- A modificação no código dos componentes para suportar comunicação deve ser minimizada; ou seja, a inserção do componente de distribuição em uma aplicação não distribuida deve causar pouco impacto no código já existente.
- Modificação da tecnologia de distribuição deve ser uma tarefa simples; é importante estruturar os aspectos de distribuição de forma modular, vislumbrando o fraco acoplamento destes em relação cliente e o negócio da aplicação.

Solução

Para resolver o problema apresentado, o DAP-EJB introduz o uso de um par de adaptadores [7] que são utilizados para encapsular o código relacionado à API de distribuição. O objetivo é permitir que a inserção, remoção, ou troca do *middleware* de distribuição de uma aplicação seja realizado de forma a minimizar as mudanças necessárias no código do sistema. O uso destes adaptadores isola a interface com o usuário e a camada de negócio da plataforma de distribuição do sistema, abstraindo desta forma a tecnologia utilizada para comunicação remota entre componentes.

Estrutura

O diagrama de classes da Figura 1 destaca a estrutura do padrão DAP-EJB. As classes em cinza denotam os adaptadores e seus colaboradores, os quais, basicamente, escondem

a API de distribuição da interface com o usuário e o código de negócio. As demais classes lidam com os aspectos de negócio da aplicação. Os elementos que fazem parte do padrão e seus colaboradores são explicados a seguir.

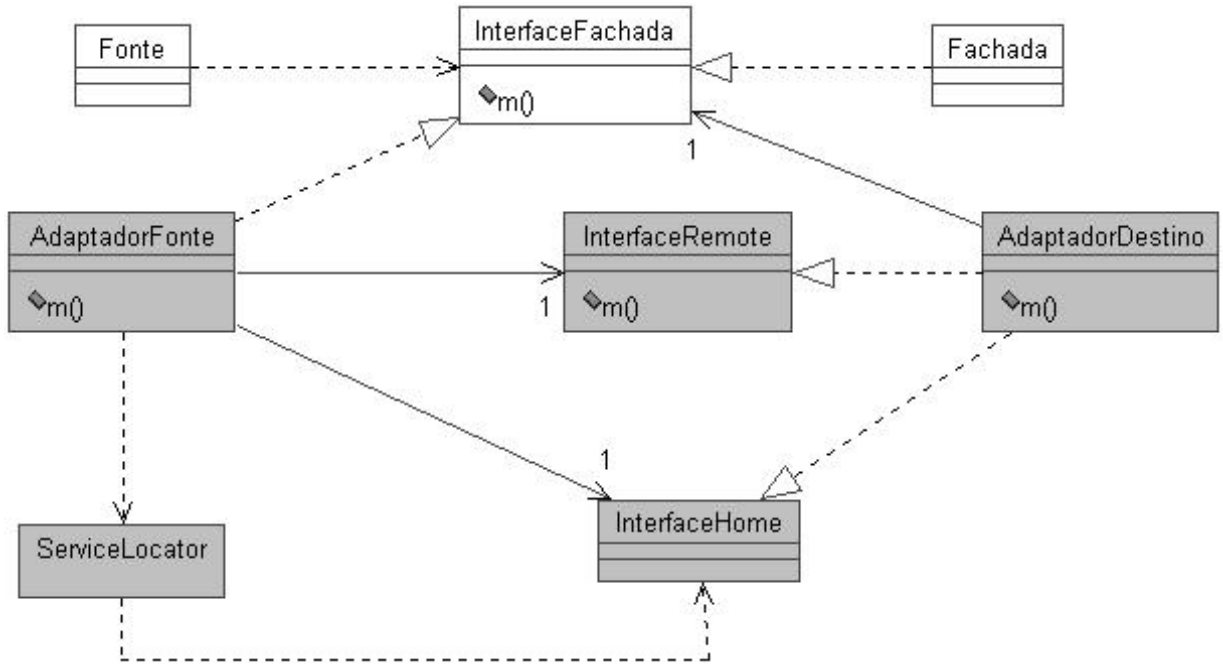


Figura 1: Estrutura do padrão DAP-EJB

- Fonte

  A classe `Fonte` representa qualquer objeto que faz o papel de cliente (GUI, por exemplo) da fachada e que está localizado em uma máquina remota em relação aos objetos do sistema.

- Fachada

  A classe `Fachada` é estruturada de acordo com o padrão *facade* [7], sendo responsável por encapsular todos os serviços oferecidos pelo sistema (no diagrama, o método `m` representa um dos possíveis serviços). Esta classe representa, na verdade, o objeto remoto a ser acessado pelo cliente.

- InterfaceFachada

  A `InterfaceFachada` é uma interface que abstrai o comportamento da fachada em um cenário distribuído. Esta classe, em conjunto com as classes `Fonte` e `Fachada` constituem uma camada independente da tecnologia de distribuição. Os demais elementos do diagrama ficam responsáveis por esse aspecto e constituem a camada de distribuição do sistema.

- AdaptadorFonte

O adaptador fonte é uma classe Java [8] "pura"e isola a classe `Fonte` do código de distribuição. Um objeto desta classe reside na mesma máquina que o objeto da classe fonte e trabalha como um *proxy* [7] para o adaptador destino. Repassa as chamadas feitas pelos clientes para o próprio adaptador destino, isolando todo o código relacionado com a plataforma de distribuição, mantendo o cliente isolado deste código (inclusive exceções de comunicação).

- AdaptadorDestino

  O adaptador destino é um *stateless session bean* [14] e, como todo componente EJB, possui duas interfaces remotas. Este componente é responsável por repassar as chamadas para o objeto fachada.

- ServiceLocator

  A classe `ServiceLocator` é uma classe auxiliar que visa abstrair a complexidade do processo de localização e criação dos *beans*, bem como melhorar o desempenho do sistema. É utilizada no padrão DAP-EJB para auxiliar o processo de criação das referências remotas ao adaptador destino.

Dinâmica

A Figura 2 apresenta o diagrama de seqüência para um cenário do DAP-EJB. Durante a inicialização, o `Fonte` cria um `AdaptadorFonte`, o qual executa o método `getHome` de `ServiceLocator` e este executa uma operação de `lookup` sobre o serviço de nomes JNDI, a fim de obter uma referência da interface *home* do `AdaptadorDestino`. Após obter a referência do *home* do adaptador destino, o `AdaptadorFonte` executa o método `create` sobre este a fim de obter uma referência à sua interface remota, no intuito de obter acesso aos serviços oferecidos pelo `AdaptadorDestino`. Ao executar uma operação `create` sobre o adaptador destino, uma única instância da `Fachada` é obtida através do método `getInstance`[1]. O `Fonte` então, invoca uma operação local `m` sobre o `AdaptadorFonte` e este invoca a operação remota `m` do adaptador destino, o qual delega esta chamada localmente para a `Fachada`.

Consequências

O DAP-EJB oferece as seguintes vantagens:

- Código modular

  O uso do padrão permite a estruturação dos aspectos de distribuição de forma modular, de modo a propiciar o fraco acoplamento destes em relação ao cliente e a camada de negócio da aplicação.

- Reutilização e extensibilidade

  Devido à estrutura modular obtida com o padrão, é possível utilizar as classes fonte e fachada mais facilmente em outras aplicações que utilizem diferentes tecnologias de distribuição. Além disso, mudanças na camada de comunicação são mais fáceis

[1] A `Fachada` é implementada como um *Singleton* [7]

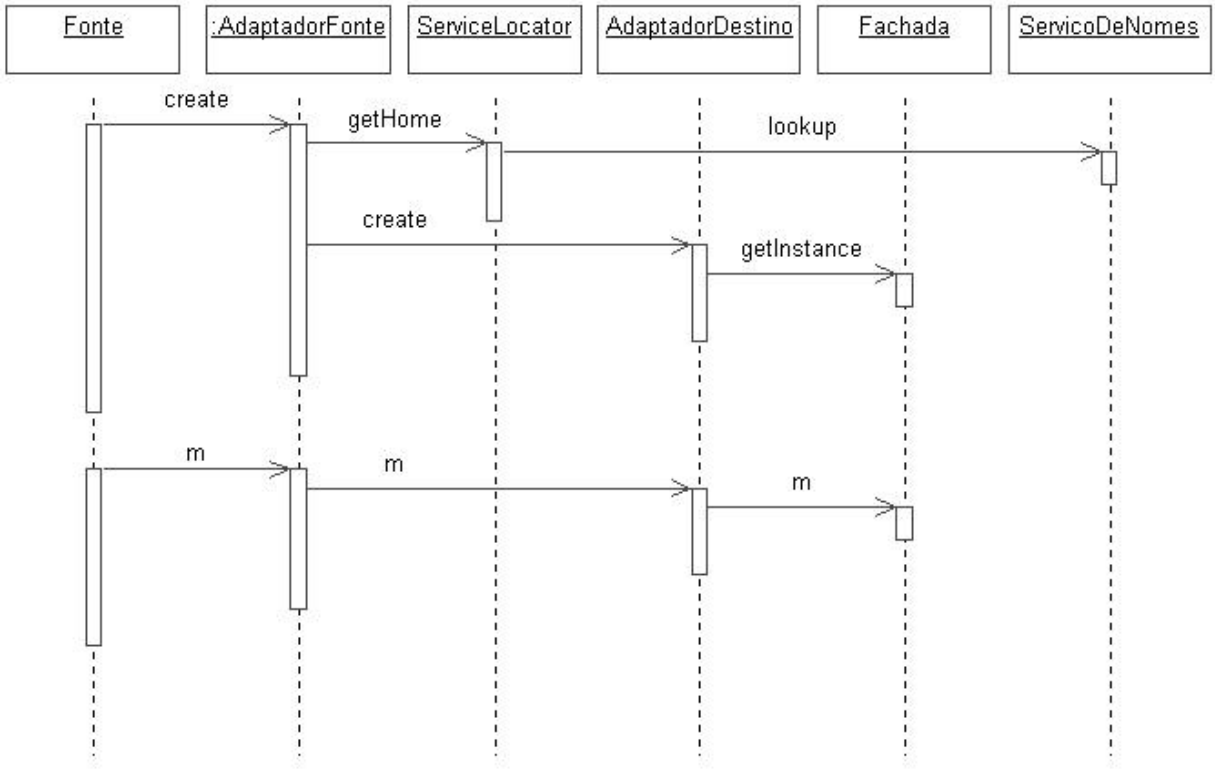


Figura 2: Diagrama de Sequência do DAP-EJB.

de realizar porque afetam somente os adaptadores fonte e destino. Um exemplo das situações citadas é um mesmo sistema ser acessado remotamente por um cliente utilizando EJB como API de distribuição ou ser acessado localmente, na máquina do cliente sem EJB. O mesmo sistema pode, ainda, ser acessado por um cliente que utiliza CORBA, por exemplo, como tecnologia de comunicação remota.

- Implementação progressiva

  O padrão suporta implementação progressiva [4]. Desenvolvedores constroem um protótipo funcionalmente completo, onde o cliente depende diretamente da fachada, e realizam os testes da funcionalidade do sistema a fim de validar seus requisitos funcionais. Depois, o componente de distribuição pode ser inserido causando pouco impacto no código já existente. Isto é possível porque o componente de distribuição implementa a mesma interface que a fachada. É importante destacar que o padrão além de permitir inserir o componente de distribuição em um sistema já existente, local, também permite adaptar um sistema distribuído em outra plataforma (por exemplo, RMI), sem que para isso seja necessário descartar as classes correspondentes ao cliente e fachada.

Por outro lado, o DAP-EJB possui as seguintes desvantagens:

- Aumento do número de classes

  Um par de adaptadores, duas interfaces remotas e um `ServiceLocator` são necessários. Todavia, essa estrutura é simples e seu código pode ser gerado de forma automática

com o auxílio de ferramentas, que são importantes para a utilização do padrão na prática. Estas ferramentas devem não somente auxiliar na geração dos adaptadores e elementos relacionados ao componente de distribuição, mas também na manutenção, quando da inserção de um método na fachada, por exemplo.

- Eficiência

  Com a introdução dos adaptadores, é necessário um maior número de invocações até que a chamada do método original seja executada no objeto remoto. Além do *overhead* causado com a introdução dos adaptadores, existe um outro fator que contribui para o *overhead* quando da invocação de métodos remotos que, neste caso, é uma característica intrínseca de EJB e que, portanto, não é uma desvantagem do padrão em si. Para se ter acesso a um objeto remoto EJB é necessário realizar duas chamadas de métodos sobre suas interfaces antes de invocar o método de negócio. Além disso, eficiência é um problema genérico da arquitetura em camadas, uma vez que um maior número de classes é introduzido no sistema, aumentando, com isso, a transferência de dados entre camadas e, por conseguinte, o número de chamadas de métodos [5].

Exemplo

O diagrama de classes UML [3] da Figura 3 ilustra a estrutura do padrão DAP-EJB através do exemplo de uma simples aplicação bancária. As classes em cinza correspondem aos adaptadores e elementos utilizados para esconder a API de distribuição do código de negócio e do cliente. As demais classes denotam os aspectos de negócio da aplicação.

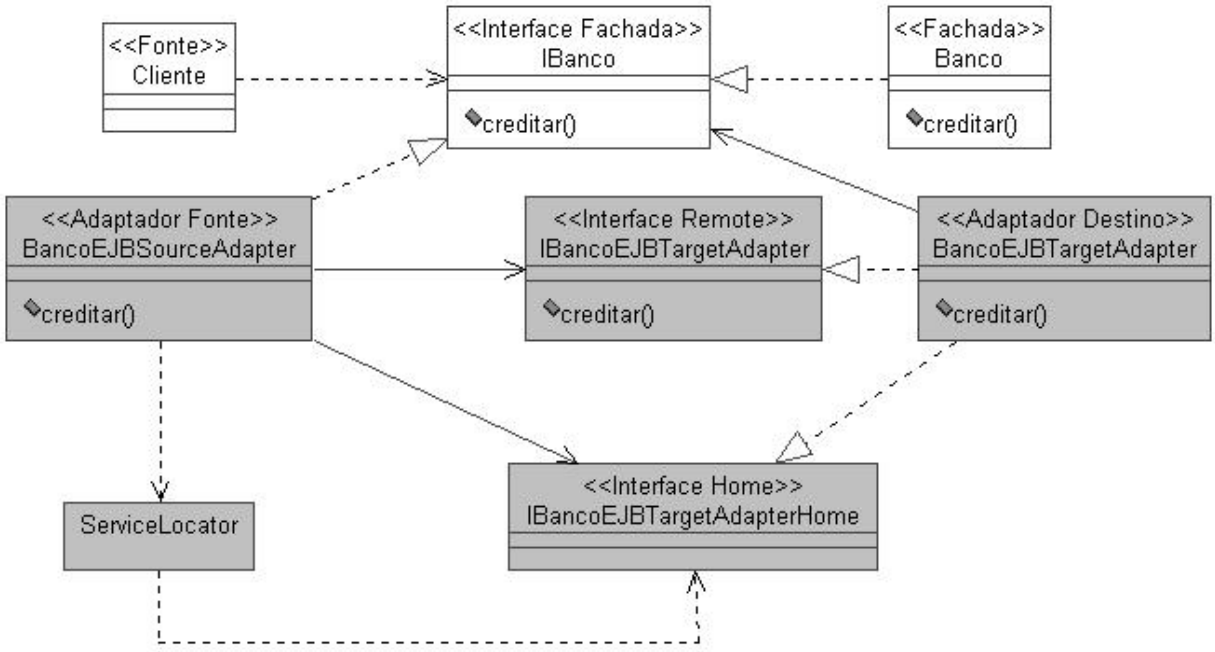


Figura 3: Estrutura de uma aplicação bancária de acordo com o padrão DAP-EJB.

Implementação

Adaptador destino implementado como *Stateless session bean*

O adaptador destino é implementado com um *stateless session bean* pelo fato de componentes dessa natureza representarem objetos cujas instâncias são equivalentes no *container*[2]. Uma mesma instância de um *stateless session bean* pode servir às requisições de diferentes clientes, minimizando os recursos necessários para suportar uma grande quantidade destes. Além disso, ao adaptador destino também podem ser atribuídos aspectos relacionados aos serviços de transações e segurança, por exemplo.

ServiceLocator para acesso às referências remotas dos *beans*

A tarefa para ter acesso a um componente EJB é comum para todos os clientes (externos ou internos) que precisam acessar seus serviços. Isto implica que muitos tipos de clientes repetidamente utilizam os serviços JNDI [10], uma API que fornece um conjunto de interfaces e classes para acessar uma vasta quantidade de recursos, entre os quais permite a localização de objetos remotos, o que resulta em código duplicado nos mesmos. Além disso, o processo necessário para localizar e obter referências remotas aos *homes* dos *beans* gasta recursos significativos do servidor de aplicação, o que pode causar impacto no desempenho do sistema. Neste contexto, a classe `ServiceLocator` [6], um padrão que visa abstrair a complexidade do processo de localização e criação dos *beans*, bem como melhorar o desempenho do sistema, é utilizada no padrão DAP-EJB para auxiliar o processo de criação das referências remotas.

Código

Nesta seção é apresentado o código para os elementos do exemplo do padrão. A interface com o usuário cria um `BancoEJBSourceAdapter` e delega as requisições do cliente para este. O adaptador fonte é uma classe Java "pura" e implementa a interface da fachada `IBanco` de modo a permitir que a interface com o usuário não tenha conhecimento da tecnologia de distribuição sendo utilizada.

O adaptador fonte declara como atributo as interfaces *home* e remota do adaptador destino, por questões de eficiência. Estes atributos são representados por `h` e `banco`, respectivamente.

```
public class BancoEJBSourceAdapter implements IBanco {
  private IBancoEJBTargetAdapter banco;
  private IBancoEJBTargetAdapterHome h;
```

Com o auxílio dos métodos da classe `ServiceLocator`, uma única instância da interface `IBancoEJBTargetAdapterHome` é obtida. O método auxiliar `conectar` realiza este processamento.

[2]Nome dado ao ambiente de execução dos compoenentes EJB.

```
    private void conectar() throws CommunicationException {
       Class home = IBancoEJBTargetAdapterHome.class;
       try{
          if (h == null){
              h = (IBancoEJBTargetAdapterHome)
                   ServiceLocator.getInstance().getHome("banco",home);
          }
          this.banco = h.create();
      } catch(ServiceLocatorException e){
                throw new CommunicationException (...);
      } catch(CreateException e){
                throw new CommunicationException (...);
      }
    }
```

A partir do método `getHome` da classe `ServiceLocator`, a referência à interface *home* do adaptador destino é obtida. Além disso, `conectar` também obtém a referência à interface remota do adaptador destino (`IBancoEJBTargetAdapter`), a partir do método `create` de sua interface *home*.

A exceção `ServiceLocatorException` é uma exceção de aplicação lançada pelo método `getHome` se alguma falha acontecer quando da obtenção do `home` do adaptador destino. A exceção `CreateException` é lançada pelo método `create` se a instância do adaptador destino não puder ser criada. As exceções de aplicação específicas de EJB são trocadas no adaptador fonte pela exceção genérica `CommunicationException`. Esta exceção não depende de qualquer tecnologia de distribuição particular e é definida de modo a permitir que o cliente seja isolado de exceções específicas da API de distribuição. No construtor do adaptador fonte o método `conectar` é chamado. Assim,

```
    public BancoEJBSourceAdapter() throws CommunicationException {
          conectar();
    }
```

quando uma instância do adaptador fonte é criada, o processo de localização e criação da instância do adaptador destino é também realizado.

Após obter a referência remota do *bean*, o adaptador fonte está pronto para executar chamadas aos métodos de negócio do adaptador destino. O método `creditar` do adaptador fonte delega as requisições da interface com o usuário para o adaptador destino.

```
  public void creditar(String numeroConta, double saldo)
           throws CommunicationException, ContaNaoExisteException {
     try{
      banco.creditar(numeroConta, saldo);
     } catch(RemoteException e){
         throw new CommunicationException(...);
     }
 }
...
}
```

Pelo fato do adaptador fonte invocar os métodos remotos do adaptador destino, a exceção `RemoteException` pode ser lançada se ocorrer alguma falha originada a partir da invocação remota. Neste caso, `RemoteException` deve ser trocada pela exceção genérica `CommunicationException`. Assim, o cliente não é exposto às exceções específicas da API de distribuição. A interface `IBancoEJBTargetHome` representa a interface *home* do adaptador destino e herda a interface `EJBHome`.

```
public interface IBancoEJBTargetAdapterHome extends EJBHome {
   public IBancoEJBTargetAdapter create()
      throws CreateException, RemoteException;
}
```

A exceção `RemoteException` deve ser declarada em todo método da interface *home* de EJB, e `CreateException` é uma exceção de aplicação específica de EJB lançada quando referências remotas do *bean* não podem ser criadas. O adaptador destino é um *stateless session bean* e por isso apresenta um único método `create`, sem argumentos, em sua interface *home*. Tal método é responsável por criar as suas referências remotas. A interface `IBancoEJBTargetAdapter` representa a interface remota do adaptador destino e declara os métodos invocados pelo adaptador fonte.

```
public interface IBancoEJBTargetAdapter extends EJBObject {
  public void creditar(String numero,double valor)
           throws ContaNaoExisteException, CommunicationException,
                  RemoteException;
 ...
}
```

Esta interface é o tipo da referência ao adaptador destino e seus métodos devem lançar `RemoteException`. A exceção `ContaNaoExisteException` é uma exceção de aplicação. Além do método `creditar`, outros métodos podem ser declarados nesta interface.

A classe `BancoEJBTargetAdapter` é a classe que representa o adaptador destino.

```
public class BancoEJBTargetAdapter implements SessionBean {
  private IBanco banco;
  private SessionContext context;
```

Esta possui uma referência à interface da fachada `IBanco` no intuito de delegar os serviços para esta executar. Um atributo da interface `SessionContext` também deve ser declarado no adaptador destino, uma vez que é um *session bean*. Esta interface é utilizada pelo *container* durante o ciclo de vida do *session bean*. Na realidade, o *container* tem acesso às informações de um *session bean* através desta interface.

O método `ejbCreate` de `BancoEJBTargetAdapter` obtém uma instância da classe fachada através do método `getInstance`.

```
  public void ejbCreate()
           throws CreateException, RepositorioException,
                  CommunicationException{
       banco = Banco.getInstance();
 }
...
```

Após obter a instância da fachada, o adaptador destino delega a invocação de seus métodos para os métodos desta. Por exemplo, quando o método `creditar` da classe `BancoEJBTargetAdapter` é executado,

```
public void creditar(String numero,double valor)
         throws ContaNaoExisteException, CommunicationException{
         banco.creditar(numero,valor);
}
  ...
```

o método `creditar` da fachada é invocado. Os métodos da interface da fachada `IBanco` declaram `CommunicationException`. Esta exceção genérica é declarada em `IBanco` prevendo que a aplicação se tornará distribuída. Por isso, os métodos do adaptador destino e sua interface remota devem também declarar esta exceção.

Por se tratar de um *session bean*, `BancoEJBTargetAdapter` deve implementar a interface *SessionBean*.

```
    public void ejbRemove() { }
    public void ejbActivate() { }

    public void ejbPassivate() { }
    public void setSessionContext(SessionContext sc){
      this.context = sc;
    }
}
```

O *container* utiliza os métodos dessa interface para notificar as instâncias dos *session beans* sobre os eventos do seu ciclo de vida.

Variações

Variações do DAP-EJB são possíveis. Por exemplo, a fachada do sistema poderia fazer o papel do adaptador fonte e o adaptador destino, um *session bean*, seria introduzido entre esta é a coleção de negócio da aplicação.

Esta é uma abordagem simplificada para o uso dos adaptadores, uma vez que um menor número de classes deve ser gerado, além disso, os benefícios como o uso de *session beans* (distribuição e gerenciamento de transações, por exemplo) são mantidos. No entanto, perde-se um pouco em extensibilidade, uma vez que código específico de EJB é inserido na camada de negócio do sistema (entre a fachada e as demais classes que compõem a camada de negócio).

Usos Conhecidos

O DAP-EJB tem sido utilizado em um sistema de informação para o serviço público de saúde. Este sistema foi desenvolvido para ser executado via *Web*. Neste caso, servlets [9] agem como clientes do adaptador fonte. O adaptador fonte interage com o adaptador destino, e este com a fachada, localizada na mesma máquina.

Um outro uso do DAP-EJB é em um sistema que fornece serviços para o gerenciamento de contabilidade, controle de acesso e serviços financeiros. Na realidade, este sistema utiliza a variação do DAP-EJB, citada na seção anterior. Neste sistema, a fachada da aplicação é utilizada como o adaptador fonte, fazendo acesso ao adaptador destino da aplicação.

O DAP-EJB poderia também ser utilizado em outros tipos de sistema, tais como:

- Um sistema para gerenciar clientes de uma empresa de telecomunicação. O sistema é capaz de registrar telefones móveis, gerenciar informações de clientes e a configuração dos serviços de telefonia. Este sistema pode ser utilizado via *Web*.

- Um sistema para provas interativas. Este sistema tem sido utilizado para fornecer diferentes tipos de provas, tais como simulados baseados em exames de seleção para a universidade, ajudando os alunos a avaliar seus conhecimentos antes de realizarem exames reais.

- Um sistema de supermercado complexo. Este sistema será usado em vários supermercados e já está sendo utilizado em outras empresas do mesmo ramo.

Padrões Relacionados

- DAP (Distributed Adapters Pattern). No trabalho *A Design Pattern for Object-Oriented Distributed Applications* [2], foi descrito o padrão DAP, o qual é utilizado no contexto de comunicação remota entre dois componentes, e visa isolar o código relacionado à API de comunicação destes. Este serviu como base para a adaptação do padrão para *Enterprise JavaBeans* (DAP-EJB) proposto aqui.

  O padrão apresentado neste artigo também utiliza adaptadores como o DAP, no entanto, estes não isolam somente os aspectos de distribuição, mas também são responsáveis pelo gerenciamento de transações e segurança, por exemplo. É importante destacar, no entanto, que o gerenciamento de transações e segurança obtidos com o DAP-EJB só é possível devido a uma característica intrínseca da tecnologia EJB, a qual permite o gerenciamento automático desses aspectos por intermédio de seus componentes. Portanto, esta não é uma característica do padrão, mas sim um benefício obtido com a tecnologia EJB.

  Na realidade, o padrão DAP-EJB, permitiu mostrar que o DAP pode ser também implementado com EJB, com pequenas adaptações, por exemplo, com relação aos aspectos de gerenciamento dos serviços citados acima. A estrutura do padrão permanece a mesma, as mudanças dizem respeito muito mais às características do adaptador destino.

- *Bussiness Delegate* [6] e *Session Facade* [6]. Estes padrões são similares aos adaptadores fonte e destino, respectivamente. No entanto, eles não visam estruturar uma aplicação independente de tecnologia, apesar de poderem fazê-lo. A forma como estão estruturados não segue esta filosofia. O *Bussiness Delegate* é utilizado para isolar o cliente das especificidades da tecnologia EJB, como exceções e *lookup*, por exemplo. No entanto, o *Session Facade* não tem o mesmo propósito do adaptador destino, uma vez que não é utilizado para isolar a tecnologia de EJB do resto da

aplicação. Ele serve como um único ponto de acesso para os demais componentes do sistema (entity beans [14] e afins).

- *Wrapper-Facade* [12]. Este padrão encapsula funções de baixo nível (tais como *sockets* e *threads*) da aplicação. O DAP-EJB encapsula a API de distribuição EJB, da aplicação.

- *Adapter*, *Facade*. DAP-EJB é implementado utilizando os padrões de projeto [7] *Adapter* e *Facade*.

- *Singleton* [7]. Um objeto da classe `Fachada` é implementado como um *Singleton*.

- *Abstract Factory*. Em conjunto com o DAP-EJB, classes auxiliares são utilizadas para o propósito de configuração. Tais classes são estruturadas de acordo com o padrão *Abstract Factory* [7]. Assim, dependendo das informações contidas, por exemplo, em um arquivo de configuração, o sistema poder ser executado localmente(resultando em uma referência para a fachada) ou remotamente (resultando em uma referência para o adaptador fonte).

  Desta forma, a utilização de fábricas permite que o código do cliente seja isolado das mudanças ligadas ao código de distribuição, aumentando, assim, a modularidade do sistema.

3 PDC-EJB: Um padrão para Persistência com EJB

Objetivo

Fornecer uma forma de estruturar aplicações complexas implementadas com EJB de modo a separar o código de acesso a dados do código de negócio e de interface com o usuário. Classes específicas são utilizadas para separar estes conceitos, e interfaces garantem a independência entre a camada de negócio e a camada de dados de um sistema.

Contexto

O padrão PDC-EJB está inserido no contexto de aplicações que utilizam algum tipo de armazenamento e acesso a dados de forma persistente.

Problema

O desenvolvimento *ad hoc* de aplicações que utilizam alguma plataforma de persistência, para armazenamento e recuperação de seus objetos, leva a sistemas que misturam código de acesso a dados com o código de negócio da aplicação. Em particular, aplicações construídas desta forma não podem dispor de objetos de negócio reutilizáveis por outras aplicações, que utilizem diferentes tecnologias para a persitência dos dados. Da mesma forma, se a plataforma de persistência for substituída (JDBC [15] por EJB, por exemplo), o impacto das mudanças no código do sistema não é localizado, ou seja, as classes relacionadas ao domínio do negócio da aplicação também devem ser modificadas.

Desta forma, caso seja necessário adaptar um sistema para utilizar outro mecanismo de persistência, tem-se, na verdade, que desenvolver um novo sistema. Ou seja, reutilização e extensibilidade são seriamente comprometidas em sistemas desenvolvidos sem estruturação alguma, pois não há uma distinção clara entre o código de negócio, que contém regras e objetos de negócio, e o código de dados.

Forças

- Problemas relacionados aos requisitos de negócio do sistema devem ser manipulados independente das operações de acesso a dados;
- A modificação no código do sistema para suportar persistência deve ser minimizada;
- O tipo de mecanismo de armazenamento[3] pode ser substituído durante a vida útil de um sistema;
- Classes de negócio podem ser reutilizadas em sistemas diferentes.

Solução

O padrão PDC-EJB utiliza um conjunto de classes para estruturar o código relacionado ao domínio de objetos do negócio e o código de acesso a dados, a fim de evitar a mistura de código relacionado a tais aspectos, obtendo, com isso, extensibilidade e reutilização das classes. Para tal, o padrão utiliza a separação das classes do sistema em dois tipos:

- classes para descrever os objetos de negócio, resultantes dos requisitos funcionais; e
- classes para manipulação e armazenamento de dados.

A comunicação entre esses dois tipos de classes é realizada através de interfaces que garantem uma maior independência do código de negócio em relação à forma como efetivamente são implementadas as operações de persistência (o acesso ao banco de dados ou outro mecanismo, como arquivos, por exemplo).

Estrutura

O diagrama de classes da Figura 4 destaca a estrutura do padrão PDC-EJB. As classes que denotam os elementos que fazem parte do padrão são explicadas a seguir.

- Fachada

  Esta classe representa todos os serviços do sistema e define uma interface que abstrai os objetos de negócio da aplicação [4]. Ela mantém uma referência para os vários objetos da classe `ColecaoDeNegocio` da aplicação e delega as chamadas para estes.

[3] Termo utilizado aqui, para descrever o meio no qual os objetos de negócio do sistema são armazenados, por exemplo, um banco de dados relacional.

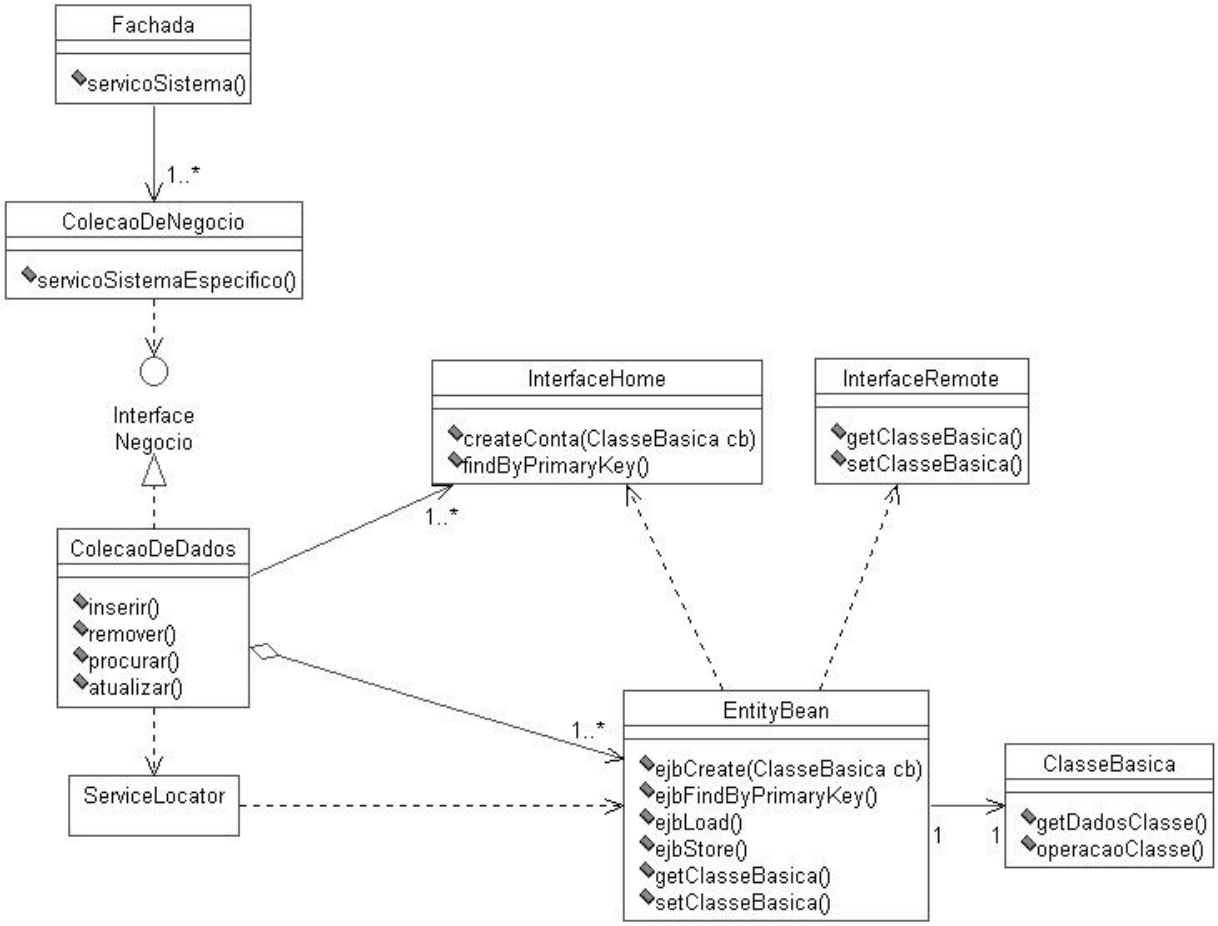


Figura 4: Estrutura do PDC-EJB.

- ClasseBasica

  A classe básica representa o objeto básico de negócio (por exemplo, conta, cliente) refletindo claramente o domínio do problema. Os métodos desta classe contêm somente operações relacionadas aos requisitos funcionais do sistema e métodos `get` e `set` para obter informações sobre os atributos desta classe.

- EntityBean

  Para cada classe básica da aplicação, deve existir uma classe *entity bean* correspondente. Esta classe possui operações para persistência dos objetos das classes básicas no sistema. A classe `EntityBean` possui uma dependência com sua classe básica correspondente e seus métodos manipulam os objetos desta. Porém, a classe básica não possui dependência com o *entity bean*, podendo ser reutilizada em outras aplicações.

- ColecaoDeNegocio

  Esta classe representa o agrupamento dos objetos básicos de negócio, tendo como operações a inserção, busca e exclusão de elementos do repositório, verificações ou testes de pré-condições relativos à estas manipulações e mais as operações que invocam as operações típicas de objetos de negócio.

- ColecaoDeDados

  A coleção de dados contém código de manipulação da estrutura de armazenamento persistente, correspondente a cada classe básica de negócio. O código dos métodos da mesma depende da API específica da plataforma utilizada para armazenamento (no caso, EJB é utilizado para persistência dos dados). Assim, mudanças na API de persistência não causam impacto na camada de negócio da aplicação, o impacto é centralizado nesta classe (uma vez que a interface negócio-dados isola essas mudanças). Uma classe coleção de dados implementa sua interface negocio-dados correspondente. Esta última é apresentada a seguir.

- InterfaceNegocioDados

  Esta interface possui assinaturas dos métodos de acesso aos dados, como inserção, atualização, consultas e exclusão. Esta interface estabelece uma comunicação entre os objetos das classes coleção de negócio e os objetos das classes coleção de dados, proporcionando extensibilidade. Uma classe `ColecaoDeNegocio` possui uma referência a esta interface. Desta forma, a classe `ColecaoDeNegocio` não precisa ser modificada quando a classe `ColecaoDeDados` mudar, desde que esta sempre implemente esta interface.

Os elementos que correspondem à fachada, coleção de negócio, coleção de dados e interface negócio-dados são implementados como classes Java "puras". Apesar de estar trabalhando com EJB, existem justificativas para não tornar tais classes *session beans*. O uso de *sessions beans* proporciona muitos benefícios. Para um *bean*, o *container* gerencia vários serviços de forma automática. Estes serviços envolvem gerenciamento de transações, controle do acesso concorrente, gerenciamento das instâncias no servidor e, por conseguinte, gerenciamento da memória, entre outros. A fim de fornecer todos esses serviços de forma transparente, o *container* executa uma vasta quantidade de processamento, incluindo geração de classes e mecanismos que visam auxiliá-lo no emprego correto e adequado dos serviços que gerencia.

Desta forma, toda vez que um método de um *bean* é requisitado, seja por um cliente externo ou mesmo por outro *bean*, o *container* intercepta cada chamada antes de reenviá-la para o objeto apropriado. Essa interceptação é necessária para que o *container* tenha conhecimento de todas as características do *bean* em questão: qual o tipo de mecanismo de transação, qual o atributo de transação para o método chamado, que componentes o *bean* referencia, que outros recursos utiliza, e assim por diante. Após ter acesso a todas essas informações, o *container* executa os processamentos devidos e finalmente repassa a chamada para a instância apropriada.

De fato, a execução de um método implica em vários processamentos que devem ser executados pelo *container* antes de passar a requisição da chamada para o *bean*. Em um sistema que utiliza muitas classes *session beans*, o desempenho do sistema tende a degradar. Por isso, as classes fachada, coleção de negócios, e interface negócio dados não são implementadas como *beans*.

Além das razões supracitadas relacionadas à eficiência, a implementação das classes de negócio como classes Java "puras", permite que estas possam reutilizadas em outras aplicações que utilizem uma plataforma de distribuição diferente, uma vez seu código não está atrelado a uma tecnologia de distribuição específica.

A coleção de dados também é implementada como uma classe Java, no entanto ela possui código relativo a API de EJB, uma vez que utiliza os serviços de persistência de *entity beans*. Esta classe, bem como a interface negócio-dados são importantes porque evitam que a coleção de negócio tenha aspectos específicos da API de EJB. Isto permite o desacoplamento entre a tecnologia utilizada para pesistência dos dados e camada de negócio do sistema.

Outro aspecto a ser considerado na estrutura do padrão PDC-EJB é o relacionamento entre as classes básicas de negócio e *entity beans*. *Entity beans* fornecem operações para persistir os dados e para realizar parte da lógica do negócio numa única classe. Isto resulta na mistura de papéis, ou seja, código de acesso a dados persistentes e código para manipulação da lógica de negócio são especificados na mesma classe. Os métodos de negócio de um *entity bean* são declarados em sua interface remota. Isto implica que toda invocação de um método de negócio de um *entity bean* é feita remotamente. Além disso, sempre que é feita uma chamada de método sobre a interface *home* de um *entity bean* (para criar ou localizar uma instância da entidade), o cliente recebe uma referência remota do objeto e não sua cópia. A abordagem com *entity beans* traz algumas consequências:

- O acesso concorrente ao *entity bean* é gerenciado pelo *container*

  Como cada cliente tem acesso a uma referência remota do *entity bean*, fica a cargo do *container* organizar o acesso e controlar a sincronização entre os clientes de modo a manter o estado do *bean* sempre coerente. Este é um dos grandes benefícios de *entity beans*, visto que o programador não precisa se preocupar em fornecer código necessário ao gerenciamento do acesso concorrente.

- Os métodos de negócio são declarados na interface remota do *bean*

  Isto implica que quando o cliente deseja executar um método de negócio, ele o faz de forma remota. Isto pode causar impacto no desempenho do sistema. Em particular, no caso da entidade possuir uma grande quantidade de informações que são acessadas por métodos `get` e `set`, pode-se ter um gargalo no tráfego dessas informações pela rede. Há um custo associado à invocação de métodos e à troca de dados remota.

- Mistura de conceitos

  Entity beans misturam código relacionado à forma como os dados são persistidos (fornecidos pelos métodos da interface *home*) com os métodos de negócio, fornecidos através da interface remota.

Com o padrão PDC-EJB é possível minimizar ou evitar o impacto causado pelos últimos itens. As classes básicas de negócio ficam responsáveis por processar parte da lógica de negócio e os *entity beans* assumem o papel de persistir os dados resultantes deste processamento. Com isso, é possível tornar clara a separação dos papéis. Para cada classe básica existe um *entity bean* associado. Para facilitar o entendimento, a parte da estrutura do padrão correspondente a este aspecto é ilustrada na Figura 5.

Nada é mudado com relação à classe básica; acrescenta-se um *entity bean* para facilitar a implementação da coleção de dados. A coleção de dados utiliza os serviços de persitência de *entity beans*. Os métodos `getClasseBasica` e `setClasseBasica` são utilizados pela

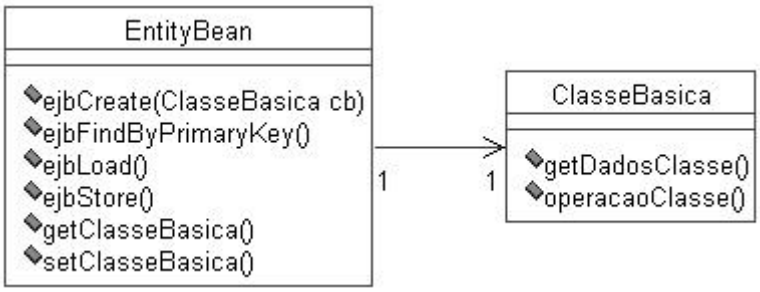


Figura 5: Estrutura para o uso de classes básicas e *entity beans*.

coleção de dados para retornar um clone do objeto `ClasseBasica` e atualizar o objeto `ClasseBasica`, respectivamente.

Para realizar esta abordagem, cada instância do *entity bean* deverá estar sincronizada com a instância da classe básica de negócio a fim de executar as alterações de forma coerente e mantendo a integridade dos dados a serem persistidos.

Com esta abordagem os clientes têm acesso às cópias dos objetos básicos de negócio em vez de referências remotas dos *entity beans*. Executam o processamento localmente, a partir destas cópias e o resultado deste processamento é persistido no banco com o auxílio de *entity beans*.

Pelo fato dos clientes manipularem cópias dos objetos básicos de negócio em vez de referências remotas de entity beans, o mecanismo de controle de concorrência automático fornecido por entity beans é perdido. Entretanto, os benefícios alcançados com a estrutura do padrão compensam essa perda uma vez que o controle de concorrência pode ser facilmente solucionado com o auxílio de mecanismos, tais como *timestamp* [13].

Dinâmica

A Figura 6 apresenta o diagrama de seqüência para um dos possíveis usos do PDC-EJB. Neste cenário, quando um método da `Fachada` é invocado, é realizada a delegação para um método da `ColecaoDeNegocio` (no exemplo, uma operação de consulta que recupera um objeto do banco de dados). O objeto da `ColecaoDeNegocio` executa possíveis validações e testes relativos aos dados informados para a consulta, e invoca a operação `procurar` sobre a `ColecaoDeDados`. Esta utiliza as operações de `EntityBean` para o acesso ao banco de dados. Através da operação `findByPrimaryKey` obtém-se a referência remota do objeto armazenado no banco. A partir desta referência, o objeto da classe básica é obtido através da operação `getClasseBasica` e retornado para o cliente, como resultado da consulta.

Conseqüências

A utilização do PDC-EJB traz as seguintes vantagens:

- Reutilização e extensibilidade

  Devido à estrutura modular do padrão, mudanças na camada de dados não causam impacto nas demais camadas. Interfaces entre a camada de negócio e a camada de dados provêem essa extensibilidade. Isto permite que a camada de negócio não tenha

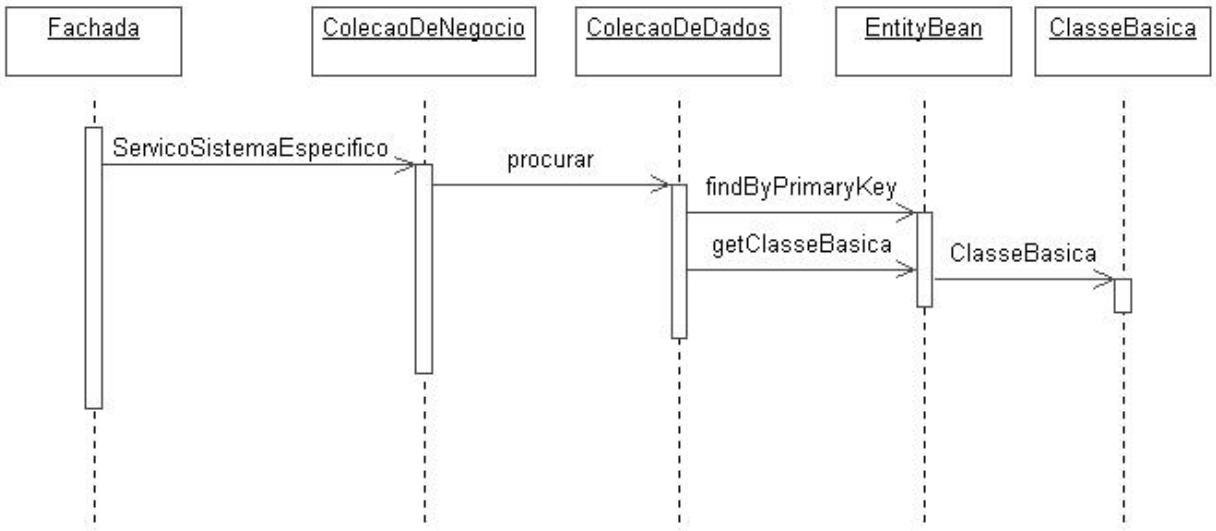


Figura 6: Diagrama de Sequência do PDC-EJB.

conhecimento se a tecnologia utilizada para persistir os dados é JDBC, EJB, etc. Além disso, a estrutura que o padrão apresenta possibilita que as classes básicas de negócio possam ser facilmente reutilizadas em outras aplicações que utilizem outras tecnologias de banco de dados.

- Redução do tráfego na rede

  Em vez de várias chamadas a métodos `get` para obter os atributos de um *bean*, o padrão fornece uma única chamada para obter todos os valores encapsulados em um objeto básico. Isso faz com que uma quantidade de dados seja transferida pela rede em uma única chamada remota. Isto sem dúvida diminui a carga imposta pelos acessos remotos e, consequentemente, melhora o desempenho do sistema.

- Facilidades para teste

  A classe básica possui apenas métodos de negócio e métodos "acessores", sem código de acesso a dados. Desta forma, fica mais fácil testar somente a funcionalidade do sistema usando uma versão volátil do mesmo, sem o uso de *entity beans* apenas com coleções de dados voláteis.

- Simplificação de *entity beans* e interfaces remotas

  Além dos métodos `get` e `set`, os *entity beans* possuem somente os métodos padrão para persistir os dados; os métodos de negócio são executados pela classe básica. O cliente do *entity bean*, no caso a coleção de dados, tem acesso ao objetos básicos de negócio a partir dos métodos `getClasseBasica` e `setClasseBasica`, os únicos métodos declarados na interface remota do *bean*. Esta estruturação também fornece um maior potencial para geração automática de código.

Por outro lado, o PDC-EJB apresenta as seguintes desvantagens:

- Não utilização do controle de concorrência de EJB

  De acordo com a estrutura do padrão, o cliente do sistema tem acesso aos clones dos objetos básicos e não às referências remotas dos *entity beans*. Os clientes executam modificações sobre cópias locais do objeto da classe básica. Uma vez que as modificações foram realizadas, o cliente invoca o método `setClasseBasica` (na verdade, quem invoca este método é a coleção de dados), passando o objeto modificado para o *entity bean*, e este se encarrega de atualizar os seus atributos e persistí-los no banco de dados. O problema acontece quando outros clientes requisitam o mesmo objeto.

  Apesar do *entity bean* atualizar os valores, este não está ciente dos vários clientes que obtiveram cópias do mesmo objeto e por isso não pode propagar a atualização do objeto para os vários clientes. Estes clientes acabam tendo instâncias de objetos que não refletem seu estado real no banco. No entanto o uso de mecanismos como *timestamp* [13], por exemplo, podem ser utilizados para resolver essa deficiência.

- Duplicação

  Os atributos da classe básica e do *entity bean* correspondente são duplicados. Isto se deve à limitação da especificação 1.1 de EJB, que exige que *entity beans* CMP declarem seus atributos públicos. De outra forma, o *bean* poderia herdar a classe básica. Os atributos da classe básica são declarados *private* por questões de encapsulamento. Isto implica que mudanças nos atributos da classe básica devem ser refletidas nos atributos do *entity bean*. No entanto, a mudança é localizada e poderia ter um apoio preconizado para manter a consistência.

- Produtividade

  Duplica-se o número de objetos que representam entidades persistentes: é necessário um *entity bean* e uma classe básica para cada entidade forte. Torna-se então necessário o uso de ferramentas para gerar o código do *entity bean* a partir da classe básica, por exemplo, bem como manter a consistência após alterações. Além disso, um gerador de código poderia também automatizar a criação dos adaptadores e classes auxiliares utilizadas na aplicação.

Exemplo

O diagrama de classes UML da Figura 7 ilustra os elementos que compõem o padrão através de uma simples aplicação bancária.

As classes `Banco`, `CadastroDeContas` e `Conta` correspondem aos objetos do domínio do problema. As classes que lidam com os aspectos de persistência dos dados e, portanto, fazem parte da camada de dados do sistema, são representadas pelas classes `IRepositorioConta`, `RepositorioDeContasEJB` e `ContaEJB`. A comunicação entre os dois tipos de classes é realizada através de interfaces. Este aspecto é importante porque permite estruturar os aspectos de forma modular reduzindo o impacto causado por possíveis modificações do sistema tanto para requisitos funcionais (como a introdução de novos serviços) quanto não funcionais (como adaptar o sistema para suportar outro mecanismo de persistência ou melhorar a *performance* das consultas).

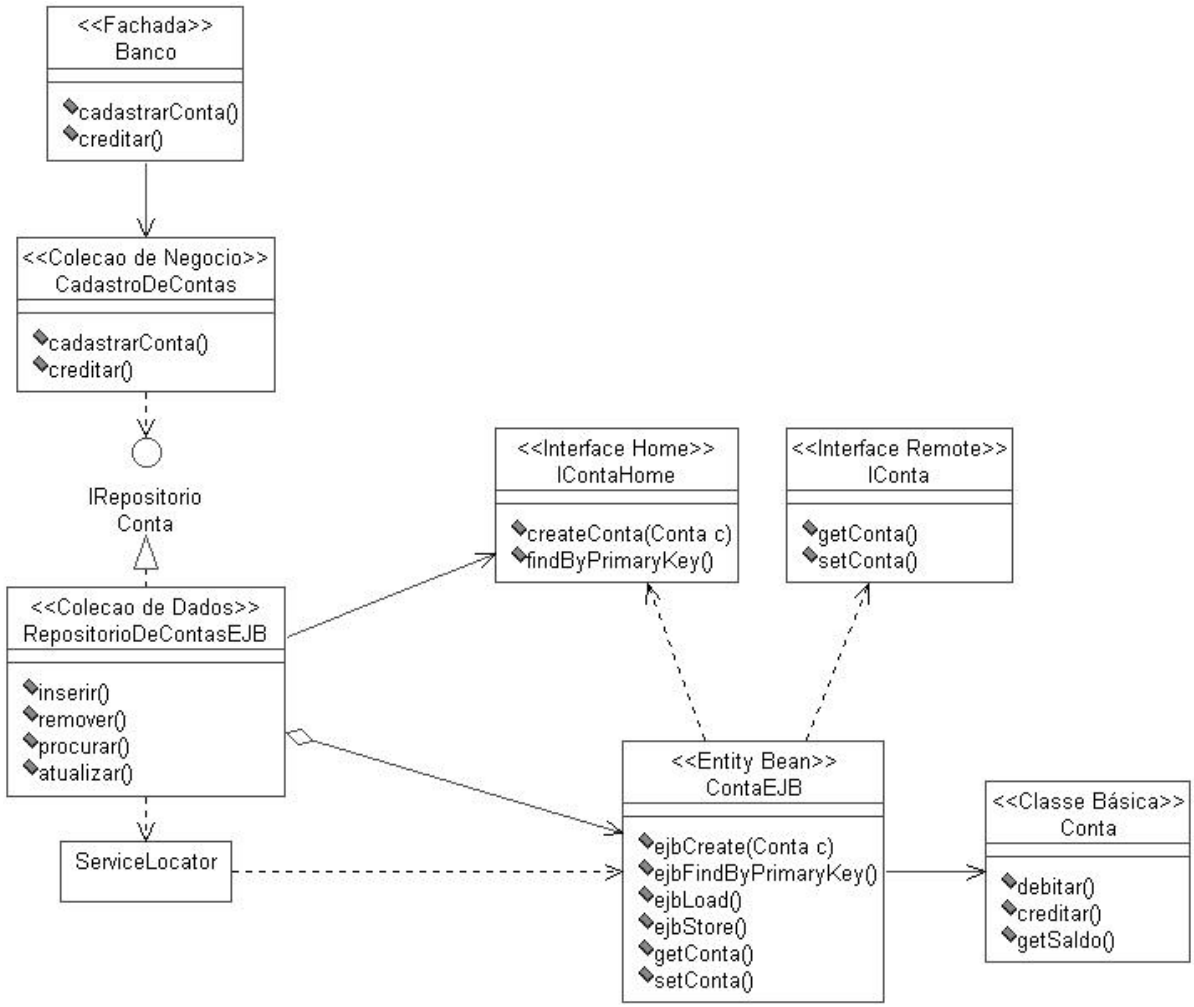


Figura 7: Exemplo de uma aplicação bancária estruturada de acordo com o padrão PDC-EJB.

Implementação

ServiceLocator para localização de *entity beans*

A tarefa para ter acesso a um componente EJB é comum para todos os clientes (externos ou internos) que precisam acessar seus serviços. Isto implica que muitos tipos de clientes repetidamente utilizam os serviços JNDI [10], uma API que fornece um conjunto de interfaces e classes para acessar uma vasta quantidade de recursos, entre os quais permite a localização de objetos remotos, o que resulta em código duplicado nos mesmos. Além disso, o processo necessário para localizar e obter referências remotas aos *homes* dos *beans* gasta recursos significativos do servidor de aplicação, o que pode causar impacto no desempenho do sistema. Neste contexto, a classe `ServiceLocator` [6], um padrão que visa abstrair a complexidade do processo de localização e criação dos *beans*, bem como melhorar o desempenho do sistema, é utilizada no padrão PDC-EJB para auxiliar o processo de criação das referências remotas de *entity beans*.

Transações

O padrão PDC-EJB é utilizado em conjunto com o padrão DAP-EJB, descrito da Seção 2. Neste caso, o adaptador destino, um *session bean*, é responsável por gerenciar as transações associadas aos seus métodos, os quais simplesmente delegam todas as invocações para os métodos da fachada.

A partir de atributos de transação especificados para os métodos do adaptador destino, o *container* gerencia o contexto transacional dos mesmos. A especificação garante que o contexto transacional é propagado a todos os objetos que participam na realização de uma operação, incluindo a fachada. Portanto, no padrão PDC-EJB, uma classe fachada não precisa especificar código para o tratamento de transações de seus métodos, uma vez que esta tarefa é delegada ao *container*.

Da mesma forma, aspectos relativos ao gerenciamento de segurança também ficam à cargo do padrão DAP-EJB. O gerenciamento de transações, bem como o gerenciamento de segurança são realizados pelo adaptador destino.

Código

Esta seção apresenta a implementação dos principais elementos do PDC-EJB. O exemplo utilizado é a aplicação bancária, introduzida na Figura 7. A explicação do PDC-EJB é realizada de forma *bottom up* a fim de facilitar o entendimento. Desta forma, o primeiro elemento a ser apresentado é a classe básica de negócio `Conta`, a qual reflete o domínio do problema.

A classe `Conta` é uma classe Java que implementa a interface `Serializable` [8]. Não existem métodos nesta interface e ela simplesmente indica para o sistema que um objeto pode ser transformado em um *stream* de bytes para poder ser transmitido pela rede. Os atributos

```
public class Conta implements java.io.Serializable {
  private String numero;
  private double saldo;
```

`numero` e `saldo` são declarados com visibilidade *private* por questões de encapsulamento e são inicializados no construtor da classe. Além disso, a classe básica também declara um construtor vazio.

Para cada atributo declarado na classe básica, são declarados métodos `get` e `set` correspondentes para a obtenção de informações sobre os atributos da classe. Os métodos

```
  public String getNumero() { return numero; }

  public double getSaldo() { return saldo; }
```

`getNumero` e `getSaldo` representam os métodos "acessores"para os atributos `numero` e `saldo`, respectivamente. Além destes, os métodos `set` para cada atributo também são declarados.

Além de métodos "acessores", a classe básica também possui métodos de negócio correspondentes ao domínio da aplicação. O método `creditar`, por exemplo,

```
    public void creditar(double valor) {
     saldo += valor;
    }
  }
```

corresponde a uma operação de negócio da classe `Conta`. Outros métodos de negócio, além de `creditar` também devem declarados nesta classe.

O *entity bean* `ContaEJB`[4] e suas interfaces *home* e remota, são apresentados. A interface *home* contém os métodos de acesso ao banco de dados `create` e `findByPrimaryKey`. Estes são métodos padrão de EJB utilizados para criar e localizar uma entidade banco, respectivamente.

```
public interface IContaHome extends EJBHome {
   public IConta create(Conta conta)
       throws RemoteException, CreateException;

   public IConta findByPrimaryKey(String numero)
       throws FinderException, RemoteException;
```

O tipo de retorno de tais métodos é a referência remota do *entity bean*. A exceção `RemoteException` é declarada na assinatura dos métodos da interface `IContaHome` pelo fato desta tratar-se de uma interface remota. `CreateException` e `FinderException` são exceções especificas de EJB e também devem ser declaradas nas assinaturas dos métodos `create` e `findByPrimaryKey`, respectivamante.

Além dos métodos específicos de EJB, outros métodos `findXX` podem ser especificados, como por exemplo o método `findByAll`

```
   public Collection  findByAll() throws FinderException, RemoteException;
}
```

que retorna uma coleção de referências a todas as entidades armazenadas no banco de dados.

A interface remota `IConta` declara somente métodos `get` e `set` para a classe básica `Conta`.

```
public interface IConta extends EJBObject {
  public Conta getConta() throws RemoteException;
  public void setConta(Conta conta) throws RemoteException;
}
```

Os métodos de negócio são executados localmente, a partir dos clones das classes básicas obtidos através do método `getConta` da interface remota. Após o processamento local dos métodos de negócio, o resultado pode ser atualizado através do método `setConta`.

A classe `ContaEJB` implementa a interface `EntityBean`, específica de EJB, a qual fornece métodos para a manipulação das entidades no banco, pelo *container*.

Os atributos do *entity bean* que devem ser mapeados para tabelas no banco de dados, são declarados públicos. Esta é uma restrição da especificação 1.1 de EJB. Neste caso, os dois atributos apresentados, `numero` e `saldo` são declarados com visibilidade `public`.

[4] O código do *entity bean* no exemplo apresentado utiliza persistência gerenciada pelo *container* (CMP).

```
public class ContaEJB implements EntityBean {
  public String numero;
  public double saldo;
  private Conta conta;
  private EntityContext context;
```

Um atributo do tipo da interface `EntityContext` também deve ser declarado para que o *container* possa ter acesso às informações dos *entity beans*. O *entity bean* também declara `conta` como atributo por questões de desempenho, para evitar que sempre seja necessário criar um objeto da classe `Conta` antes de enviá-lo para o cliente, através do método `getConta`.

Os atributos de `ContaEJB` são inicializados no método `ejbCreate`, o qual corresponde ao método `create` da interface `IContaHome` e é responsável por criar uma entidade no banco. O método recebe como parâmetro um objeto da classe `Conta`. Os atributos do *bean* são inicializados a partir do estado do atributo `c`. Desta forma, quando da criação de uma entidade no banco de dados, o estado da instância do *entity bean* (`ContaEJB`) é sincronizado com o estado da instância de sua classe básica (`Conta`) correspondente.

```
public String ejbCreate(Conta c) throws CreateException {
    numero = c.getNumero();
    saldo = c.getSaldo();
  return null;
}
```

Um objeto da classe básica é passado como parâmetro para o método `ejbCreate` de seu *entity bean* correspondente para evitar o tráfego de parâmetros pela rede, em vez de enviar vários atributos pela rede, o objeto completo e enviado, diminuindo o *overhead* associado. Após os atributos do *bean* serem inicializados, estes são inseridos no banco de dados. O tipo de retorno do método é um objeto do tipo da chave primária armazenada na tabela. No código apresentado, o retorno é `null` porque em persistência gerenciada pelo *container* (CMP), o *container* é responsável por gerar a chave primária da tabela é retorná-la como resultado da inserção. O código para inserção no banco é gerado pelo *container*.

Os métodos oriundos da interface `EntityBean`, são declarados e implementados pelas classes geradas pelo *container*, tendo seus métodos com o corpo vazio. Eles são responsáveis pela persistência da entidade no banco de dados,

```
void ejbStore() { }
void ejbLoad() { }
void ejbRemove() { }
```

sendo invocados pelo *container* durante a execução dos métodos pelo cliente.

Os métodos `getConta` e `setConta` são responsáveis por manter o estado das instâncias dos objetos de negócio (Conta, no exemplo) sincronizado com o estado das instâncias dos *entity beans* (ContaEJB, no exemplo).

```
public void setConta(Conta c) {
     numero = c.getNumero();
```

```
        saldo = c.getSaldo();
        conta = c;
  }

  public Conta getConta() {
      conta.setNumero(this.numero);
      conta.setSaldo(this.saldo);
      return conta;
  }

}
```

A classe `RepositorioDeContasEJB` representa a coleção de dados do padrão PDC-EJB. É uma classe Java "pura"e utiliza os serviços de persistência de *entity beans*. Isto acontece devido ao fato de *entity beans* representarem entidades persistentes e, portanto, em uma aplicação EJB são responsáveis por persistir tais entidades no banco de dados.

Esta classe implementa a interface negócio-dados `IRepositorioConta`, a qual fornece métodos para manipular os dados armazenados no banco de dados. Esta interface é apresentada mais adiante. Um atributo da interface *home* do *entity bean* é declarado na coleção de dados por questões de eficiência.

O acesso às referências dos *homes* dos *entity beans* é realizado com o auxílio do método auxiliar `getHome`. Este método faz acesso ao método de mesmo nome (getHome) da classe `ServiceLocator`. Através deste método, é possível localizar um *entity bean* mantendo uma única instância da referência remota à sua interface *home* durante a execução dos clientes.

```
  class RepositorioDeContasEJB implements IRepositorioConta {
      private IContaHome home;
        private IContaHome getHome()
           throws ServiceLocatorException {
           if (home ==null) {
              home= (IContaHome)
                     ServiceLocator.getInstance().getHome("conta",
                                                    IContaHome.class);
          }
        return home;
      }
}
```

Todos os métodos desta classe que acessam as operações de persistência de *entity beans*, utilizam o método auxiliar `getHome` para localizar o *entity bean*.

O método `inserir` é utilizado para incluir uma entidade no banco de dados. Recebe como parâmetro um objeto da classe básica (`Conta`) e chama o método `create` do *entity bean* passando `conta` como parâmetro. As exceções específicas de EJB relacionadas ao mecanismo de armazenamento de dados, são substituídas na coleção de dados pela exceção genérica `RepositorioException`. Por isso, esta é declarada na assinatura do método. A troca de exceções permite isolar a coleção de negócio da API de EJB.

```
  void inserir(Conta conta)  throws RepositorioException{
    try {
      IContaRemote contaBean = getHome().create(conta);
    } catch (Exception e) {
              throw new RepositorioException(e);
      }
  }
```

O método `procurar` localiza uma entidade no banco a partir de `codigo` e retorna um objeto da classe básica `Conta`. Para tal, utiliza o método de EJB `findByPrimaryKey`, o qual retorna uma referência remota da entidade armazenada no banco. Após obter a referência remota, o método `getConta` é invocado.

```
      public Conta procurar(String codigo) throws  RepositorioException  {
         Conta c = null;
           try {
            IContaRemote contaBean = getHome().findByPrimaryKey(codigo);
             c = contaBean.getConta();
          } catch (Exception e) {
                throw new RepositorioException(e);
          }
           return c;
       }
}
```

Este permite obter um clone do objeto `Conta`, o qual é retornando para o cliente. Isto permite que os clientes da aplicação manipulem cópias dos objetos remotos, em vez de manipulá-los remotamente, melhorando com isso, o desempenho, visto que invocação de métodos e manipulação destes remotamente são tarefas que degradam o desempenho do sistema.

Assim, a coleção de dados isola dos clientes das camadas acima, o acesso às referências remotas dos *entity beans*, uma vez que repassa para as camadas superiores os clones das classes básicas em vez de referências remotas. Isto pode trazer problemas de inconsistência dos dados acessados por clientes concorrentes. No entanto este aspecto pode ser facilmente solucionado com a introdução de mecanismos como *timestamp*.

A interface `IRepositorioConta` é a interface negócio-dados. Esta interface é implementada pela classe RepositorioDeContasEJB.

```
public interface IRepositorioConta {
    public Conta procurar(String numero)
        throws RepositorioException;
    public void atualizar(Conta conta)
        throws RepositorioException;
    public void inserir(Conta conta)
        throws RepositorioException;
    public Boolean existe (String numero)
        throws RepositorioException;
}
```

A classe coleção de negócio corresponde, no exemplo, à classe `CadastroDeContas`. Esta classe representa uma coleção de objetos da aplicação e fornece serviços para manipular um cadastro de contas. Esta classe utiliza os serviços da coleção de dados através da interface `IRepositorioConta` e seu código é apresentado a seguir.

```
public class CadastroDeContas {
  private IRepositorioConta repConta;

    public CadastroDeContas(IRepositorioConta rep) {
      repConta = rep;
    }
```

O construtor de `CadastroDeContas` recebe como argumento um objeto que implementa a interface negócio-dados. A partir do atributo `repConta`, a coleção de negócio invoca os métodos da coleção de dados.

Duas das operações para esta classe são apresentadas. O método `cadastrarConta` é utilizado para inserir um objeto `Conta` no sistema. Para tal, primeiramente é verificado se um objeto de mesmo número já existe, lançando a exceção `ContaJaExisteException` em caso positivo. Caso o objeto a ser cadastrado ainda não exista no sistema, o método da coleção de negócio invoca o método `inserir` da coleção de dados, através da interface negócio-dados.

```
    public void cadastrarConta(Conta conta)
             throws ContaJaExisteException, RepositorioException{
        if (repConta.existe(conta.getNumero()))
              throw new ContaJaExisteException();
        else
          repConta.inserir(conta);

    }
```

O método `creditar` consulta o sistema para uma determinada conta e, se a consulta for bem sucedida, um valor é adicionado ao saldo da conta e as informações são atualizadas na coleção de dados. Todavia, se a conta não existe, uma exceção é lançada.

```
   public void creditar(String numero,double valor)
            throws ContaNaoExisteException, RepositorioException {
      if (repConta.existe(numero)){
          Conta c = repConta.procurar(numero);
          c.creditar(valor);
          repConta.atualiza(c);
       }
          else throw new ContaNaoExisteException();
       }
  ...
}
```

A implementação dos demais métodos da coleção de negócio é feita de forma similar, e são omitidos aqui por questões de brevidade.

A classe `Banco` (fachada do sistema) contém todos os serviços oferecidos pela aplicação. Esta classe possui uma referência à classe `CadastroDeContas`. O construtor cria um objeto do tipo coleção de negócio e o atribui ao atributo `cadConta`. A partir deste ponto, a fachada pode invocar os métodos da coleção de negócio.

```
class Banco {
  private CadastroDeContas cadConta;
      Banco(){
        cadConta =
        new CadastroDeContas(new RepositorioDeContasEJB());
     }
```

Dois dos métodos da fachada são apresentados. O método `cadastrarConta` invoca o método `cadastrarConta` da coleção de negócio.

```
   void cadastrarConta(Conta conta)
     throws RepositorioException, ContaJaExisteException {
       cadConta.cadastrarConta(conta);
 }
```

O mesmo acontece com o método `creditar`

```
  void creditar(String numero, double saldo)
    throws RepositorioException, ContaNaoExisteException {

      cadConta.creditar(numero, saldo);
}
  ...
}
```

que também utiliza o atributo `cadConta` para invocar métodos da coleção de negócio, no caso, o método de mesmo nome `creditar`. As exceções `ContaJaExisteException` e `ContaNaoExisteException` são exceções específicas da aplicação.

Usos Conhecidos

Como parte do padrão DAP-EJB, o PDC-EJB é também utilizado nos mesmos sistemas que este.

Para o sistema de informação do serviço público de saúde, o PDC-EJB é utilizado tal como descrito nesta seção. Esta aplicação tem como funcionalidade, receber e controlar as denúncias, notificações, além de fornecer informações importantes sobre o sistema público de saúde, que sejam do interesse da população.

O PDC-EJB é também utilizado no sistema que fornece serviços para o gerenciamento de contabilidade, controle de acesso e serviços financeiros. Neste sistema, a coleção de negócio da aplicação é opcional em algumas partes da aplicação. Nos casos onde a coleção de negócio é opcional, o adaptador destino, acessa a coleção de dados, através da interface negócio-dados do sistema.

Outros possíveis usos do PDC-EJB:

- Um sistema para gerenciar clientes de uma empresa de telecomunicação. O sistema é capaz de registrar telefones móveis, gerenciar informações de clientes e a configuração dos serviços de telefonia. Este sistema pode ser utilizado via *Web*.

- Um sistema para provas interativas. Este sistema tem sido utilizado para fornecer diferentes tipos de provas, tais como simulados baseados em exames de seleção para a universidade, ajudando os alunos a avaliar seus conhecimentos antes de realizarem exames reais.

- Um sistema de supermercado complexo. Este sistema será usado em vários supermercados e já está sendo utilizado em outras empresas do mesmo ramo.

Padrões Relacionados

- *Persistent Data Collections* (PDC) [11]. Este padrão também foi adaptado para o padrão apresentado nesta seção. O PDC promove código modular fornecendo um conjunto de classes e interfaces que separam o código de acesso a dados do código de negócio e interface com o usuário [11]. O padrão para EJB também observa tais aspectos com a diferença que não faz uso de uma classe auxiliar (MecanismoDePersistencia) para gerenciamento das transações e conexão com banco, como faz o PDC. Além disso, a classe básica é associada a um *entity bean*. Neste padrão, a primeira possui métodos `get` e `set` e métodos de negócio, enquanto a segunda possui somente métodos de acesso ao banco e métodos que permitem obter clones dos objetos das classes básicas associadas.

- Value Object [6]. É similar à classe básica apresentada no PDC-EJB. Este padrão tem como função encapsular os dados do negócio, ou seja, em vez de um cliente fazer várias requisições remotas a métodos `get` e `set` para obter os dados da entidade, ele o faz através de um único método que é utilizado para encapsular todos os dados necessários à requisição do cliente.

  Apesar de fornecer um mecanismo que melhora consideravelmente o desempenho do sistema, uma vez que evita o fluxo de chamadas remotas à métodos `get` e `set`, os métodos de negócio continuam sendo executados remotamente, uma vez que permanecem sendo declarados na interface remota do *bean*. Além disso, o método `ejbCreate` recebe os atributos do bean como parâmetro e não o objeto correspondente à sua classe básica, como sugere o padrão apresentado aqui.

- *Facade* [7]. A classe `Fachada` do PDC-EJB é a implementação direta do padrão *Facade*.

- *Singleton* [7]. Por questões de eficiência, os objetos da classe `Fachada` são implementados como *Singleton*. Desta forma, geralmente somente um objeto fachada é requerido na aplicação.

- *Bridge* [7]. A interface negócio-dados do padrão PDC-EJB é implementada como *Bridge*. Desta forma, ela é utilizada para permitir a comunicação entre as camadas de negócio e dados mantendo a primeira isolada da API de persistência da aplicação.

Agradecimentos

Nossos agradecinmentos especiais a Márcio Barros, nosso *shepherd*, pelos comentários e sugestões importantes que proporcionaram melhorias no nosso padrão.

Referências

[1] Vander Alves. Desenvolvimento Progressivo de Programas Distribuídos Orientados a Objetos. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, Fevereiro 2001.

[2] Vander Alves and Paulo Borba. Distributed Adapters Pattern: A Design Pattern for Object–Oriented Distributed Applications. In *First Latin American Conference on Pattern Languages Programming, Sugarloaf PLoP*, Rio de Janeiro, Brazil, 3th–5th October 2001.

[3] Grady Booch et al. *The Unified Modeling Language User Guide*. Object Technology. Addison-Wesley, first edition, 1999.

[4] Paulo Borba, Saulo Araújo, Hednilson Bezerra, Marconi Lima, and Sérgio Soares. Progressive implementation of distributed Java applications. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 40–47, Los Angeles, USA, 17th–18th May 1999.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern–Oriented Software Architecture:A System of Patterns, volume 1*. John Wiley & Sons, 1996.

[6] John Crupi DeepPAK Alur and Dan Malks. *core J2EE Patterns –Best Practices and Design Strategies*. Prentice Hall, first edition, 2001.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison–Wesley, second edition, June 2000.

[9] Marty Hall. *Core Servlets and JavaServer Pages*. Prentice-Hall, second edition, 2000.

[10] Rosanna Lee and Scott Seligman. *JNDI API Tutorial and Reference: Building Directory-Enabled Java(TM) Applications*. Addison-Wesley, 2000.

[11] Tiago Massoni, Vander Alves, Sérgio Soares, and Paulo Borba. PDC: Persistent Data Collections pattern. In *First Latin American Conference on Pattern Languages Programming, Sugarloaf PLoP*, Rio de Janeiro, Brazil, 3th–5th October 2001.

[12] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern–Oriented Software Architecture: Patterns for Concurrent and Networked Objects, volume 2*. John Wiley & Sons, 2000.

[13] Sérgio Soares. Desenvolvimento Progressivo de Programas Concorrentes Orientados a Objetos. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, Fevereiro 2001.

[14] Sun Microsystems. The Enterprise JavaBeans 1.1 Specification. Disponível em http://-java.sun.com/products/ejb/docs.html, Outubro 2000.

[15] Seth White, Maydene Fisher, Rick Cattell, Graham Hamilton, and Mark Hapner. *JDBC(tm) API Tutorial and Reference: Universal Data Access for the Java(tm) 2 Platform*. Addison–Wesley, second edition, June 1999.

PaDA: A Pattern for Distribution Aspects

Sérgio Soares*
Universidade Católica de Pernambuco
Departamento de Estatística e Informática
Centro de Informática
Universidade Federal de Pernambuco

Paulo Borba†
Centro de Informática
Universidade Federal de Pernambuco

Abstract

This paper presents a pattern that provides a structure for implementing distribution using AOP — aspect-oriented programming. The main goal is to achieve better separation of concerns avoiding tangled code (code with different concerns interlacing to each other) and spread code (code regarding one concern scattered in several units of the system). Therefore, system modularity, and hence, maintainability and extensibility are increased. The paper also presents an example of distribution aspects using AspectJ, an aspect-oriented extension to Java.

Intent

PaDA (Pattern for Distribution Aspects) provides a structure for implementing distribution code by achieving better separation of concerns. This is obtained through the use of aspect-oriented programming [7]. It increases system modularity, and hence, maintainability and extensibility.

Context

When implementing a distributed system that requires high modularity, meaning that the system should be independent of the distribution concern. To achieve better separation of concern we should use aspect-oriented programming by applying *PaDA*. An aspect defines a crosscutting concern, for example, distribution, which is automatically woven to a system changing its original behavior. Therefore, the system should be implemented in a programming language that has an aspect-oriented like extension. Examples of these languages with the respective aspect-oriented extensions are the following:

- Java — AspectJ [10], HyperJ [12], DemeterJ [11], Composition Filters [3];

*Supported by CAPES. Emails: scbs@cin.ufpe.br, sergio@dei.unicap.br

†Partially supported by CNPq, grant 521994/96-9. Email: phmb@cin.ufpe.br

- C++ — AspectC [6], Composition Filters [3];
- Smalltalk — AspectS [9], Composition Filters [3].

Problem

Tangled code (code with different concerns interlacing to each other) and spread code (code regarding one concern scattered in several units of the system) decrease system modularity. Therefore, maintainability and extensibility are also decreased.

Forces

To distribute a system, *PaDA* balances the following forces:

- Remote communication. The communication between two components of a system should be remote in order to allow several clients accessing the system, considering that the user interface is the distributed part of the system.
- API independence. The system should be completely independent of the communication API and middleware to facilitate system maintenance, as communication code is not tangled with business or user interface code. This also allows changing the communication API without impacting other system code.
- A same system can use different middleware at the same time. This would allow, for instance, two clients accessing the system, one using RMI and the other CORBA.
- Dynamical middleware changing. The system should allow changing the middleware without changing or recompiling its source code.
- Facilitate functional tests. Functional tests are easier by testing the system with its local version; therefore, distribution code errors will not affect the tests.

Solution

In order to solve the problem previously presented, *PaDA* uses aspect-oriented programming [7] to define distribution aspects [16] that can be woven to the system core source code. This separation of concern is achieved by defining aspects to implement a specific concern. After identifying and implementing the crosscutting concerns of a system, they can be automatically composed (woven) with the system source code, resulting on the system version with the required concerns.

Figure 1 illustrates the aspectual decomposition, which identifies the crosscutting concerns of a system, and the aspectual recomposition, or weaving, which composes the identified concerns with the system to obtain the final version with the required functions. In our case, *PaDA* defines just one concern (distribution), which is implemented by three aspects, as we show in next section.

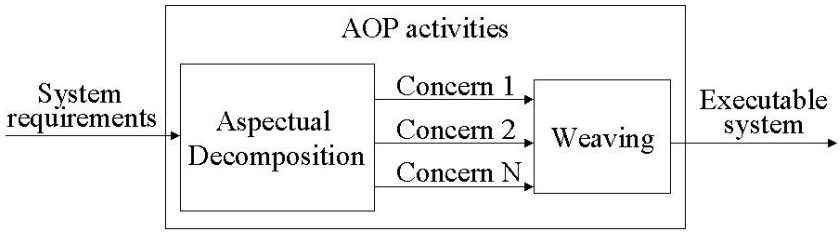


Figure 1: AOP development phases.

Structure

The *PaDA* pattern defines three aspects: one to crosscut the target component (server), another to crosscut the source components (client classes), and the third crosscuts both target and source component to provide the exception handling, as shown in Figure 2. In fact, the third aspect defines a concern that is crosscutting to distribution itself, namely exception handling. In fact, the `ServerSide` aspect might crosscuts others classes that are return types or arguments type of the target component methods.

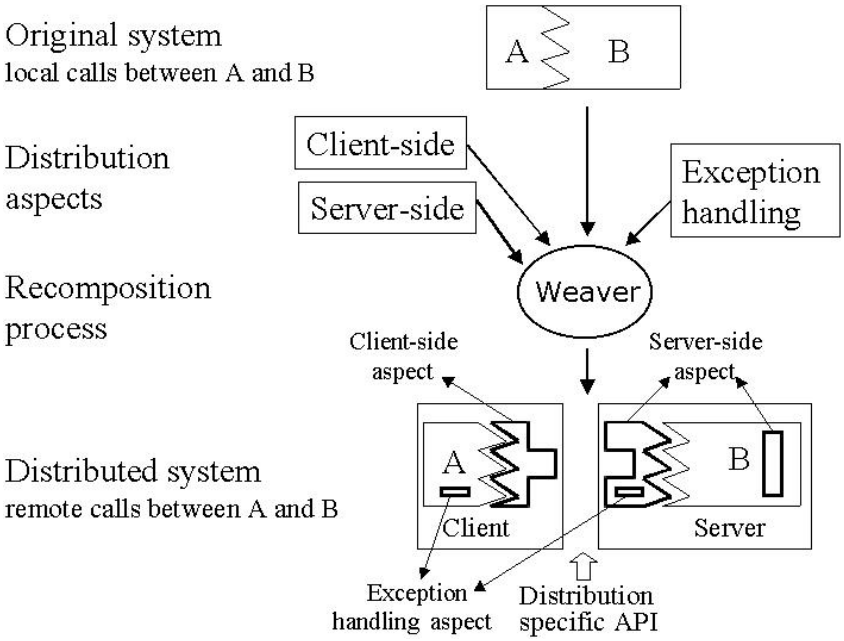


Figure 2: *PaDA*'s structure.

Figure 3 presents a UML class diagram that shows the aspects and their with the components to allow them to be remotely accessed. In that figure, `TargeComponent` is the one to be remotely accessed by instances of `SourceComponent`.

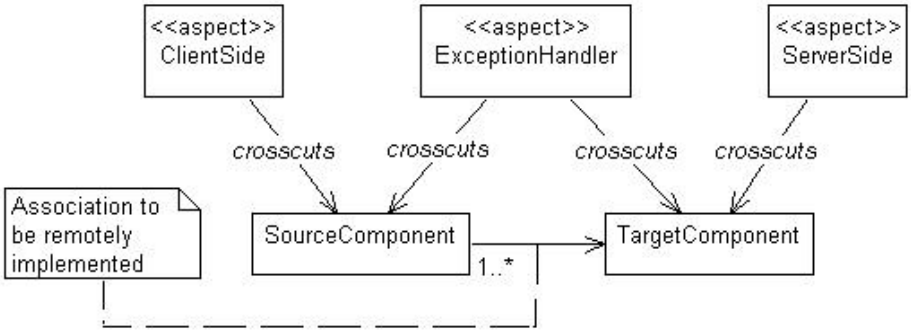


Figure 3: *PaDA* class diagram.

Dynamics

Figure 4 shows a sequence diagram of what is the original system behavior: a `SourceComponent` instance makes local calls to some methods of a `TargetComponent` instance.

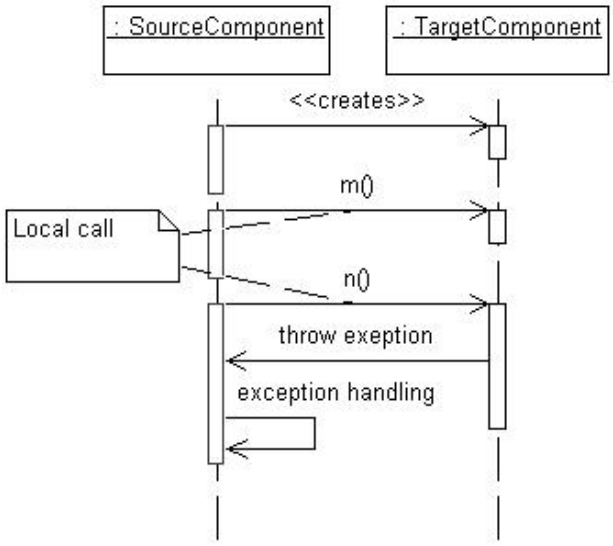


Figure 4: Original system behavior.

Figure 5 is the sequence diagram that states what is the behavior after weaving the aspects to the system: `SourceComponent` local calls are intercepted by the `ClientSide` aspect that gets the reference to the remote instance and redirect the local call to it. Note that the `ServerSide` aspect creates and makes the remote instance (a `TargetComponent` instance) available to response remote calls.

If the remote call raises an exception, like in the `n` method call, the `ExceptionHandler` aspect wraps the exception to an unchecked one and throws it, in the server-side. Note that the message that wraps and throws the unchecked exception is a message to the `ExceptionHandler` aspect itself, because the aspect is also responsible for catching the unchecked exception providing the necessary handling in the client-side (`SourceComponent`).

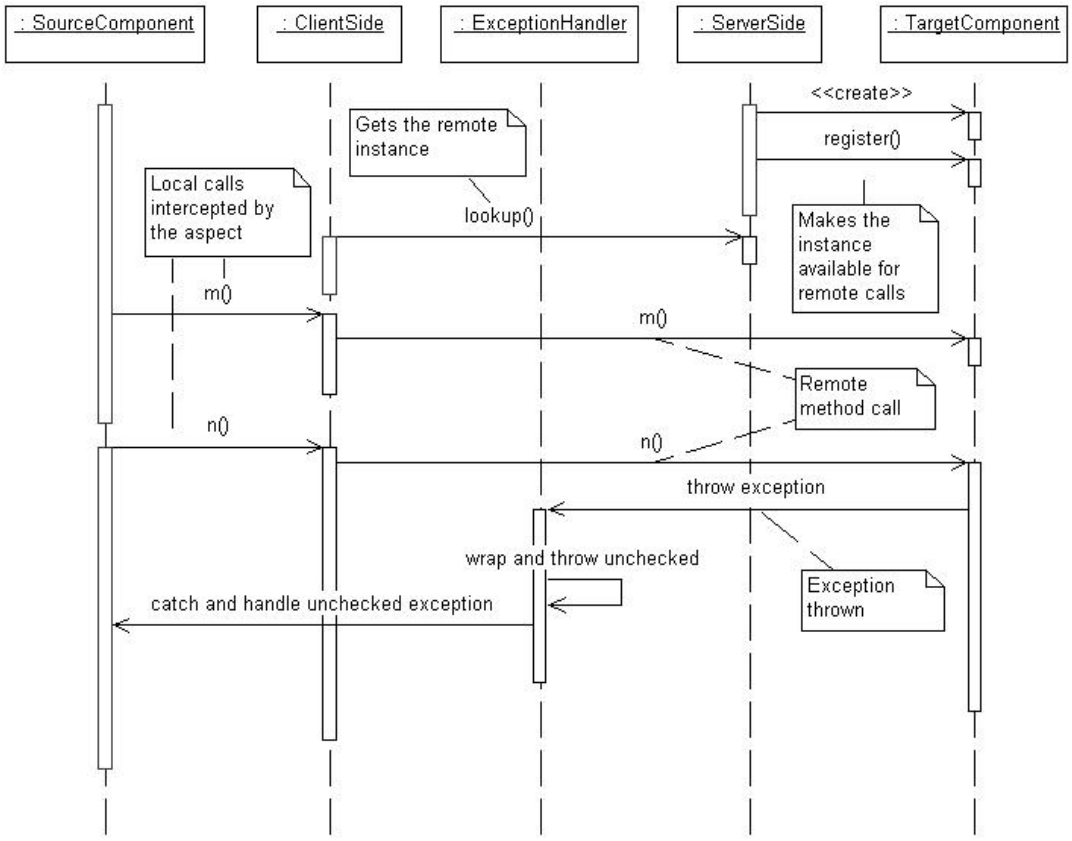


Figure 5: System behavior after applying *PaDA*.

Consequences

The *PaDA* pattern has several benefits:

- *Distributed Implementation.* The pattern provides remote communication between two components of a system;
- *Modularity. PaDA* structures the distribution code in aspects, which is completely separated of the system code, making the system source code API-independent;
- *Maintenance and extensibility.* As the distribution code is completed separated of the system code, changing the communication API is simpler and has no impact in the system code. The programmers should just write another distribution aspects, to the new specific API, or change the aspects already implemented to correct errors and them woven it to the original system source code.
- *Incremental implementation. PaDA* allows incremental implementation [15]. The system can be completely implemented and tested before implementing the distribution aspects, since the distribution aspects are separated from the system source code. This abstraction increases productivity, since the programmers should not take care about distribution problems. This incremental implementation also allows requirements validation without the impact of distribution.

- *Additional separation of concern.* *PaDA* structure defines exception handling as a crosscutting concern, which is not done by object-oriented programming techniques. Therefore, the exception handling can be changed without impacting in the original system source code and in the distribution aspects, or in others aspects that can be implemented, as the system requires.

- *Facilitate testing of functional requirements.* Tests of the functional requirements can be done easily if made using the system without the distribution. The full separation of concerns preserves the original system source code. This means that the distribution aspect is added to the system just if the composition process (weaving) is executed. Therefore, to obtain the monolithic system, just use the original source code, or remove the distribution aspects from the weaving, in case of the need of another concern, like data management.

The *PaDA* pattern also has the following liabilities:

- *New programming paradigm.* The pattern uses a new programming technique that implies in learning a new programming paradigm to use the pattern. Another impact of being a new programming paradigm is regarding the separation of code that usually was together in the same module. The programmer of the functional requirements cannot see the resultant code that will implement the required concern, decreasing code legibility.

- *Increased number of modules.* *PaDA* adds three new modules into the system, increasing the modules management complexity.

- *Increased bytecode.* Each aspect definition will result in a class after woven it into the system, which will increase the system bytecode.

- *Name dependence.* The aspects definition depends of the system classes, methods, attributes, and argument names, which decreases the aspects reuse. However, tools can mostly automate the aspects definition, increasing the aspects productivity and reuse.

- *Dynamic change of middleware.* At the moment, the AOP languages do not allow dynamic crosscutting, which does not allow changing the distribution protocol at execution time. This can be done by other design pattern, *DAP* [1], however, without achieving the separation of concerns we achieve with *PaDA*.

- *Allow using a same system through different middleware.* The idea of AOP is generate versions of a system including concerns. The feature of using a same system through different middleware can be achieved if several versions of the system were generated. However, this implies in having several instances of the system (server-side) executing, beside a single one, which may affect or invalidate concurrency control. On the other hand, *DAP* [1] can do this easily.

Implementation

The *PaDA* pattern implementation is composed of four major steps:

- Identify the components, server and client, to have the communication between them distributed.
- Write the server-side aspect. The server-side aspect is responsible to use specific distribution API code changing the server component, making it available to response remote calls. This aspect may also have to change others components used as parameters or return values of the server component, depending of the distribution API.
- Write the client-side aspect. The client-side aspects are responsible to intercept the original local calls made by the client component redirecting them to remote calls made to the remote component (server).
- Write the exception handler aspect. The exception handler aspect is responsible handle with new exceptions added by the aspects definition. These exceptions raised in the server-side are wrapped to an unchecked exception to throw them without changing the signature of the original system source code. Therefore, the exception handler aspect should also provide the necessary handling in the client-side classes.

Example

To exemplify the pattern we now consider a banking application and the RMI API to distribute the communication. Figure 6 presents a UML class diagram that models the banking example.

The `BankServlet` class is a servlet Java that provides a HTML and JavaScript user interface making requests to the `Bank` object. This is the communication to be distributed, therefore the `ServerSide` aspect should crosscuts the `Bank` class and the `ClientSide` aspect should crosscuts the `BankServlet` class. The `Bank` class is the system *Facade* [8] and has attributes like accounts and customers records

```
public class Bank {
  private AccountRecord accounts;
  public void deposit(String number, double value)
      throws AccountNotFoundException {
    accounts.deposit(number,value);
  } ...
}
```

and the operations to manipulate them.

In AspectJ the aspects can affect the dynamic structure of a program changing the way a program executes, by intercepting points of the program execution flow, called *join points*, and adding behavior *before*, *after*, or *around* (instead of) the *join point*. Examples of *join points* are method calls, method executions, instantiations, constructor

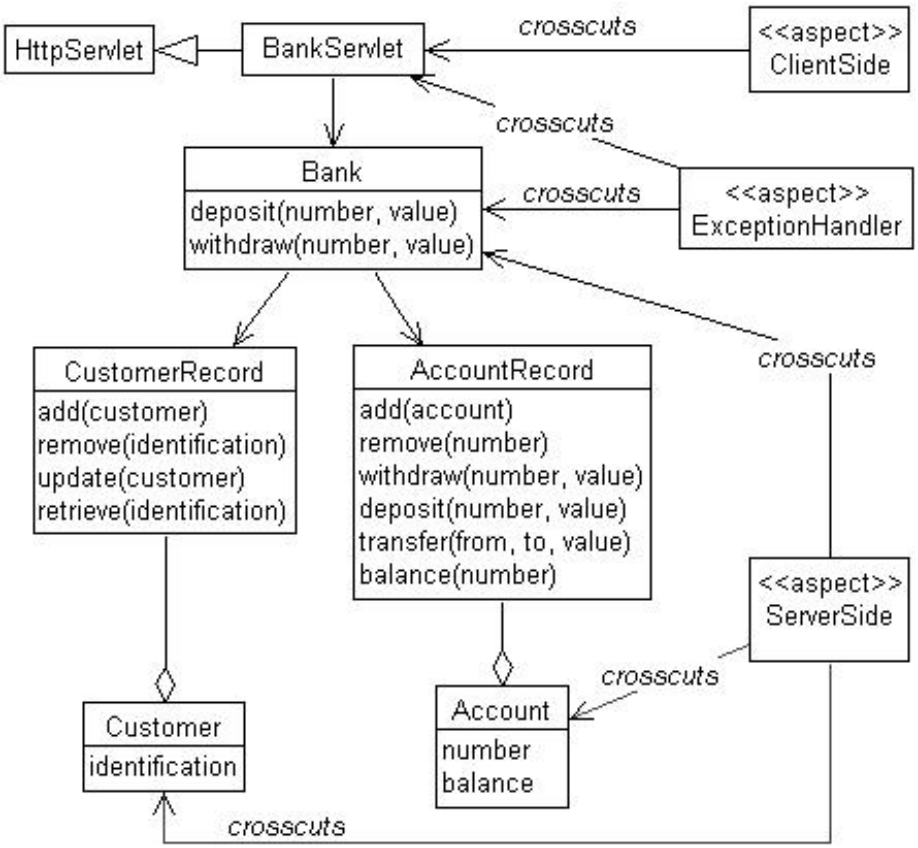


Figure 6: Class diagram of a banking application using *PaDA*.

executions, field references (get and set), exception handling, static initializations, others, and combinations of these by using the !, && or || operators. Usually, an aspect defines a *pointcut* that selects some *join points* and values at those *join points*. Then an *advice* defines the code that is executed when a *pointcut* is reached. The *advice* is who defines what code should execute *before*, *after*, or *around* the *pointcut*.

Server-side aspect

The `ServerSide` aspects should make the `Bank` instance available to remote calls. Besides being the system facade, the `Bank` class also implements the *Singleton* [8] design pattern. The server-side aspect should intercept the `Bank` initialization to make it available to be remotely accessed. The first step is defining a pointcut to identify the `Bank` object initialization, which is shown in following piece of code

```
public aspect ServerSide {
  pointcut bankInit(Bank b): execution(Bank.new(..)) && this(b);
```

where the pointcut designator `execution` join points when any constructor of `Bank` is executed, and the `this` designator join points when the currently executing object is an instance of the type of `b` (`Bank`).

This pointcut is used by the following advice

```
1:  after(Bank b): bankInit(b) {
2:    try {
3:      UnicastRemoteObject.exportObject(b);
4:      java.rmi.Naming.rebind("/BankingSystem", b);
5:    } catch (Exception rmiEx) {  ... }
6:  }
```

that adds some code (lines 2 to 5) after the pointcut, i.e., after the execution of any `Bank` constructor. The added code is responsible to make the `Bank` instance available to be remotely accessed, through the name "BankingSystem".

The server-side aspect has to define a remote interface that has all facade methods signatures adding a specific RMI API exception (`java.rmi.RemoteException`).

```
public interface IRemoteBank extends java.rmi.Remote {
  void deposit(String number, double value)
      throws AccountNotFoundException, java.rmi.RemoteException;
  ...
}
```

The aspect also has to modify the classes whose objects will be remotely transmitted over the distributed communication channel. They just have to use the Java Object Serialization mechanism, by implementing the `java.io.Serializable` interface. We use the AspectJ's introduction mechanism that can modify the static structure of programs to do it, as in the following piece of code

```
declare parents: Bank implements IRemoteBank;
declare parents: Account || Client implements java.io.Serializable;
```

Client-side aspect

The client-side aspect should define a pointcut to identify all executions of the `Bank` methods (lines 3 and 4), and advices to redirect local calls to facade's remote instance, like the one in lines 6 to 13

```
 1:  public aspect ClientSide {
 2:    private IRemoteBank remoteBank;
 3:    pointcut facadeCalls(): within(HttpServlet+) &&
 4:                            call(* Bank.*(..));
 5:
 6:    Object around(double value) throws /* ... */:
 7:        facadeCalls() && call(void deposit(double)) && args(value) {
 8:      Object response = null;
 9:      try {
10:        response = remoteBank.deposit(value);
11:      } catch (RemoteException ex) { ...  }
12:      return resposta;
13:    } ...
14:  }
```

where `remoteBank` (lines 2 and 10) references the facade remote instance whose local call will be redirected to. In this case the `around` advice executes its code instead the code identified by the pointcut `facadeCalls` and the additional join points (line 7), that identify calls to the `deposit` methods that gets a `double` as argument, which should be used as argument to the remote call (line 10).

Exception handling

The AspectJ police to handle with exceptions introduced by the aspects definition is encapsulating them in to an unchecked exception, called *soft* exception. To do it we use the `declare soft` declaration to wrap the `NewException` that gets thrown at any join point picked out by the pointcut `mightThrowNewException`

```
public aspect ExceptionHandler {
  declare soft: NewException: mightThrowNewException();
```

Therefore, this unchecked exception (`SoftException`) should be handled in the user interface class. Note that exception handling is a natural crosscutting concern, usually spread in the system units. To handle this exception we should define an `after throwing` advice that runs after the join points defined by the pointcut `facadeCalls` if it throws the `SoftException`

```
  after() throwing (SoftException ex): facadeCalls() {
    // exception handling, for example, messages to the user
  }
}
```

providing the convenient exception handling.

Variants

An extension of this pattern can define other aspects to provide additional non-functional requirements, such as fault-tolerance, caching, and object transmission on demand to increase both system robustness and efficiency. Aspects can also provide functional requirements.

Another extension can define the three aspects as a single one that crosscuts source and target components and other classes that are return types or arguments type of the target component methods.

Known Uses

This pattern was used in an experiment to implement distribution in a system that allows citizens to complain about health problems and to retrieve information about the public health system, such as the location or the specialties of a health unit. The client-side aspect was defined to the system servlets, and the server-side aspect was defined to the facade class. The system facade was not in the web server due to security and performance reasons.

Another use of *PaDA* in Web based information systems can define the client-side aspect to an applet, but we have not implemented or seen that.

Developers have been using patterns [17, 2] similar to *PaDA* to implement distribution. In particular, the pattern in the first work is similar to *PaDA*'s client-side aspect, and the pattern in the second work is similar to the *PaDA*'s server-side aspect.

In fact, we know several real software projects that implement distribution and could use this pattern. Some of these systems are the following:

- The real system for registering health system complaints.
- A system to manage clients of a telecommunication company. The system is able to register mobile telephones and manage client information and telephone services configuration. The system can be used over the Internet.
- A system for performing online exams. This system has been used to offer different kinds of exams, such as simulations based on previous university entry exams, helping students to evaluate their knowledge before the real exams.
- A complex supermarket system. A system that is responsible for the control of sales in a supermarket. This system will be used in several supermarkets and is already been used in other kinds of stores.

In addition, *PaDA* can be used as one of the basic patterns of the Progressive Implementation Method (Pim) [4]. Pim is a method for the systematic implementation of complex object-oriented applications in Java. In particular, this method supports a progressive approach for object-oriented implementation, where persistence, distribution and concurrency control are not initially considered in the implementation activities, but are gradually introduced, preserving the application's functional requirements. The *PaDA* design pattern can be applied for dealing with distribution.

See Also

- *DAP — Distributed Adapters Pattern [1].* This pattern and *PaDA* has the same objectives, however, *DAP* uses plain object-oriented programming techniques and others design patterns, which do not provide full separation of concerns. Another difference is that *DAP* does not separate the exception handling as a crosscutting concern like *PaDA* does.
- *Reflection [5].* This pattern is related to aspect-oriented programming. It provides a mechanism for changing structure and behavior of software systems dynamically. This pattern splits the application into two levels. A base level that implements the functional requirements, and a meta level that can modify the base level behavior. Comparing it with *PaDA* the base level is analog to the functional requirements, for example, implemented in Java, and the meta level is analog to the aspects, for example, implements in AspectJ.
- *Distributed Proxy Pattern [14].* This pattern and *PaDA* have similar objectives, like making the incorporation of distribution transparent. However, as the previous one, this pattern does not provide full separation of concerns.

- *Wrapper-Facade [13].* Like *PaDA* this pattern has the goal of minimizing platform-specific variation in application code. However, Wrapper-Facade encapsulates existing lower-level non-object-oriented APIs (such as operating systems mutex, sockets, and threads), whereas *PaDA* encapsulates object-oriented distribution APIs, such as RMI and CORBA. Again, this pattern does not provide full separation of concerns.

- *Broker and Trader [5].* These architectural patterns focus mostly on providing fundamental distribution issues, such as marshalling and message protocols. Therefore, they are mostly tailored to the implementation of distributed platforms, such as CORBA. *PaDA* provides a higher level of abstraction: distribution API transparency to both clients and servers.

- *Chain of Responsibility [8].* Similar to *PaDA* this patterns decouples the sender of a request from its receiver. However, it does not perform isolation of the distribution platform's API.

- *Model-View-Controller (MVC) [5]* is used in the context of interactive applications with a flexible human-computer interface. Its goal is to make changes to user interface easy and even possible at run time. *PaDA* is used in the context of distributed applications and aims at making changes to the distribution platform a simple task, not impacting in other parts of the system.

Acknowledgments

We would like to give special thanks to Jorge L. Ortega Arjona, our shepherd, for his important comments, helping us to improve our pattern. We also thanks Rossana Andrade, Jonivan Lisbôa, Marcos Quináia, and Rubens Ferreira for the suggestions made at the conference.

References

[1] Vander Alves and Paulo Borba. Distributed Adapters Pattern: A Design Pattern for Object-Oriented Distributed Applications. In *First Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.

[2] Dan Becker. Design Networked Applications in RMI Using the Adapter Design Pattern. *Java World*, May 1999.

[3] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[4] Paulo Borba, Saulo Araújo, Hednilson Bezerra, Marconi Lima, and Sérgio Soares. Progressive Implementation of Distributed Java Applications. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 40–47, Los Angeles, EUA, 17th–18th May 1999.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

[6] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path–specific customization in operating system code. *FSE*, 2001.

[7] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect–Oriented Programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] R. Hirschfeld. AspectS: AOP with Squeak. In *OOPSLA'01 Workshop on Advanced Separation of Concerns*, Tampa FL, 2001.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[11] Karl Lieberherr and Doug Orleans. Preventive program maintenance in Demeter/Java. In *International Conference on Software Engineering*, pages 604–605, Boston, MA, 1997.

[12] Harold Ossher and Peri Tarr. Hyper/J: multi–dimensional separation of concerns for Java. In *22nd International Conference on Software Engineering*, pages 734–737. ACM, 2000.

[13] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern–Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*. Wiley & Sons, 2000.

[14] Antonio Rito Silva, Francisco Rosa, and Teresa Goncalves. Distributed proxy: A design pattern for distributed object communication. In *PLoP'97*, Monticello, USA, September 1997. http://jerry.cs.uiuc.edu/~plop/plop97/Proceedings/ritosilva.pdf.

[15] Sérgio Soares and Paulo Borba. Progressive implementation with aspect–oriented programming. In Springer Verlag, editor, *The 12th Workshop for PhD Students in Object–Oriented Systems, ECOOP02*, Malaga, Spain, June 2002.

[16] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA'02, Object Oriented Programming Systems Languages and Applications*. ACM Press, November 2002. To appear.

[17] Gregg Sporar. Retrofit Existing Applications with RMI. *Java World*, January 2001.

FaPRE/OO: Uma Família de Padrões para Reengenharia Orientada a Objetos de Sistemas Legados Procedimentais

Edson Luiz Recchia
DC - Universidade Federal de São Carlos / Universidade Anhembi Morumbi
erecchia@terra.com.br

Rosângela Penteado
DC - Universidade Federal de São Carlos
rosangel@dc.ufscar.br

Resumo

Padrões de engenharia reversa e de reengenharia registram como profissionais experientes conduzem esses processos em sistemas legados. A Linguagem de Padrões de Engenharia Reversa encontrada na literatura conduz esse processo no contexto da orientação a objetos. Com sua aplicação a sistemas legados procedimentais não se consegue realizar plenamente o processo de engenharia reversa. Neste trabalho, uma Família de Padrões de Reengenharia é elaborada, para conduzir esse processo a partir de sistemas legados procedimentais para sistemas alvos orientados a objetos. Sistemas legados implementados em Clipper são utilizados para ilustrar, passo a passo, o uso dessa Família.

Abstract

Reengineering Patterns, including reverse engineering patterns, record how experienced software engineers conduct these processes in legacy systems. The Pattern Language for Reverse Engineering found in the literature conducts this process in an object-oriented context. With its application to procedural legacy systems, one cannot wholly conduct the reverse engineering process. In this paper, a Family of Patterns for Reengineering is proposed to conduct this process from procedural legacy systems to object-oriented target systems. Legacy systems implemented in the Clipper language are used to illustrate, step by step, the usage of the Family.

1. Introdução

Segundo Chikofsky et al. [5], o termo engenharia reversa teve origem na análise de hardware. Afirmam que engenharia reversa, usada com tecnologia de desenvolvimento de software, pode fornecer ganhos significativos em termos de produtividade. Definem engenharia reversa como um processo de análise de um sistema existente, que identifica seus componentes e os representa em um nível mais alto de abstração.

A engenharia reversa tem um ótimo potencial de retorno econômico e é importante organizar e disseminar essas técnicas a fim de oferecer uma documentação comprovada para problemas comuns.

Dewar et al. [7], relatam que pretendiam entender como profissionais experientes conduziam a reengenharia de sistemas legados, a fim de desenvolver técnicas e materiais para

transferir essa experiência. Em particular, queriam tratar o problema de sintetizar experiência com reengenharia de sistemas e de organizações, para ajudar engenheiros de software a adquiri-las, em paralelo.

Estamos interessados em elaborar padrões para o processo de reengenharia, especialmente para engenharia reversa orientada a objetos de sistemas legados procedimentais, particularmente no domínio de sistemas de informação, bem como, para técnicas de condução desse processo.

Nossa abordagem para resolver esse problema é uma Família de Padrões de Reengenharia, denominada FaPRE/OO, contendo um conjunto de três *clusters* de padrões para o Processo de Engenharia Reversa. A maior contribuição da FaPRE/OO é que ela conduz o engenheiro de software a gerar conjuntos de padrões para os processos de engenharia reversa e de engenharia avante especializados em uma particular linguagem de programação procedimental.

Neste trabalho mostramos como a FaPRE/OO foi elaborada a partir de sistemas legados em Clipper [15] e Cobol [3], para sistemas alvo em Delphi [15] e Java [3]. Os padrões da FaPRE/OO foram aplicados a casos concretos de elaboração de orçamentos de obras da construção civil [15], controle de material em uma mineradora [3] e controle contábil [8].

Este trabalho está organizado da seguinte forma: na Seção 2 são introduzidos os padrões para o processo de engenharia reversa, propostos pela Família de Padrões de Reengenharia (FaPRE/OO); na Seção 3 são apresentados exemplos de como se gera conjuntos de padrões a partir dessa família; na Seção 4 é apresentada uma comparação da FaPRE/OO com outros trabalhos, finalmente, na seção 5 são apresentados os comentários finais.

2. Padrões para o Processo de Engenharia Reversa da Família de Padrões de Reengenharia - FaPRE/OO

A FaPRE/OO é uma Família de Padrões de Reengenharia para gerar processos de engenharia reversa e de engenharia avante orientados a objetos, a partir de sistemas legados procedimentais. É composta de quatro *clusters,* cada um agrupando os padrões relacionados a situações similares da reengenharia, sendo três *clusters* para o processo de engenharia reversa e um para o processo de engenharia avante. A Figura 1 ilustra graficamente os *clusters* e os padrões existentes em cada um deles.

Cluster 1: Modelar os Dados do Legado: agrupa padrões que extraem informações a partir dos dados e do código fonte do sistema legado gerando o MER [15] - Modelo Entidade Relacionamento (visão procedimental dos dados) e o MASA [11] - Modelo de Análise do Sistema Atual - Diagrama de Pseudo-Classes (visão orientada a objetos dos dados). Esses padrões conduzem as ações do engenheiro de software quando se tem o primeiro contato com um sistema de software. Fazem parte desse *cluster* os seguintes padrões: Iniciar Análise dos Dados; Definir Chaves; Identificar Relacionamentos; Criar Visão OO dos Dados.

Cluster 2: Modelar a Funcionalidade do Sistema: agrupa padrões para obter a funcionalidade do sistema, criando modelos que recuperem as Regras de Negócio da Empresa, contidas no sistema legado. Esses padrões habilitam o engenheiro de software a obter um entendimento detalhado dos componentes (partes) do sistema de software, aprofundando, assim, sua compreensão sobre o sistema legado. Fazem parte desse cluster os seguintes padrões: Obter Cenários; Construir Diagramas de Use Cases; Elaborar a Descrição de Use Cases; Tratar Anomalias.

Cluster 3: Modelar o Sistema Orientado a Objetos: agrupa padrões para se obter o Diagrama de Classes e os Diagramas de Seqüência do sistema, através da interação dos

produtos obtidos pelos padrões dos *clusters* anteriores. Esses padrões habilitam o engenheiro de software a obter o MAS [11] - Modelo de Analise do Sistema, sendo o modelo orientado a objetos a servir de suporte ao processo de engenharia avante. Fazem parte desse *cluster* os seguintes padrões: Definir as Classes; Definir Atributos; Analisar Hierarquias; Definir Métodos; Construir Diagramas de Seqüência.

Cluster 4: Gerar o Sistema Orientado a Objetos: agrupa padrões que completam o processo de reengenharia do sistema, transformando o sistema legado, do paradigma procedimental para o paradigma orientado a objetos. Fazem parte desse *cluster* os seguintes padrões: Definir a Plataforma; Converter o Banco de Dados; Implementar os Métodos; Realizar Melhorias na Interface.

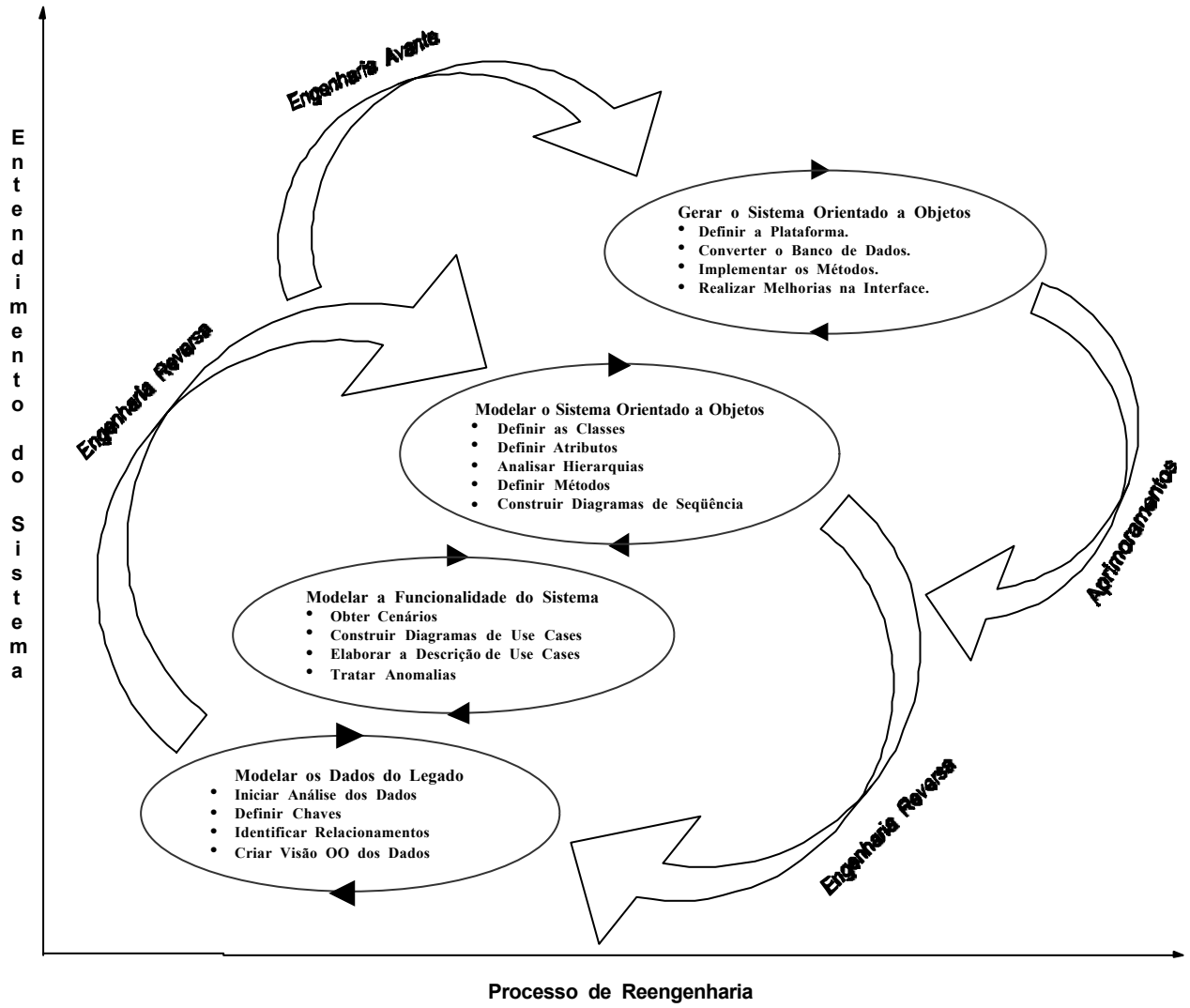


Figura 1: FaPRE/OO - Família de Padrões para Reengenharia Orientada a Objetos [15]

O leitor observará que muitos padrões terão como entrada a saída de algum padrão aplicado anteriormente, exigindo então sua aplicação seqüencial. No entanto, como se pode observar na Figura 1, o modelo é evolutivo, podendo-se, de qualquer padrão, avançar ou retroceder após a sua aplicação. A aplicação seqüencial somente será necessária durante o

primeiro ciclo. Como exemplo, pode-se citar o caso de sistemas legados com centenas de arquivos de dados, divididos em módulos funcionais. Inicia-se a engenharia reversa a partir de um módulo qualquer, aplicando-se todos padrões apresentados na Figura 1, construindo-se assim todos os modelos envolvidos. Com isso, vai-se dominando paulatinamente o sistema, podendo-se incorporar outros módulos, num processo cíclico e progressivo.

Embora a FaPRE/OO seja composta de padrões para tratar tanto a engenharia reversa como a engenharia avante, somente os padrões referentes ao processo de Engenharia Reversa são considerados neste trabalho.

Os padrões apresentados a seguir obedecem o seguinte formato proposto originalmente por Demeyer [6]: Nome, Intuito, Problema, Contexto (Influências), Solução, Avaliação, Justificativa, Usos Conhecidos, Padrões Relacionados e Produtos Obtidos. O item correspondente à **Solução** a ser adotada é apresentado na forma de passos, sempre que necessário. Os itens que tratam da **Avaliação** (*Trade-off*) e **Justificativa** não são apresentados por não serem pertinentes, uma vez que devem conter a opinião dos engenheiros de software quando da utilização desses padrões em seus processos de engenharia reversa. Finalmente, o item **Usos Conhecidos** é explicitado a seguir, por ser comum a todos os padrões.

Usos Conhecidos dos Padrões Propostos:

- O uso dos conceitos de cada padrão, sem estarem na forma de padrão, foram reconhecidos nos processos de engenharia reversa realizados em [1] [2] [3] [8] [10] [11] [14] [17].
- Procedimentos utilizados para solucionar problemas análogos, já sob a forma de padrões, foram usados em [4] [15].

2.1. *Cluster 1*: Modelar os Dados do Legado

Sistemas legados desenvolvidos de forma procedimental e implementados em linguagens como Clipper, Cobol, etc., têm, geralmente, as seguintes informações como documentação: código fonte, arquivos de dados, arquivos de índices e o próprio sistema num arquivo executável. No entanto, esses arquivos em sua maioria, representam entidades importantes do sistema, sendo possível, por meio deles, gerar um modelo de dados do sistema.

1. Nome: Iniciar Análise dos Dados

Intuito:

Iniciar a construção do MER, Modelo Entidade Relacionamento, utilizando uma tabela que relaciona todos os programas/módulos que fazem parte do sistema, com seus respectivos arquivos de dados.

Problema:

Existe uma relação de programas/módulos e de arquivos de dados na Organização, porém não se sabe como eles se relacionam.

Influências:

- Ausência de documentação, exceto o código fonte (programas) e os arquivos de dados.
- Em sistemas complexos, há grande quantidade de arquivos de programas e de dados, dificultando o seu entendimento.

Solução:

1. Construir uma tabela, denominada Programas x Arquivos, com duas colunas. A primeira coluna contém os nomes dos programas e, a segunda, o nome dos arquivos de dados a que esses programas têm acesso. O nome original desses arquivos deve ser mantido.
2. Iniciar a construção do MER, a partir da tabela construída no passo 1. Considere cada arquivo de dados do sistema legado como uma entidade do MER.

Padrões Relacionados:

Esse padrão é a entrada para o padrão Identificar Relacionamentos.

Produtos Obtidos:
Tabela Programas x Arquivos;
MER – Modelo Entidade Relacionamento Inicial.

2. Nome: Definir Chaves

Intuito:

Obter as chaves primárias e estrangeiras para cada entidade identificada pelo padrão Iniciar Análise dos Dados, a partir de arquivos de dados do sistema legado, a fim de identificar os relacionamentos entre as entidades que compõem o MER, iniciado no passo anterior.

Problema:

Em sistemas procedimentais não existe o conceito de chaves primárias e estrangeiras. Nesses sistemas as chaves podem estar em arquivos separados ou em cláusulas especiais dentro do próprio arquivo de dados.

Influências:

- Sistemas implementados em linguagens como Clipper, Cobol, etc., utilizam arquivos para persistir os dados.
- Para obter a chave primária, a partir dos arquivos de dados, é necessário analisar arquivos de índices (por exemplo, Clipper) ou o próprio arquivo (por exemplo, Cobol).
- Para obter a chave estrangeira, necessita-se fazer uma análise dos arquivos de dados e do código fonte.
- O engenheiro de software tem à sua disposição utilitários que acompanham a linguagem de implementação, os quais viabilizam a análise dos arquivos de dados e de índices.
- Têm-se todos os índices de acesso aos dados do sistema, nos quais estão identificadas as chaves de acesso aos dados.

Solução:

1. Construir a Tabela Entidades x Chaves, com duas colunas. A primeira coluna contém os nomes das entidades e, a segunda, as chaves primárias identificadas para essas entidades.
2. Utilizar o código fonte e uma ferramenta apropriada, se existir, para identificar as chaves estrangeiras das entidades, adicionando-as na Tabela Entidades x Chaves em uma nova coluna (chaves estrangeiras).

Padrões Relacionados:
Esse padrão é a entrada para o padrão Identificar Relacionamentos.

Produto Obtido:
Tabela Entidades x Chaves.

3. Nome: Identificar Relacionamentos

Intuito:
Identificar os relacionamentos entre as entidades do MER.

Problema:
Não se sabe como os arquivos de dados do sistema legado estão relacionados entre si.

Influências:

- Sistemas implementados em linguagens como Clipper, Cobol, etc., em geral não utilizam banco de dados relacional, no qual os relacionamentos entre as tabelas são explícitos.
- A partir da inspeção dos dados e da análise do código fonte é possível reconstruir o modelo de dados do sistema legado.

Solução:

1. Utilizando-se a Tabela Entidades x Chaves e trechos do código fonte que auxiliaram na elaboração dessa tabela, pode-se identificar os relacionamentos entre entidades. Sempre que o campo de um arquivo de dados, que representa a chave primária de alguma entidade, for atribuído a um campo de um outro arquivo de dados, que representa a chave

estrangeira de outra entidade (ou dela mesmo, no caso de auto-relacionamento), há um relacionamento entre essas entidades, sendo esse representado no MER.

2. Para explicitar a cardinalidade deve-se observar, nos trechos de código fonte, se há condição restritiva quanto à gravação da informação. Caso haja, a própria condição informa a cardinalidade mínima e a máxima. Caso não haja, a cardinalidade mínima é "Zero" e a máxima é "N".

Padrões Relacionados:

Esse padrão tem como entrada os seguintes padrões: Iniciar Análise dos Dados e Definir Chaves.

Produto Obtido:

MER – Modelo Entidade Relacionamento do sistema legado.

4. Nome: Criar Visão OO dos Dados

Intuito:

Criar uma visão orientada a objetos dos dados.

Problema:

Transformar um modelo de dados desenvolvido de forma procedimental em um modelo de dados orientado a objetos.

Influências:

- Reconhecer classes/objetos e seus relacionamentos a partir do código procedimental é difícil.
- O engenheiro de software tem conhecimento dos conceitos da UML, Linguagem Unificada de Modelagem, para gerar um modelo Orientado a Objetos, a partir do MER.

Solução:

A partir do MER, construir o Diagrama de Pseudo-Classes (candidatas a classes) do sistema legado, gerando, assim, o MASA, Modelo de Análise do Sistema Atual. O nome original das entidades do MER deve ser mantido.

1. Considerar cada entidade do MER como uma pseudo-classe.

2. Buscar pares de relacionamentos n-para-n, no MER, que podem ser representados como *link* de atributo no modelo orientado a objetos, de acordo com a funcionalidade do sistema legado e com os conceitos da orientação a objetos.

3. Buscar nos relacionamentos um-para-n, no MER, aqueles que podem representar o Princípio Todo-Parte (Agregação por: Referência ou Valor), considerando a funcionalidade do sistema legado e os conceitos da orientação a objetos.

4. Casos de Especialização serão tratados no padrão Analisar Hierarquias.

Padrões Relacionados:

Esse padrão é a entrada para o padrão Tratar Anomalias.

Produto Obtido:

MASA - Modelo de Analise do Sistema Atual: Diagrama de Pseudo-Classes.

2.2. *Cluster 2*: Modelar a Funcionalidade do Sistema

Os padrões desse *cluster* recuperam a funcionalidade do sistema, com o enfoque de orientação a objetos, criando modelos que representam as regras de Negócio.

5. Nome: Obter Cenários

Intuito:

Obter os Cenários do Sistema através da análise das interfaces do sistema em operação.

Problema:

Há grande variedade de interfaces nos sistemas legados. No entanto, é necessário identificar os Cenários do Sistema para obter um modelo que represente a sua funcionalidade.

Influências:

- Sistemas implementados em linguagens como Clipper, Cobol, etc., utilizam telas de menus para o acesso à funcionalidade.
- Não existe uma forma padronizada para a construção desses menus.
- Cada Cenário, em sistemas legados como Clipper, Cobol, etc., é representado por um Menu e cada opção do menu corresponde a um ou mais programas fonte.

Solução:

Construir a Tabela denominada Cenários do Sistema, com duas colunas. A primeira coluna contém as opções do menu principal e, a segunda, os submenus de cada opção do menu principal. Para essa construção, obter os cenários do sistema através da observação do sistema legado em operação.

Informações Adicionais:

- Quando sistemas legados possuem interface poluída (não possuindo menus e submenus) contendo todas as opções do sistema em uma tela, o engenheiro de software deve usar sua experiência para extrair Cenários de forma análoga aos menus.
- Para submenus que contenham outros submenus, como por exemplo um submenu Manter Clientes que possua submenu com as opções de Incluir, Alterar, Excluir, etc., o engenheiro de software deve registrar essas opções secundárias juntamente com o primeiro submenu: Manter Clientes (Incluir, Alterar, Excluir, etc.).
- Quando a interface possuir poucos menus, a Tabela Cenários do Sistema pode ser construída a partir de botões, links, ou qualquer outro meio de acesso à informação.

Padrões Relacionados:

Esse padrão é a entrada para o padrão Construir Diagramas de Use Cases.

Produto Obtido:

Tabela Cenários do Sistema.

6. Nome: Construir Diagramas de Use Cases

Intuito:

Construir todos os Diagramas de Use Cases do sistema a partir da Tabela Cenários do Sistema elaborada pelo padrão Obter Cenários.

Problema:

Documentar as informações da funcionalidade de um sistema legado procedimental, durante a engenharia reversa orientada a objetos.

Influências:

- Obter os Use Cases a partir do código procedimental é difícil.
- Os cenários obtidos a partir das opções de menu, viabilizam a construção dos respectivos diagramas de use cases.

Solução:

Considerar como Use Case cada item da segunda coluna da Tabela Cenários do Sistema. O engenheiro de software deve usar sua experiência para definir nomes para cada Use Case, bem como definir os eventos associados a cada um deles, pela observação do sistema legado em operação, quando da ativação de cada opção do Menu correspondente. Obtêm-se as especializações *uses* e *extends/includes* por observação do código fonte de cada item da segunda coluna (Conteúdo), dessa tabela, que chama ou é chamado por outro programa/módulo, que corresponde a outro item dessa mesma coluna.

Padrões Relacionados:

Esse padrão deve ser aplicado em conjunto com o padrão Entrevistar o Usuário Durante o Sistema em Operação (*Interview During Demo*) proposto por Demeyer [6] e é utilizado como

entrada para o padrão Elaborar a Descrição de Use Case.

Produto Obtido:

Diagramas de Use Cases do sistema.

7. Nome: Elaborar a Descrição de Use Cases

Intuito:

Elaborar a Descrição correspondente a cada Use Case obtido pelo padrão Construir Diagramas de Use Cases.

Problema:

É necessário registrar a lógica da funcionalidade do sistema para facilitar sua futura reengenharia.

Influências:

- Há falta de documentação do sistema que registre a lógica da funcionalidade.
- Tem-se o código fonte disponível para obter a lógica da funcionalidade.
- O engenheiro de software tem experiência de uso da linguagem de programação.

Solução:

Para cada Use Case obtido pelo padrão Construir Diagramas de Use Cases, elaborar a sua Descrição a partir do código fonte do sistema legado.

Padrões Relacionados:

Esse padrão tem como entrada o padrão Construir Diagramas de Use Cases e é utilizado como entrada para o padrão Tratar Anomalias.

Produto Obtido:

Descrição de cada Use Case do sistema.

8. Nome: Tratar Anomalias

Intuito:

Analisar as Descrições de Use Cases para tratar as anomalias, definindo, assim, os possíveis métodos das pseudo-classes (candidatas a classe) do sistema, obtidas pelo padrão Criar Visão OO dos Dados.

***Anomalia*:** Quando a Descrição do Use Case faz acesso/atualização a arquivos de dados que não pertencem à pseudo-classe a que ele se relaciona, então infere-se que esse procedimento (Descrição do Use Case) é anômalo. A anomalia pode ser do tipo:

o+ Quando o procedimento é observador de duas ou mais classes;
c+ Quando o procedimento é construtor de duas ou mais classes;
oc Quando o procedimento é observador de uma classe e construtor de outra;
o+c Quando o procedimento é observador de duas ou mais classes e construtor de outra;
oc+ Quando o procedimento é observador de uma classe e construtor de duas ou mais classes;
o+c+ Quando o procedimento é observador de duas ou mais classe e construtor de duas ou mais classe;
i Quando o procedimento é dependente da implementação e que não se refere a classe alguma.

Problema:

Em sistemas construídos de forma procedimental pode existir um número elevado de anomalias em cada procedimento (código fonte). Deve-se eliminar essas anomalias uma vez que, em sistemas orientados a objetos, os métodos estão associados a uma única classe. Então, é necessário transformar o código fonte correspondente ao procedimento anômalo (Descrição do Use Case) em métodos.

Influências:

- Quando o sistema não tem implementação orientada a objetos, a análise dos procedimentos pode ser complexa.
- Requer do engenheiro de software experiência para extrair as anomalias a partir da Descrição do Use Case.
- Têm-se as Descrições de Use Cases geradas a partir do código fonte.

Solução:

Para cada Descrição de Use Case, obtida pelo padrão Elaborar a Descrição de Use Cases, construa a Tabela Detalhes de Implementação, com seis colunas:

- A primeira coluna contém o nome do Use Case;
- A segunda coluna contém o nome de todos os programas/módulos (códigos fontes) utilizados na construção do Use Case;
- A terceira coluna contém o nome das pseudo-classes do MASA correspondentes aos arquivos de dados a que se tem acesso em cada programa/módulo;
- A quarta coluna contém a classificação quanto ao tipo de acesso ao arquivo de dados, realizado pelo procedimento;
- A quinta coluna contém os nomes dos possíveis métodos quando as anomalias forem eliminadas;
- A sexta coluna contém os nomes das pseudo-classes revisadas, a que os respectivos métodos estão associados.

1. Iniciar a construção da Tabela Detalhes de Implementação, verificando para cada Descrição de Use Case o tipo de acesso aos arquivos. Transformar o trecho fonte que consulta/altera diversos arquivos em possíveis métodos, eliminando assim o múltiplo acesso a arquivos. A parte do código que somente consulta um arquivo é retirada desse, e uma chamada a esse método é feita para que a funcionalidade do sistema seja mantida.

2. Continuar a construção da Tabela Detalhes de Implementação, renomeando as pseudo-classes que não representam completamente a informação dentro do contexto do sistema. Por exemplo, uma pseudo-classe com o nome ITPedido poderá ser renomeada para ItensPedido.

3. Acrescentar na Tabela Detalhes de Implementação, os nomes de entidades do MER (arquivo de dados) existentes no código fonte que foram usados apenas para atender a necessidades de implementação. Esses arquivos são temporários e não fazem parte da funcionalidade do sistema. Portanto, toda pseudo-classe que é criada por algum procedimento classificado com a anomalia **i** deve ser desconsiderada.

4. Acrescentar na coluna pseudo-classes revisadas, da Tabela Detalhes de Implementação, as pseudo-classes correspondentes a trechos do código fonte usados para implementar informações adicionais em memória. Por exemplo, tabelas de descontos em função da quantidade comprada, que foram implementadas em trechos de código ao invés de estarem persistidas em arquivos de dados. Em sistemas procedimentais esses artifícios eram comuns para melhorar o desempenho do sistema. Como essas pseudo-classes são geradas diretamente do código fonte, elas não aparecem como entidades do MER e, conseqüentemente, como pseudo-classes do MASA, pelo fato de não existir um arquivo de dados correspondente.

Informações Adicionais:

Na abordagem Fusion/RE [11] [12] [13] são analisados procedimentos verificando-se qual estrutura de dados está associada a cada um. Além disso, analisa-se a forma de associação entre classes e procedimentos. Para isso, Fusion/RE adota a convenção denominada de anomalias. No padrão Presumir Prováveis Objetos (*Speculate about Domain Objects*), Demeyer [6] utiliza adaptações no modelo de classes tais como: renomeando, remodelando e estendendo para resolver inconsistências.

Padrões Relacionados:

Esse padrão tem como entrada o padrão Elaborar a Descrição de Use Cases e é utilizado como entrada para os padrões do *Cluster* Modelar o Sistema Orientado a Objetos.

Produto Obtido:

Tabela Detalhes de Implementação.

2.3. *Cluster 3*: Modelar o Sistema Orientado a Objetos

Os padrões desse *cluster* são aplicados para concluir o processo de engenharia reversa do sistema legado. Nesse momento, a documentação levantada da engenharia reversa é consultada para fins de consolidação das informações.

9. Nome: Definir as Classes

Intuito:

Iniciar a construção do Diagrama de Classes do sistema.

Problema:

Existem várias informações levantadas pelo processo de engenharia reversa: MER, MASA, Diagramas de Use Cases, Descrições de Use Cases e Tabela Detalhes de Implementação. No entanto, elas precisam ainda ser consolidadas e apresentadas na visão de orientação a objetos.

Influências:

- As Pseudo-Classes Revisadas, da Tabela Detalhes de Implementação, são fortes candidatas às classes do modelo orientado a objetos, pois já passaram por diversos refinamentos durante a aplicação do padrão Tratar Anomalias.
- O mesmo ocorre em relação aos relacionamentos entre as pseudo-classes.

Solução:

1. Transformar cada pseudo-classe revisada, da Tabela Detalhes de Implementação, em uma classe no Diagrama de Classes.
2. Gerar os relacionamentos no diagrama de classes a partir dos relacionamentos existentes do MASA. Além disso, para as pseudo-classes que foram geradas diretamente do código fonte, criar relacionamentos que representam o Princípio Todo-Parte (Agregação por: Referência ou Valor), bem como *link* de atributo, conforme os conceitos da orientação a objetos, discutidos pelo padrão Criar Visão OO dos Dados.

Padrões Relacionados:

Esse padrão tem como entrada o padrão Tratar Anomalias e serve como entrada para o padrão Definir Atributos.

Produto Obtido:

Diagrama de Classes (Inicial) do sistema: classes definidas.

10. Nome: Definir Atributos

Intuito:

Definir os atributos das classes pertencentes ao Diagrama de Classes em construção.

Problema:

Identificar os atributos das classes a partir dos arquivos de dados do sistema legado.

Influências:

- Transformar os campos dos arquivos de dados em atributos das classes pode requerer uma análise complexa das estruturas de dados.

- Existem ferramentas que auxiliam a análise das estruturas dos dados do legado.

Solução:

1. Os campos dos arquivos de dados, representados como pseudo-classes no MASA são, na sua maioria, transformados em atributos das classes correspondentes. Dessa forma, representar esses atributos das classes, no Diagrama de Classes.
2. Os casos em que não há correspondência direta deverão ser tratados em paralelo avaliando-se as questões de hierarquias abordadas pelo padrão Analisar Hierarquias.

Padrões Relacionados:

Esse padrão tem como entrada o padrão Definir as Classes e serve como entrada para o padrão Definir Métodos. Pode ser aplicado, se necessário, em paralelo ao padrão Analisar Hierarquias.

Produto Obtido:

Diagrama de Classes (continuação) do Sistema: classes e atributos definidos.

11. Nome: Analisar Hierarquias

Intuito:

Fornecer um mecanismo para descobrir possíveis papéis (mais de uma função) e hierarquias de herança que possam estar contidas nos arquivos de dados do sistema legado.

Problema:

Encontrar papéis e hierarquias de herança que possam estar persistidos como campos em vários arquivos de dados do sistema legado.

Influências:

- Hierarquias de Herança podem estar contidas nos arquivos de dados das seguintes formas:
  a) Por meio de conjuntos de campos opcionais (campos que assumem valores dependendo de um contexto. Por exemplo, um arquivo de dados de nome Cliente possui os campos CPF e CGC, sendo que para cada cliente, somente um desses campos assumirá valor, dependendo do tipo do cliente – pessoa física ou jurídica);
  b) Por meio de conjuntos de campos semelhantes (mesmos campos em diferentes arquivos de dados).
- Hierarquias de Herança podem também estar contidas em relacionamentos do tipo um-para-um, ligando várias pseudo-classes do MASA.
- Papéis podem estar contidos nos arquivos de dados, por exemplo, por meio da cláusula *Redefines* do Cobol.
- O engenheiro de software possui experiência para definir super-classes e transformar super-classes em várias subclasses, a partir de pseudo-classes do MASA.

Solução:

1. Encontrar, no sistema legado, os arquivos de dados com campos opcionais. Isso indica uma situação em que uma hierarquia de classe completa está representada em um único arquivo de dados. Nesse caso, transformar esse arquivo de dados numa super-classe e gerar tantas classes quantas forem necessárias para representar esse conjunto de informações.
2. Encontrar, no sistema legado, os arquivos de dados com campos semelhantes. Isso indica que a hierarquia de classe está distribuída entre vários arquivos de dados. Nesse caso, definir uma super-classe movendo os campos comuns para ela e, os não comuns, continuam pertencendo às respectivas subclasses.
3. Encontrar, no MASA, as pseudo-classes nas quais a chave primária também serve como chave estrangeira de outra pseudo-classe. Isso pode representar um relacionamento um-

para-um, indicando, talvez, um relacionamento de herança. Nesse caso, deve-se analisar a funcionalidade do sistema legado e representar esse conjunto de pseudo-classes como uma Hierarquia de Herança entre Classes, quando a essa característica for validada.

4. Encontrar, no sistema legado, os arquivos de dados com campos possuindo um ou mais papéis. Nesse caso, para cada papel gerar uma nova classe no Diagrama de Classes.

Padrões Relacionados:

Esse padrão tem como entrada os padrões Definir Classes e Criar Visão OO dos Dados. Pode ser aplicado, se necessário, em paralelo ao padrão Definir Atributos.

Produto Obtido:

Diagrama de Classes (continuação) do Sistema: classes e atributos definidos.

12. Nome: Definir Métodos

Intuito:

Definir os métodos das classes pertencentes ao Diagrama de Classes em construção.

Problema:

Identificar métodos de cada classe de forma a implementar toda a funcionalidade requerida por tal classe.

Influências:

- Definir cada um dos métodos a partir dos procedimentos anômalos existentes no código fonte legado é difícil.
- Existe indício dos prováveis métodos na Tabela Detalhes de Implementação.

Solução:

Representar os métodos que estão relacionados na coluna Possíveis Métodos, da Tabela Detalhes de Implementação, nas classes respectivas às pseudo-classes, especificadas na coluna Pseudo-Classes Revisadas, da mesma tabela. Assim, obtêm-se os métodos das classes no Diagrama de Classes em construção por meio da Descrição do Use Case correspondente.

Informações Adicionais:

O engenheiro de software deve observar trechos do código fonte que possam ser caracterizados como Polimorfismo, gerando tantos métodos quantos forem necessários, consolidando, assim, o diagrama de classes do sistema orientado a objetos.

Padrões Relacionados:

Esse padrão tem como entrada o padrão Tratar Anomalias.

Produto Obtido:

Diagrama de Classes do Sistema.

13. Nome: Construir Diagramas de Seqüência

Intuito:

Documentar as Regras de Negócio do sistema legado para facilitar a sua futura reengenharia.

Problema:

Necessidade de explicitar as Regras de Negócio, muitas vezes embutidas no código fonte.

Influências:

- Representar a interação dos métodos obtidos no padrão Definir Métodos, pode ser complexo.
- Métodos já definidos a partir das Descrições de Use Cases, viabilizam o problema.

Solução:

Construir todos os Diagramas de Seqüência, a partir de cada Descrição de Use Case, obtida pelo padrão Elaborar Descrição de Use Cases. Considere, para a sua construção, o

processo de eliminação de anomalias efetuado pelo padrão Tratar Anomalias, quando foram gerados tantos métodos quantos foram necessários, para manter a mesma funcionalidade do sistema legado. Agora, no Diagrama de Seqüência, é apresentada graficamente a interação desses métodos.

Padrões Relacionados:

Esse padrão tem como entrada os padrões Elaborar Descrição de Use Cases, Definir Métodos e Tratar Anomalias.

Produtos Obtidos:

Diagramas de Seqüência do Sistema.

3. Exemplos de Uso

Para ilustrar como se geram conjuntos de padrões a partir da FaPRE/OO, foram especializados, para a linguagem Clipper, todos os padrões do processo de engenharia reversa dessa família. Dessa forma, gerou-se um conjunto completo de todos padrões dos três *clusters* iniciais, derivando, assim, um processo de engenharia reversa orientada a objetos a partir da linguagem procedimental Clipper.

3.1. *Cluster 1*: Modelar os Dados do Legado

1. Nome: Iniciar Análise dos Dados

Intuito:

Iniciar a construção do MER, Modelo Entidade Relacionamento, utilizando uma tabela que relaciona todos os programas que fazem parte do sistema, com seus respectivos arquivos de dados.

Solução:

1. Construir a Tabela Programas x Arquivos.

A Tabela 1 ilustra esse passo, sendo que a coluna 1 mostra os programas e a coluna 2 os arquivos de dados, tratados em cada programa.

Tabela 1: Tabela Programas x Arquivos

Programa (.prg)	**Arquivos (.dbf)**
ManterCliente.prg	Cliente.dbf CLI_Sort.dbf Pais.dbf PS_Sort.dbf
RegistrarPedido.prg	Pedido.dbf Cliente.dbf CLI_Sort.dbf Produto.dbf PRD_Sort.dbf ItPedido.dbf
•••	•••

2. Iniciar a construção do MER, a partir da tabela construída no passo 1. Considere cada arquivo de dados do sistema legado (.dbf) como uma entidade do MER, Figura 2.

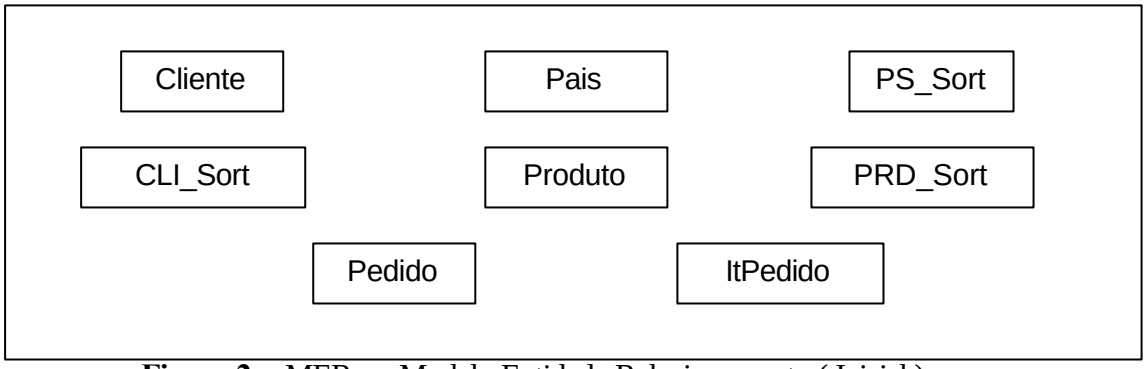


Figura 2: MER – Modelo Entidade Relacionamento (Inicial)

2. Nome: Definir Chaves

Intuito:

Obter as chaves primárias e estrangeiras para cada entidade identificada pelo padrão Iniciar Análise dos Dados, a partir de arquivos de dados do sistema legado (.dbf), a fim de identificar os relacionamentos entre as entidades que compõem o MER, iniciado no passo 1.

Solução:

1. Construir a Tabela Entidades x Chaves, inicialmente com duas colunas.

Por meio do comando USE, da linguagem Clipper, é possível obter todos os arquivos de índices (.ntx) para cada arquivo de dados (.dbf). Um arquivo .dbf pode ter até 15 arquivos de índices.

Por exemplo: **USE Cliente INDEX CliCod, CliNom** sendo CliCod.ntx e CliNom.ntx arquivos de índices do arquivo de dados Cliente.dbf.

Identifica-se a chave primária para cada entidade analisando o respectivo arquivo de dados .dbf com a ferramenta DBU [15], que acompanha a linguagem Clipper, observando-se a linha que contém a palavra **`Key`**. A Figura 3 ilustra as telas da ferramenta DBU para o comando USE descrito anteriormente.

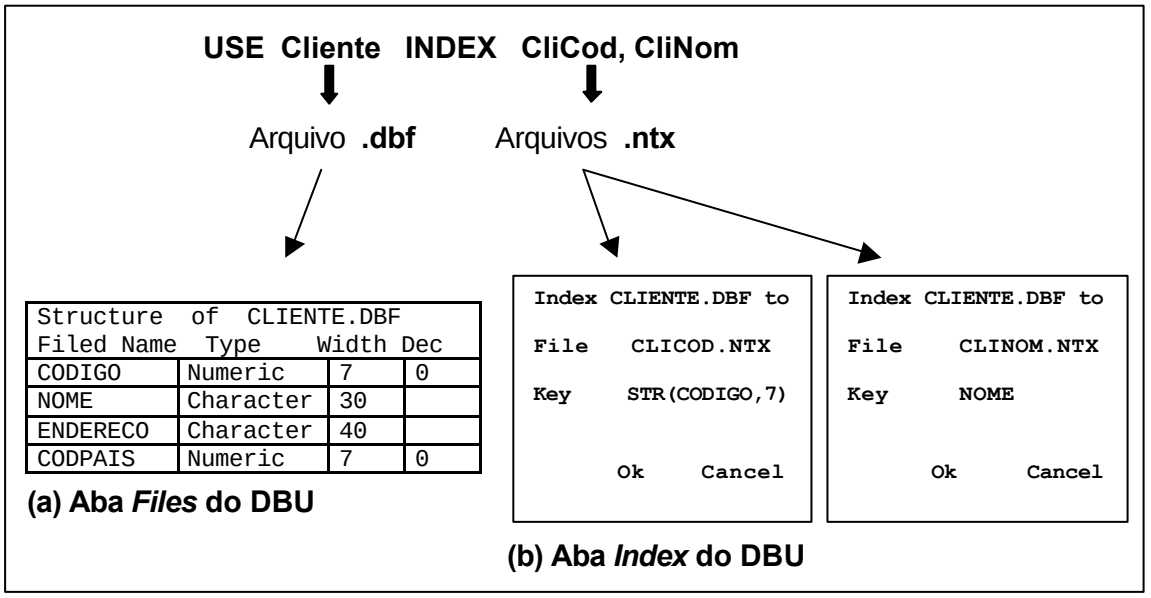


Figura 3: Identificação da Chave Primária utilizando o DBU

Analisando a aba *index*, Figura 3(b), conclui-se que o campo CODIGO, do arquivo de dados CLIENTE.DBF, é a chave primária da entidade Cliente, uma vez que o campo NOME não identifica univocamente um registro desse arquivo de dados.
Repete-se esse passo para cada comando USE encontrado no código fonte. Um exemplo do resultado obtido nesse passo é a Tabela 2.

Tabela 2: Tabela Entidades x Chaves (Inicial)

Entidades	Chave Primária
Cliente	CODIGO
Produto	CODIGO
Pais	CODIGO
•••	•••

2. Utilizar o código fonte e a ferramenta DBU para identificar as chaves estrangeiras das entidades, adicionando-as na Tabela Entidades x Chaves em uma nova coluna (chave estrangeira). A Figura 4 ilustra a construção da Tabela 3.

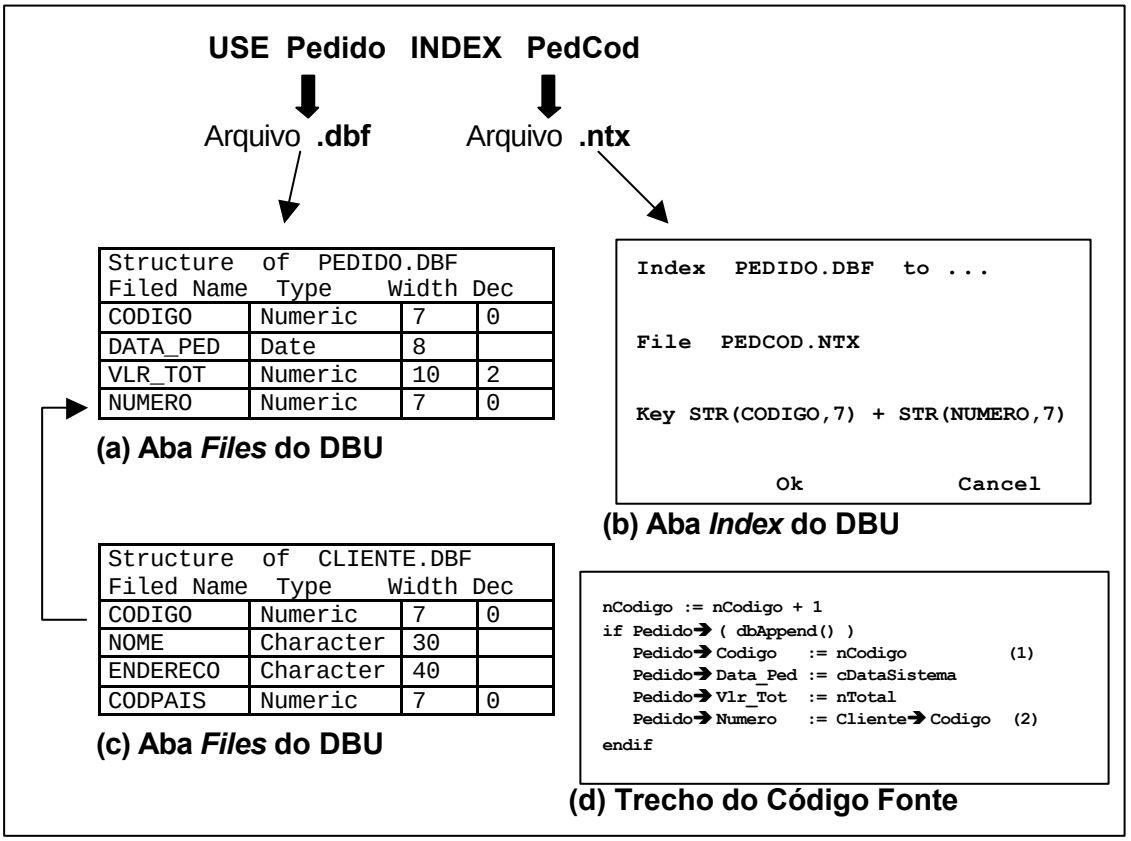


Figura 4: Identificação da Chave Estrangeira utilizando o DBU

Analisando-se:

1) O arquivo CLIENTE.DBF, já com a chave primária definida para a entidade Cliente, o campo CODIGO;

2) O arquivo PEDIDO.DBF, cuja chave primária para a entidade Pedido é a concatenação dos campos CODIGO e NUMERO;

3) O trecho de código exibido na Figura 4(d).

Tem-se que NUMERO é chave estrangeira da entidade Pedido, definido pelo comando (2) da Figura 4(d). O campo CODIGO, que participa da chave primária, não é, também, uma chave estrangeira porque esse campo é do arquivo de dados PEDIDO.DBF, sendo atualizado pelo comando (1) da Figura 4(d).
Dessa forma, completa-se a Tabela Entidades x Chaves, contendo três colunas, sendo que a última contém a chave estrangeira, Tabela 3.

Tabela 3: Tabela Entidades x Chaves

Entidades	Chave Primária	Chave Estrangeira
Cliente	CODIGO	—
Produto	CODIGO	—
Pais	CODIGO	—
Pedido	CODIGO + NUMERO	NUMERO
ItPedido	CODPEDIDO + CODPRODUTO	CODPEDIDO, CODPRODUTO
•••	•••	•••

3. Nome: Identificar Relacionamentos

Intuito:

Identificar os relacionamentos entre as entidades no MER.

Solução:

Utilizando-se a Tabela Entidades x Chaves e trechos do código fonte que auxiliaram na elaboração dessa tabela, pode-se identificar os relacionamentos entre entidades. Sempre que o campo de um arquivo de dados, que representa a chave primária de alguma entidade, for atribuído a um campo de um outro arquivo de dados, que representa a chave estrangeira de outra entidade, há um relacionamento entre essas entidades, sendo esse representado no MER. Para explicitar a cardinalidade deve-se observar, nos trechos de código fonte, se há condição restritiva quanto à gravação da informação.

Por exemplo, considere a chave estrangeira NUMERO, Tabela 3, e o trecho de código da Figura 4(d). Note que o comando (2) `Pedido➔Numero := Cliente➔Codigo` atribui a chave primária da entidade Cliente à chave estrangeira da entidade Pedido.

Quando se encontra a chave estrangeira de uma entidade, deve-se gerar um relacionamento entre entidades do MER. Por exemplo, o comando (2) da Figura 4(d) gera o relacionamento **faz**, denotado por (a) na Figura 5, que relaciona as entidades Cliente e Pedido. Dessa forma, determinam-se todos os relacionamentos entre entidades, gerando o Modelo Entidade Relacionamento (MER) correspondente ao sistema legado.

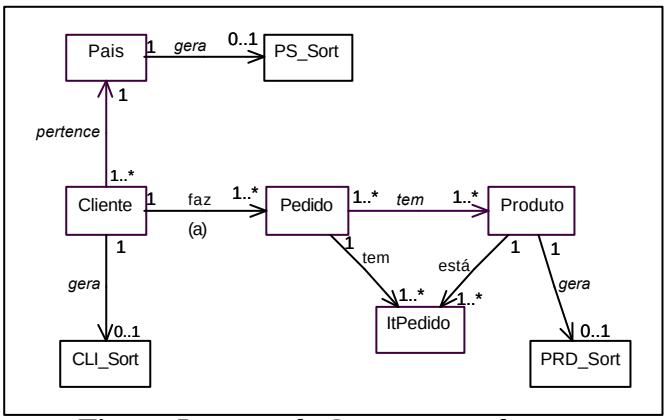


Figura 5: MER do Sistema Legado

4. Nome: Criar Visão OO dos Dados

Intuito:

Criar uma visão orientada a objetos dos dados.

Solução:

A partir do MER, construir o Diagrama de Pseudo-Classes (candidatas a classes) do sistema legado, gerando, assim, o MASA, Modelo de Análise do Sistema Atual. O nome original das entidades do MER deve ser mantido. A Figura 6 apresenta o MASA para o MER do Sistema Legado (Figura 5).

1. Considerar cada entidade do MER como uma pseudo-classe.
2. Buscar pares de relacionamentos n-para-n, no MER, que podem ser representados como *link* de atributo no modelo orientado a objetos, de acordo com a funcionalidade do sistema legado e com os conceitos da orientação a objetos.
3. Buscar nos relacionamentos um-para-n, no MER, aqueles que podem representar o Princípio Todo-Parte (Agregação por: Referência ou Valor), considerando a funcionalidade do sistema legado e os conceitos da orientação a objetos.

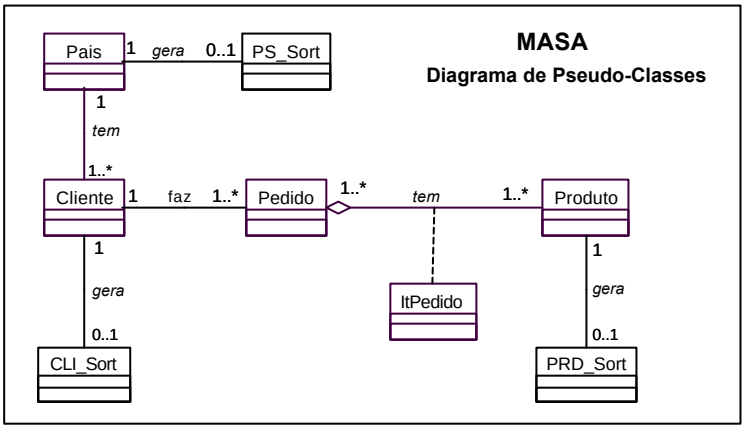


Figura 6: MASA do Sistema Legado

3.2. *Cluster 2*: Modelar a Funcionalidade do Sistema

5. Nome: Obter Cenários

Intuito:

Obter os Cenários do Sistema através da análise das interfaces do sistema em operação.

Solução:

Construir a Tabela Cenários do Sistema, com duas colunas. A primeira coluna contém as opções do menu principal e, a segunda, os sub-menus de cada opção do menu principal. Para essa construção, obter os cenários do sistema através da observação do sistema legado em operação.

Considere, como exemplo, a execução de um sistema hipotético de Gestão Administrativa, denominado GEST_ADM, implementado na linguagem Clipper. A Figura 7 exibe a interface inicial desse sistema, estando o Prompt do DOS posicionado em: menu de ***VENDAS***, opção ***1> Pedidos***. A Tabela 4 mostra os cenários obtidos dessa interface.

Informações Adicionais:

Quando sistemas legados possuem interface poluída (não possuindo menus e submenus) contendo todas as opções do sistema em uma tela, o engenheiro de software deve usar de sua experiência para extrair Cenários da forma análoga aos menus, como na do caso da Figura 8.

Como a interface da Figura 8 possui as mesmas opções mostradas na Figura 7, a Tabela de Cenários do Sistema é a exibida na Tabela 4. Não há garantia que diferentes engenheiros de software gerem a mesma Tabela de Cenários do Sistema.

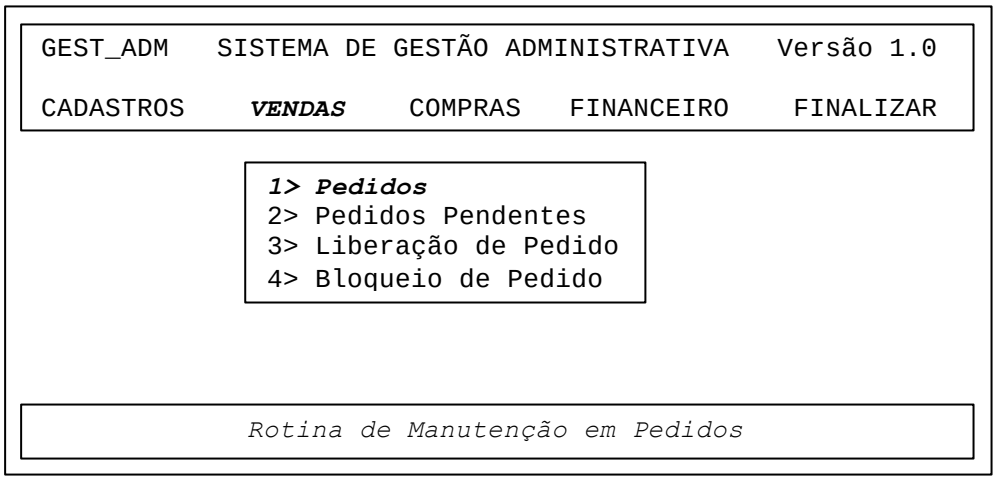


Figura 7: Menu de Abertura do Sistema de Gestão Administrativa (GEST_ADM)

Tabela 4: Tabela Cenários do Sistema

Cenários (opções do menu principal)	**Conteúdo (opções do sub-menus de cada opção principal)**
Cadastros	Clientes Produtos Paises
Vendas	Pedidos Pedidos Pendentes Liberação de Pedido Bloqueio de Pedido
Compras	Fornecedores Condições de Pagamento Ordem de Compra
Financeiro	Lançamento de Pagamentos Pagamentos por Vencimento
Finalizar	Sair do Sistema

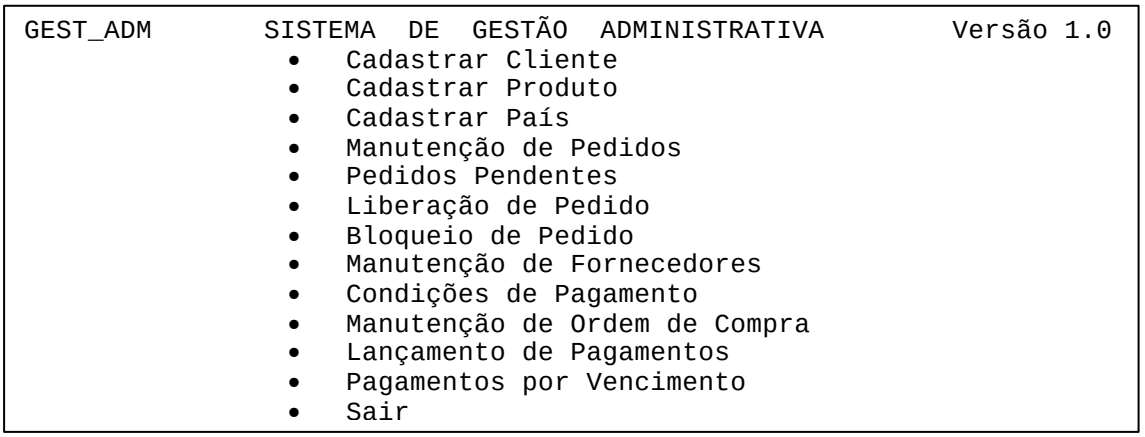


Figura 8: Menu de Abertura do Sistema GEST_ADM (Interface Poluída)

6. Nome: Construir Diagramas de Use Cases

Intuito:

Construir todos os Diagramas de Use Cases do sistema a partir da Tabela Cenários do Sistema elaborada no padrão Obter Cenários.

Solução:

Considerar como Use Case cada item da coluna "Conteúdo" da Tabela Cenários do Sistema. Observar que cada item da coluna "Cenários" possui um Conteúdo com várias opções, conforme mostra a Tabela 4. Cada opção do Conteúdo de um Cenário passa a corresponder a um Use Case no Diagrama em construção.

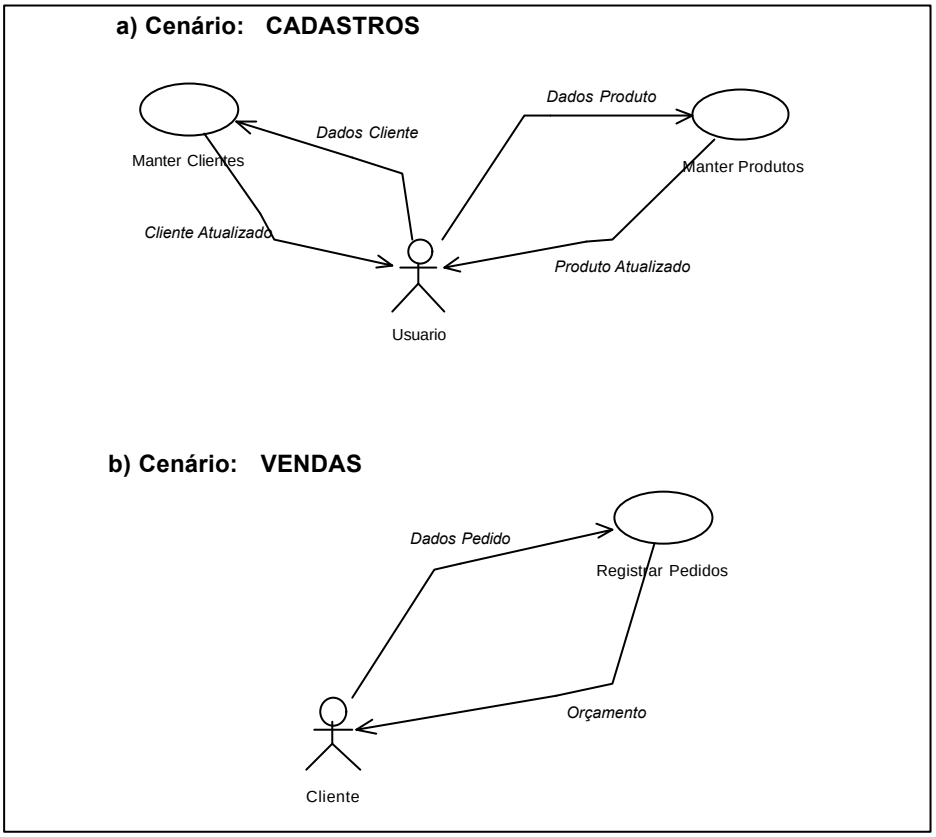


Figura 9: Diagrama de Use Cases do Sistema GEST_ADM

A Figura 9 exibe o Diagrama de Use Cases do Sistema de Gestão Administrativa GEST_ADM para os Cenários CADASTROS e VENDAS.

7. Nome: Elaborar a Descrição de Use Cases

Intuito:

Elaborar a Descrição correspondente a cada Use Case obtido no padrão Construir Diagramas de Use Cases.

Solução:

Para cada Use Case obtido no padrão Construir Diagramas de Use Cases, elaborar a sua Descrição a partir de trechos do código fonte correspondente à opção do Menu ativada, quando da construção do respectivo Use Case. A Figura 10 mostra a Descrição do Use Case Registrar Pedidos, do Cenário VENDAS, apresentado na Figura 9(b).

1 - Cliente solicita um Pedido
2 - Sistema solicita a Identificação do Cliente
3 - Cliente fornece a Identificação (seu Nome)
4 - Sistema usa parte da Identificação para gerar, a partir do arquivo Cliente.dbf , uma relação de todos Clientes semelhantes, gravando esses dados no arquivo CLI_Sort.dbf
5 - Sistema seleciona o Cliente associado com a Identificação
6 - Sistema mostra os dados do Cliente
7 - Cliente confirma seus dados
8 - Sistema grava o cabeçalho do Pedido no arquivo Pedido.dbf
9 - Para cada Item do Pedido
9.1 - Cliente informa o Produto e a Quantidade
9.2 - Sistema usa parte da Descrição do Produto para gerar, a partir do arquivo Produto.dbf, uma relação de todos Produtos semelhantes, gravando esses dados no arquivo PRD_Sort.dbf
9.3 - Sistema seleciona o Produto informado pelo Cliente
9.4 - Cliente confirma o Item do Pedido
9.5 - Sistema grava o Item do Pedido no arquivo ITPedido.dbf, aplicando o seguinte desconto:
- Para Quantidade Comprada < 10, Conceder Desconto de 3 %
- Para Quantidade Comprada entre 10 e 20, Conceder Desconto de 7 %
- Para Quantidade Comprada > 20, Conceder Desconto de 10 %

Se o País do Cliente for diferente do Brasil, atualizar o Desconto, de acordo:
- ✓ Para o País = 1, Brasil, não tem índice de alteração do Desconto;
- ✓ Para o País = 2, Argentina, multiplicar o Desconto pelo índice 0.95;
- ✓ Para o País = 3, Uruguai, multiplicar o Desconto pelo índice 0.92;
- • • •
- ✓ Para o País = 15, México, multiplicar o Desconto pelo índice 0.79;
- • • •

10 - Sistema solicita confirmação do Pedido
11 - Cliente confirma o Pedido
12 - Sistema emite Cópia do Pedido para ser enviada ao Cliente

Figura 10: Descrição do Use Case Registrar Pedidos

8. Nome: Tratar Anomalias

Intuito:

Analisar as Descrições de Use Cases para tratar as anomalias, definindo, assim, os possíveis métodos das pseudo-classes (candidatas a classe) do sistema, obtidas pelo padrão Criar Visão OO dos Dados.

Solução:

Para cada Descrição de Use Case, obtida pelo padrão Elaborar a Descrição de Use Cases, construir a Tabela Detalhes de Implementação.

1. Iniciar a construção da Tabela Detalhes de Implementação, verificando para cada Descrição de Use Case o tipo de acesso aos arquivos. Transformar o trecho fonte que consulta/altera diversos arquivos em possíveis métodos, eliminando assim o múltiplo acesso a arquivos. A parte do código que somente consulta um arquivo é retirada desse, e uma chamada a esse método é feita para que a funcionalidade do sistema seja mantida.

A Tabela 5 exemplifica o caso de anomalia **o+c+** para a Descrição do Use Case Registrar Pedidos, apresentada na Figura 10.

2. Continuar a construção da Tabela Detalhes de Implementação, renomeando as pseudo-classes que não representam completamente a informação dentro do contexto do sistema. Por exemplo, a pseudo-classe ITPedido (correspondente ao arquivo de dados ITPedido.dbf e a entidade ITPedido) poderá ser renomeada para ItemPedido, conforme pode ser visto na Tabela 5.

Tabela 5: Tabela Detalhes de Implementação

Use Case	Códigos Fontes Correspondentes	Pseudo Classes (MASA)	Tipo da Anomalia	Possíveis Métodos	Pseudo Classes Revisadas
Registrar Pedidos	RegPed.prg (o+c+)	Pedido	c	Pedido()	Pedido
		Cliente	o	SelecionarCliente()	Cliente
		Produto	o	SelecionarProduto()	Produto
		ITPedido	c	ItemPedido()	ItemPedido
•••	•••	•••	•••	•••	•••

3. Acrescentar na Tabela Detalhes de Implementação, os nomes de entidades do MER (arquivo de dados) existentes no código fonte que foram usados apenas para atender a necessidades de implementação, como é o caso das entidades CLI_Sort e PRD_Sort da Tabela 6. Esses arquivos são temporários e não fazem parte da funcionalidade do sistema. Portanto, toda pseudo-classe que é criada por algum procedimento classificado com a anomalia **i** deve ser desconsiderada.
4. Acrescentar na coluna pseudo-classes revisadas, da Tabela Detalhes de Implementação, as pseudo-classes correspondentes a trechos do código fonte usados para implementar informações adicionais em memória. Por exemplo, tabelas de descontos em função da quantidade comprada, que foram implementadas em trechos de código ao invés de estarem persistidas em arquivos de dados. Em sistemas procedimentais esses artifícios eram comuns para melhorar o desempenho do sistema. Como essas pseudo-classes são geradas diretamente do código fonte, elas não aparecem como entidades do MER e, conseqüentemente, como pseudo-classes do MASA, pelo fato de não existir um arquivo de dados (.dbf) correspondente.

Tabela 6: Tabela Detalhes de Implementação

Use Case	Códigos Fontes Correspondentes	Pseudo Classes (MASA)	Tipo da Anomalia	Possíveis Métodos	Pseudo Classes Revisadas
Registrar Pedidos	RegPed.prg (o+c+)	Pedido	c	Pedido()	Pedido
		Cliente	o	SelecionarCliente()	Cliente
		Produto	o	SelecionarProduto()	Produto
		ITPedido	c	ItemPedido()	ItemPedido
		CLI_Sort	i	—	—
		PRD_Sort	i	—	—
		—	—	AplicarDesconto()	Desconto
•••	•••	•••	•••	•••	•••

Por exemplo, na Descrição do Use Case Registrar Pedidos, Figura 10, o Passo 9.5 é uma funcionalidade que foi implementada em código fonte, pois não existe no sistema legado um arquivo de dados (.dbf) contendo os descontos concedidos. Portanto, será criada a pseudo-classe Desconto, representando essa funcionalidade, conforme ilustrado na Tabela 6.

3.3. *Cluster 3*: Modelar o Sistema Orientado a Objetos

9. Nome: Obter as Classes

Intuito:

Iniciar a construção do Diagrama de Classes do sistema.

Solução:

1. Transformar cada pseudo-classe, da coluna Pseudo-Classe Revisada, da Tabela Detalhes de Implementação, em uma classe no Diagrama de Classes, conforme mostra a Figura 11.
2. Gerar os relacionamentos no diagrama de classes a partir dos relacionamentos existentes do MASA. Além disso, para as pseudo-classes que foram geradas diretamente do código fonte, como é o caso da pseudo-classe Desconto, criar relacionamentos que representam o Princípio Todo-Parte (Agregação por: Referência ou Valor), bem como *link* de atributo, conforme os conceitos da orientação a objetos, discutidos pelo padrão Criar Visão OO dos Dados. A Figura 12 ilustra os respectivos relacionamentos.

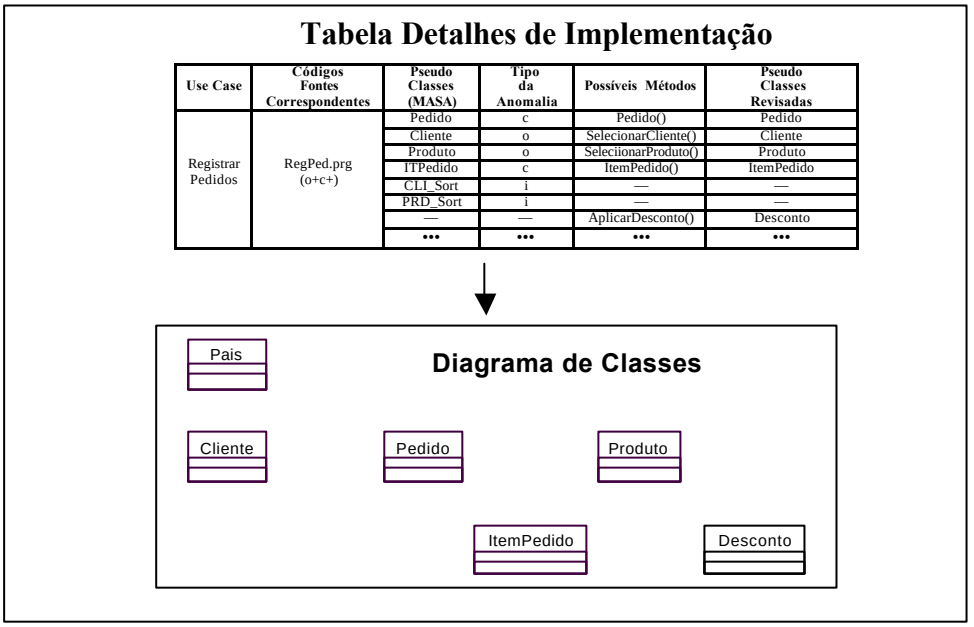


Figura 11: Diagrama de Classes (Construção das Classes)

10. **Nome:** Definir Atributos
    Intuito:
    Definir os atributos das classes pertencentes ao Diagrama de Classes em construção.
    Solução:

1. Os campos dos arquivos de dados (.dbf), representados como pseudo-classes no MASA são, na sua maioria, transformados em atributos das classes correspondentes. Dessa forma, representar esses atributos das classes, no Diagrama de Classes. A Figura 13 ilustra o Diagrama de Classes com os atributos.
2. Os casos em que não há correspondência direta deverão ser tratados em paralelo avaliando-se as questões de hierarquias abordadas pelo padrão Analisar Hierarquias.

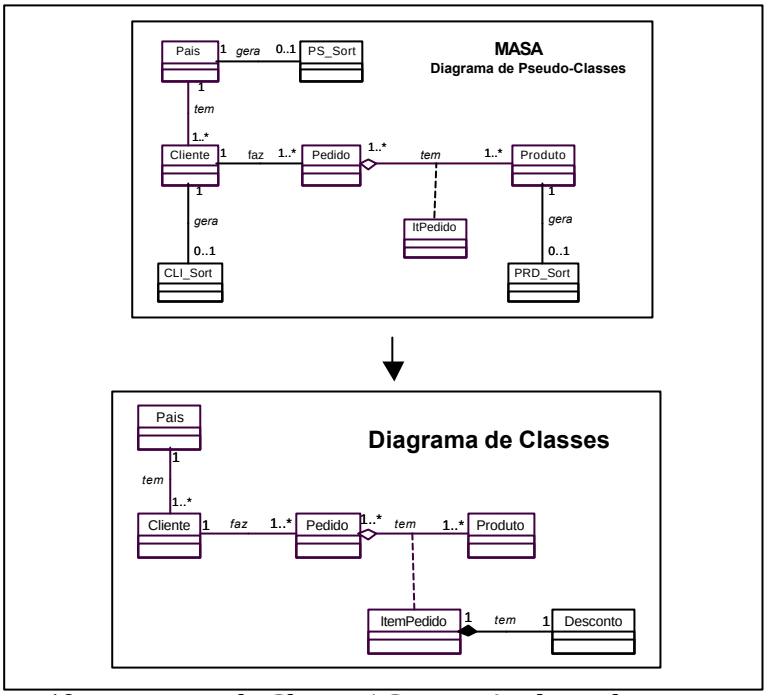


Figura 12: Diagrama de Classes (Construção dos Relacionamentos)

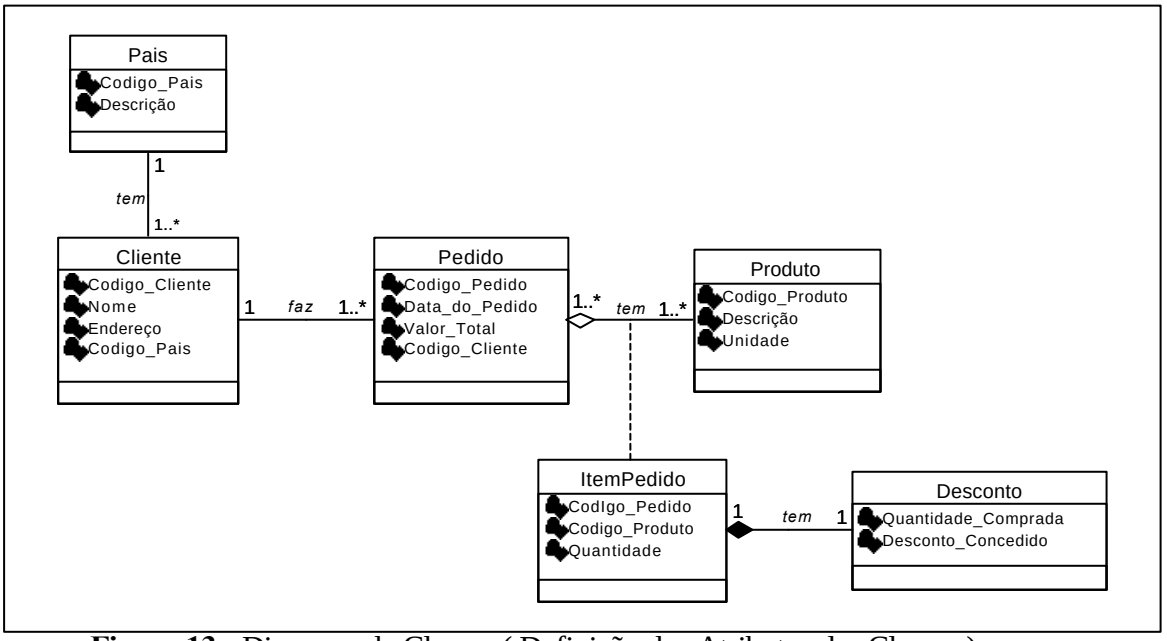


Figura 13: Diagrama de Classes (Definição dos Atributos das Classes)

11. Nome: Analisar Hierarquias

Intuito:

Fornecer um mecanismo para descobrir possíveis papéis (mais de uma função) e hierarquias de herança que possam estar contidos nos arquivos de dados (.dbf) do sistema legado.

Solução:

1. Encontrar, no sistema legado, os arquivos de dados com campos opcionais. A Figura 14(a) mostra outro sistema legado no qual o arquivo de dados Cliente.dbf (entidade

Cliente) é considerado um arquivo com atributos opcionais, contendo informações tanto de Cliente Pessoa Física, como de Cliente Pessoa Jurídica. Portanto, esse arquivo de dados é definido como uma super-classe, sendo criadas as subclasses Pessoa Física e Pessoa Jurídica, Figura 14(b).

2. Encontrar, no sistema legado, os arquivos de dados com campos semelhantes. A Figura 15(a) mostra, em um outro sistema legado, que foram encontrados dois arquivos de dados (.dbf) com campos semelhantes, trata-se dos arquivos: PessoaFísica e PessoaJurídica. Nesse caso, é definida a super-classe Cliente, a qual terá como atributos os ***atributos repetidos*** dos dois arquivos de dados encontrados, Figura 15(b).

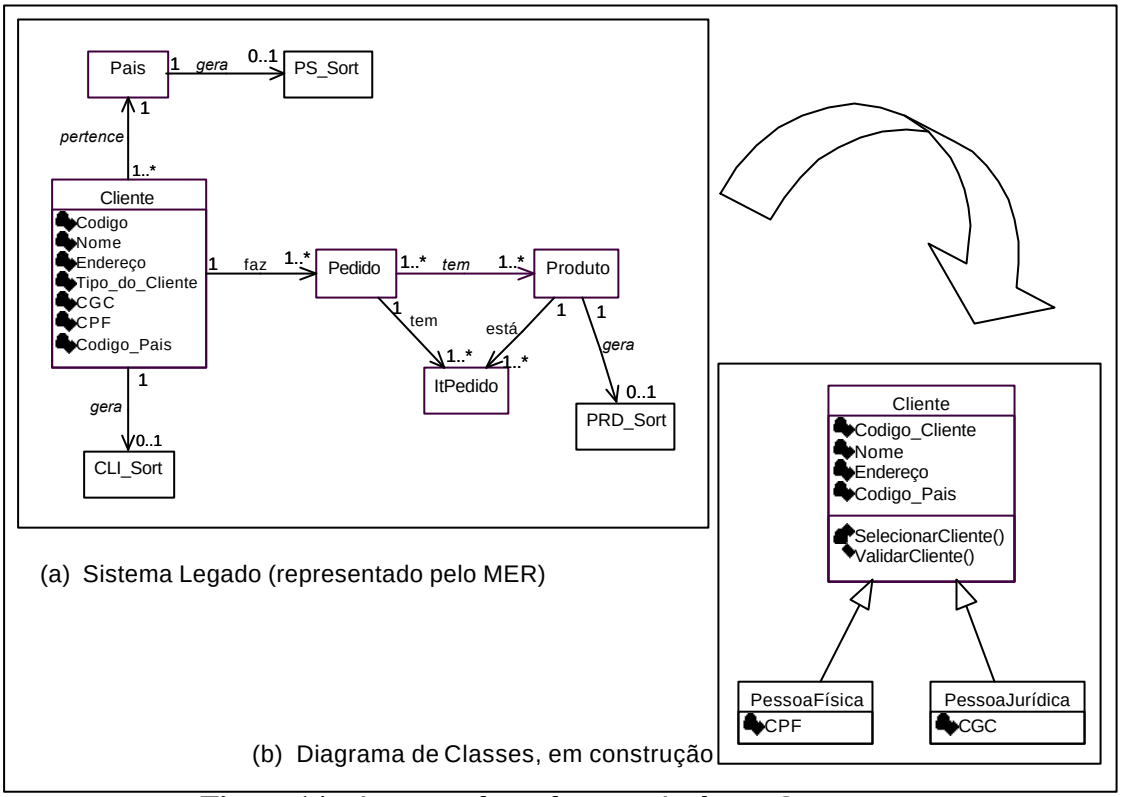


Figura 14: Arquivos de Dados com Atributos Opcionais

3. Encontrar, no MASA, as pseudo-classes nas quais a chave primária também serve como chave estrangeira de outra pseudo-classe. A Figura 16(a) mostra, para outro sistema legado, que foram encontradas, no respectivo MASA, três pseudo-classes relacionadas (por relacionamentos um-para-um), trata-se das pseudo-classes Cliente, Pessoa Física e Pessoa Jurídica. Nesse caso, após analisar a funcionalidade do sistema legado, representa-se diretamente esse conjunto de pseudo-classes como uma hierarquia de herança entre classes, Figura 16(b).

O engenheiro de software deverá observar, no MASA, situações de relacionamentos desse tipo, devendo sempre representá-los como uma hierarquia de herança entre classes no diagrama de classes em construção, para, assim, consolidar cada vez mais esse diagrama no conceito de orientação a objetos.

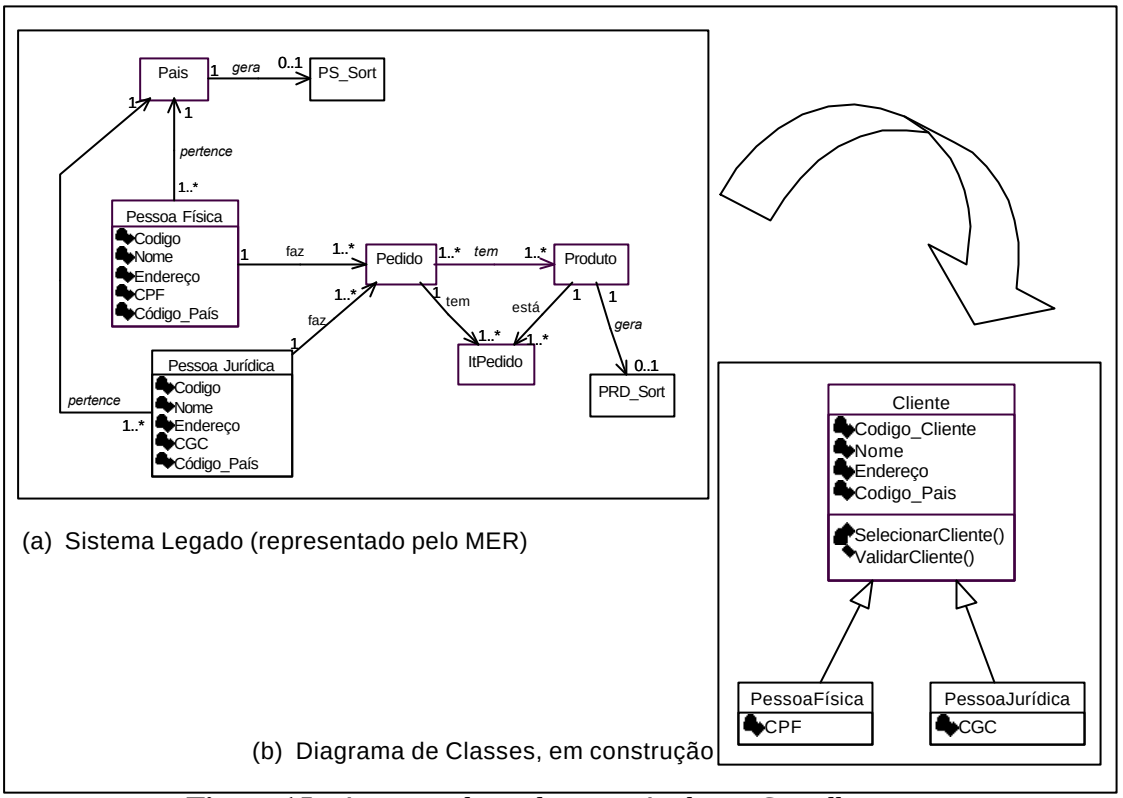


Figura 15: Arquivos de Dados com Atributos Semelhantes

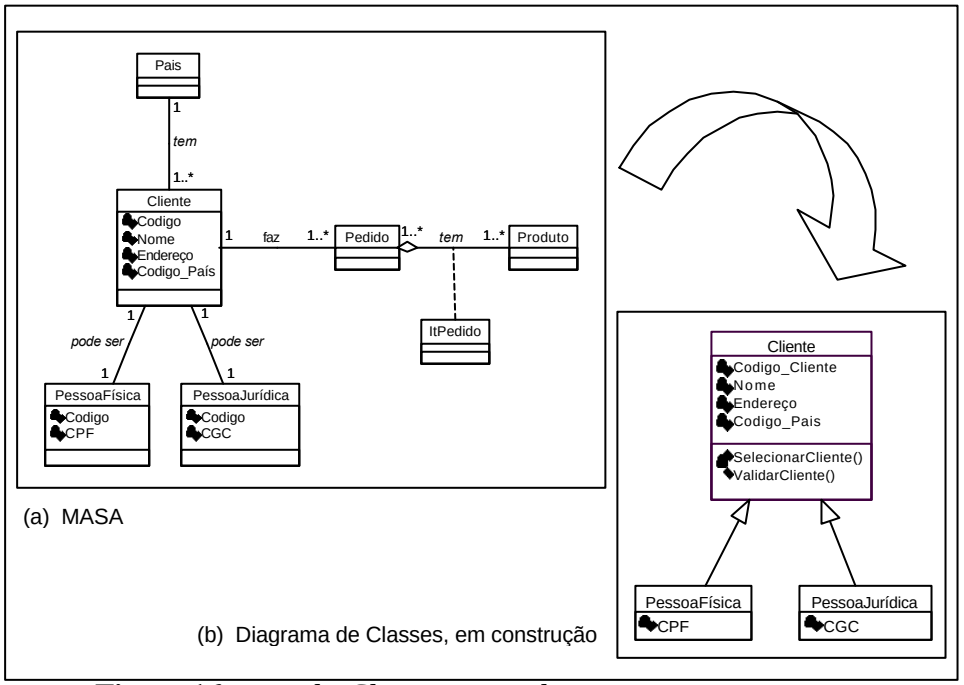


Figura 16: Pseudo-Classes com Relacionamentos Um-Para-Um

4. Encontrar, no sistema legado, os arquivos de dados (.dbf) com campos possuindo um ou mais papéis. Nesse caso, para cada papel gere uma nova classe no Diagrama de Classes.

Essa situação ocorre com freqüência em programas Cobol (cláusula *Redefines*), no entanto, nos programas em Clipper os campos de cada arquivo de dados possui unicamente um papel, logo, esse passo não requer detalhamento nesta Especialização para Clipper da FaPRE/OO.

12. Nome: Definir Métodos

Intuito:

Definir os métodos das classes pertencentes ao Diagrama de Classes em construção.

Solução:

Representar os métodos que estão relacionados na coluna Possíveis Métodos, da Tabela Detalhes de Implementação, nas classes respectivas às pseudo-classes, especificadas na coluna Pseudo-Classes Revisadas, da mesma tabela. Assim, obtém-se os métodos das classes no Diagrama de Classes em construção por meio da Descrição do Use Case correspondente. A Figura 17 ilustra o Diagrama de Classes já com os métodos representados.

Informações Adicionais:

O engenheiro de software deve observar trechos do código fonte que possam ser caracterizados como Polimorfismo, gerando tantos métodos quanto forem necessários, consolidando, assim, o diagrama de classes do sistema no paradigma de orientação a objetos. A Figura 18 ilustra um trecho do código fonte em que uma situação de Polimorfismo foi observada.

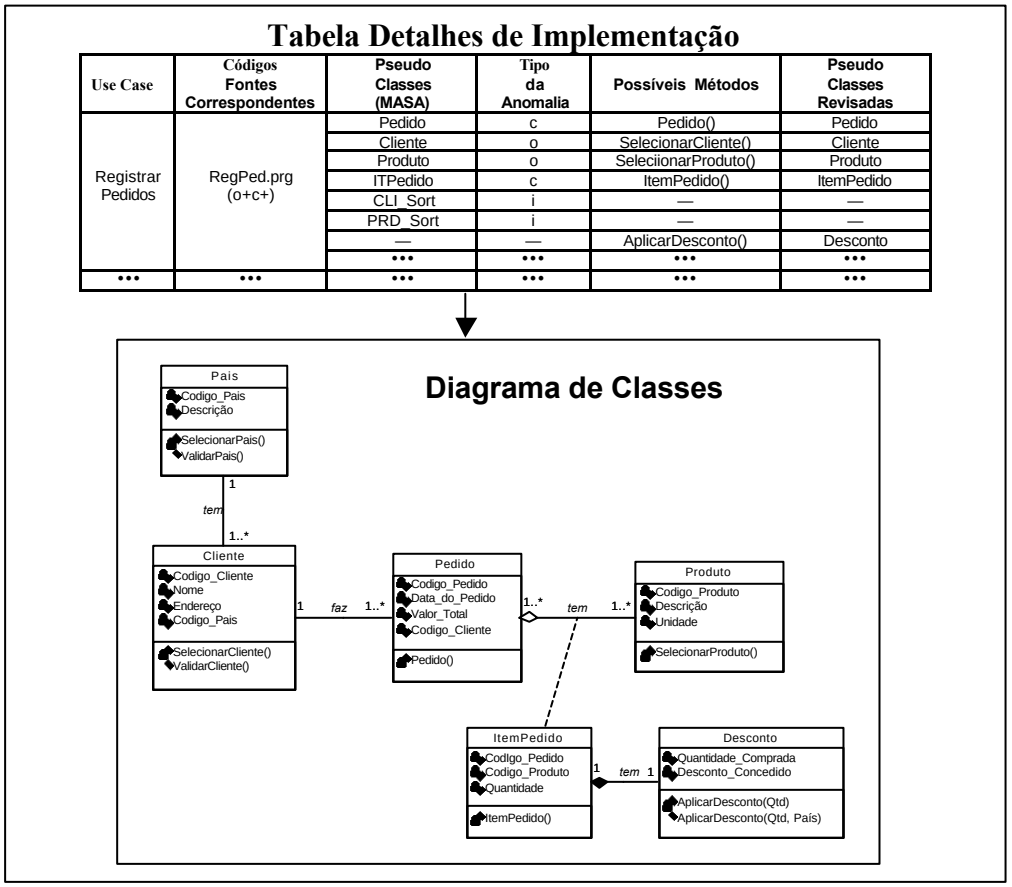


Figura 17: Diagrama de Classes (Definição dos Métodos das Classes)

```
if     Qtd < 10          //  Qtd. Comprada
         Desconto  :=  3      //    3%
elseif Qtd < 20
         Desconto  :=  7      //    7%
else
         Desconto  :=  10     //    10%
endif

if  Cliente➔CodPais <> 1              // 1: Brasil
    if     Cliente➔CodPais = 2        // 2: Argentina
             Desconto := Desconto * 0.95
    elseif Cliente➔CodPais = 3        // 3: Uruguai
             Desconto := Desconto * 0.92
    •••
    elseif Cliente➔CodPais = 15       // 15: México
             Desconto := Desconto * 0.79
    endif
endif
```

Método: AplicarDesconto (Qtd)

Método: AplicarDesconto (Qtd, País)

Figura 18: Trecho do Código Fonte, com características de Polimorfismo

13. Nome: Construir Diagramas de Seqüência

Intuito:

Documentar as regras de Negócio do legado para facilitar a sua futura engenharia avante.

Solução:

Construir todos os Diagramas de Seqüência, a partir de cada Descrição de Use Case, obtida pelo padrão Elaborar Descrição de Use Cases. Considere, como exemplo, a Descrição do Use Case Registrar Pedidos, Figura 10, que passou pelo processo de eliminação de anomalias efetuado pelo padrão Tratar Anomalias, Tabela 6, tendo sido gerados diversos métodos de forma a manter a mesma funcionalidade do legado. Agora, no Diagrama de Seqüência, Figura 19, é apresentada a interação desses métodos, mostrando graficamente a lógica da Regra do Negócio (Registrar Pedidos), no conceito de orientação a objetos.

4. Comparação Com Outros Trabalhos

Usando como suporte a abordagem Fusion/RE [11] [12] [13] e a linguagem de padrões de engenharia reversa, proposta por Demeyer et al [6], elaborou-se os padrões de engenharia reversa da Família de Padrões de Reengenharia - FaPRE/OO, para que possam ser gerados processos, passo a passo, para conduzir a engenharia reversa de sistemas legados implementados em linguagens procedimentais.

Os padrões de Demeyer foram pesquisados e avaliados durante a realização dos estudos de casos quanto à sua aplicabilidade em sistemas procedimentais [16]. Embora não se consiga realizar plenamente a engenharia reversa de sistemas procedimentais com esses padrões, eles deram suporte para a elaboração dos padrões de engenharia reversa da FaPRE/OO em sistemas legados procedimentais. A Tabela 7 mostra quais os padrões de Demeyer foram utilizados durante a construção dessa família.

A abordagem Fusion/RE auxilia o processo de engenharia reversa orientada a objetos, a partir de sistemas legados procedimentais. A Tabela 8 apresenta os padrões para o processo de engenharia reversa onde são feitas as atividades incluídas nos passos do Fusion/RE.

É importante observar que os resultados obtidos com a especialização dessa família em uma determinada linguagem de programação procedimental, gera um conjunto de padrões que irão fornecer modelos de análise e projeto orientados a objetos, viabilizando a reengenharia orientada a objetos desses sistemas.

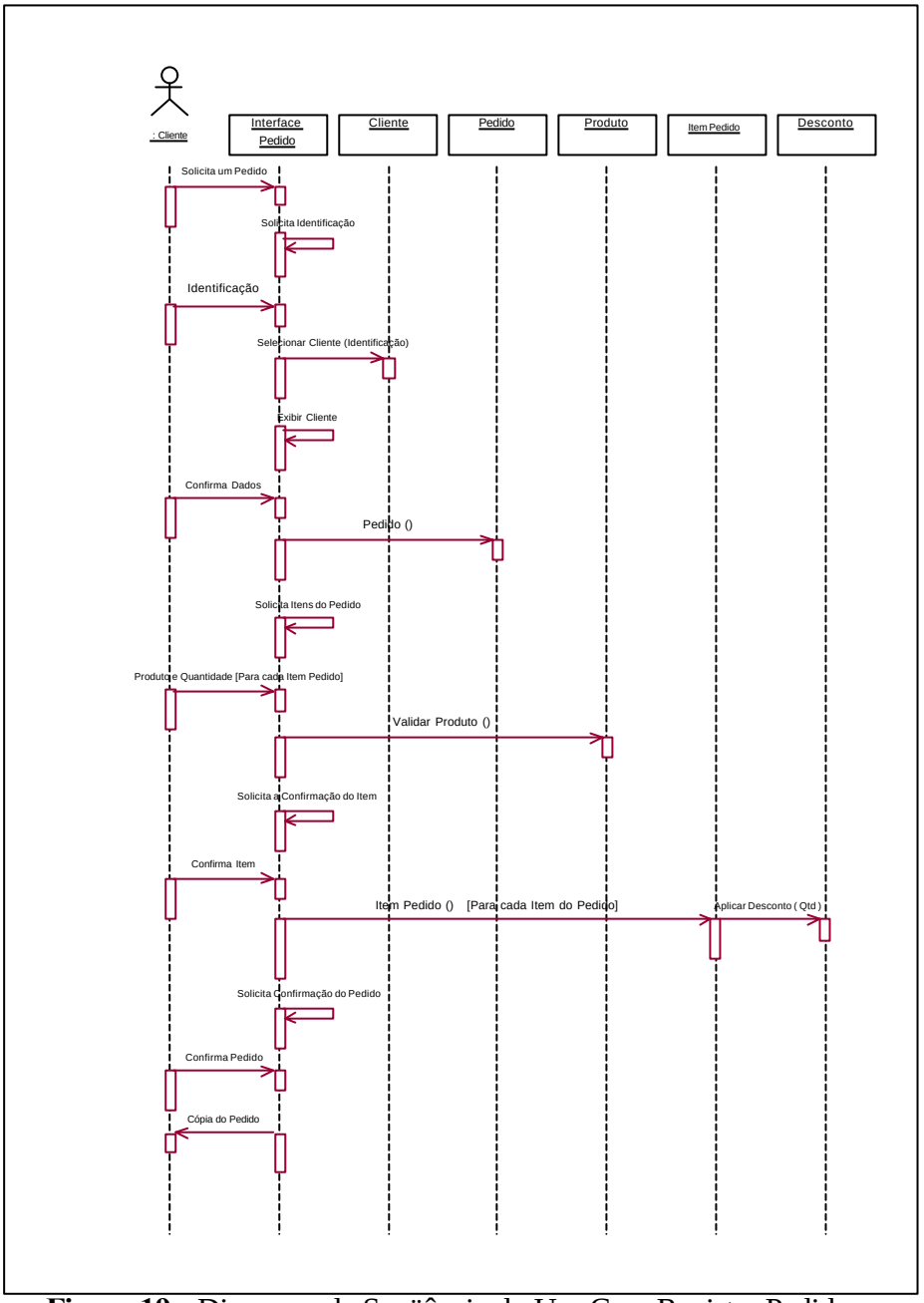


Figura 19: Diagrama de Seqüência do Use Case Registar Pedidos

Os modelos orientados a objetos produzidos são baseados em regras que vão do sistema implementado até modelos abstraídos. Seguem estritamente a notação UML [18] e fazem parte de um conjunto maior de padrões de reengenharia que inclui um *cluster* de padrões para o processo de engenharia avante com mudança de paradigma de desenvolvimento e de linguagem de programação para o orientado a objetos.

Tabela 7: Padrões de Engenharia Reversa de Demeyer utilizados durante a Construção dos Padrões de Engenharia Reversa da FaPRE/OO

Demeyer		**FaPRE/OO**	
Clusters	**Padrões**	**Padrões**	***Clusters***
Iniciação ao Sistema Legado	. Ler Todo o Código em uma Hora	—	—
	. Estudar Superficialmente a Documentação	—	—
	. Entrevistar o Usuário Durante o Sistema em Operação	. Construir Diagramas de Use Cases . Obter Cenários	Modelar a Funcionalidade do Sistema
Entendimento Inicial	. Presumir Prováveis Objetos	. Definir as Classes	Modelar o Sistema Orientado a Objetos
		. Tratar Anomalias	Modelar a Funcionalidade do Sistema
	. Examinar a Base de Dados	. Iniciar Análise dos Dados . Definir Chaves	Modelar os Dados do Legado
		. Analisar Hierarquias	Modelar o Sistema Orientado a Objetos
	. Inspecionar as Maiores Construções	—	—
	. Explorar Possíveis Modificações	—	—
Detalhamento do Sistema	. Verificar as Invocações dos Métodos	—	—
	. Observar a Execução dos Componentes	—	—
Preparação da Reengenharia	. Refazer para Entender	. Tratar Anomalias	Modelar a Funcionalidade do Sistema

Tabela 8: Especialização da Abordagem Fusion/RE

Fusion/RE		***Padrões para o Processo de Engenharia Reversa da FaPRE/OO***
Passos	**Produtos**	
1) Revitalização da Arquitetura	. Lista E/S . Estrutura de Programas	. Iniciar Análise dos Dados . Definir Chaves . Identificar Relacionamentos
2) Recuperação do Modelo do Sistema Atual (MASA)	. Temas . Modelo de Objetos . Modelo do Ciclo de Vida . Modelo de Operações	. Criar Visão OO dos Dados . Obter Cenários . Construir Diagramas de Use Cases . Elaborar a Descrição de Use Cases . Tratar Anomalias
3) Abstrair o Modelo de Análise do Sistema (MAS)	. Modelo de Objetos . Modelo do Ciclo de Vida . Modelo de Operações	. Definir as Classes . Analisar Hierarquias . Construir Diagramas de Seqüência
4) Mapear o MAS dentro do MASA	. Atributo/Elementos de Dados . Métodos/Procedimentos	. Definir Atributos . Definir Métodos . Tratar Anomalias

5. Comentários Finais

A realização do processo de reengenharia de sistemas legados é considerada como um desafio para os engenheiros de software, pois esse processo envolve muitos fatores de risco. Então, há interesse em tornar os engenheiros de software especialistas nesse processo. Para isso, surgem os padrões de engenharia reversa e de engenharia avante com o objetivo de

registrar as técnicas e mecanismos que os engenheiros de software experientes utilizam para conduzir esses processos.

Este trabalho apresentou todos os padrões para o Processo de Engenharia Reversa da FaPRE/OO, para realizar a engenharia reversa orientada a objetos de sistemas legados desenvolvidos de forma procedimental e implementados em linguagens como Algol, Clipper, Cobol, RPG II, etc.

Um sistema, originalmente desenvolvido de forma procedimental e implementado na linguagem Clipper [15], foi submetido ao processo de engenharia reversa seguindo, passo a passo, todos os padrões conforme proposto nesta Família. Assim, é realizada a engenharia reversa procedimental do sistema legado e, a partir dos resultados dessa atividade, efetua-se a engenharia reversa orientada a objetos do legado. Em outras palavras, na primeira fase obtém-se uma documentação procedimental e, na segunda fase, com base na anterior, constrói-se a documentação de análise orientada a objetos.

Outro sistema, originalmente desenvolvido de forma procedimental e implementado na linguagem Cobol [4], foi submetido ao processo de engenharia reversa usando esta Família. Nesse trabalho a engenharia reversa é realizada diretamente, isto é, sem a necessidade do produto intermediário. A documentação de análise orientada a objetos é obtida diretamente do código procedimental a fim de identificar possíveis objetos. Assim, a FaPRE/OO dá plena cobertura para conduzir a engenharia reversa diretamente orientada a objetos a partir do sistema legado procedimental. A única diferença é a ordem de aplicação dos vários padrões da Família. Como o modelo do processo é evolutivo, isso só fortalece o seu potencial em questão do domínio de sistemas de informação.

Ainda um outro sistema, desenvolvido na linguagem Delphi [9], foi submetido, com pleno sucesso, ao processo de engenharia reversa seguindo, passo a passo, os padrões propostos nesta Família. Embora o ambiente de desenvolvimento desse trabalho tivesse sido Delphi, o qual viabiliza a construção de sistemas orientados a objetos, o sistema envolvido nesse estudo de caso foi originalmente implementado sem os conceitos da orientação a objetos.

A FaPRE/OO tem as seguintes características:

- Possui quatro *clusters* de padrões claramente definidos, com regras para guiar a passagem de um padrão para outro;
- Cada padrão é dirigido a documentos que devem ser produzidos;
- Tem uma forma cíclica e evolutiva de aplicação, podendo-se, de qualquer padrão, avançar ou retroceder à sua aplicação;
- O resultado produzido é baseado, principalmente, no sistema atual e os requisitos são mínimos: o sistema executável e o código fonte;
- É um processo global que incorpora estratégias específicas para os processos de engenharia reversa e de engenharia avante, como partes do processo de reengenharia.

Agradecimentos

Agradecemos ao Shepherd Fernão Germano pelas sugestões e acompanhamento dado a este trabalho.

Referências Bibliográficas

[1] Bianchini, C. de P.; Morais, R. M. de, **"Engenharia Reversa e Reengenharia Orientada a Objetos"**, Documento de Trabalho, PPGCC-DC. Universidade Federal de São Carlos, 2000.

[2] Cagnin M. I., **"Avaliação das Vantagens quanto à facilidade de Manutenção e Expansão de Sistemas Legados submetidos a Engenharia Reversa e Reengenharia",** São Carlos-SP, 1999. Dissertação de Mestrado. Universidade Federal de São Carlos.

[3] Camargo, V., **"Reengenharia Orientada a Objetos de Sistemas COBOL com a Utilização de Padrões de Projetos e Servlets"**, São Carlos-SP, 2001. Dissertação de Mestrado. Universidade Federal de São Carlos.

[4] Camargo, V.; Recchia, E. L.; Penteado, R. – **"Aplicabilidade da Família de Padrões de Reengenharia FaPRE/OO na Engenharia Reversa Orientada a Objetos de Sistemas Legados COBOL"**, Artigo apresentado no The Second Latin American Conference on Pattern Languages of Programming – Software Pattern Applications. (SugarLoafPLoP–SPA), Itaipava-RJ, Agosto/2002.

[5] Chikofsky, J. E.; Cross, J. H. - **"Reverse Engineering and Design Recovery: A Taxonomy",** IEEE Software, v. 7, n. 1, p. 13-17, Jan. 1990.

[6] Demeyer, S.; Ducasse, S.; Nierstrasz, O., **"A Pattern Language for Reverse Engineering. Proceedings",** of the 5th European Conference on Pattern Languages of Programming and Computing, (EuroPLOP'2000), Andreas Ruping(Ed.), 2000.

[7] Dewar, R.; Lloyd, A.D.; Pooley, R.; Stevens, P. **"Identifying and Communicating Expertise in Systems Reengineering: a patterns approach".** IEEE Proceedings – Software, v.146, n.3, pp.145-152, 1999.

[8] Kulk, E.; Camargo, V. V.; Masiero, P. C.; Penteado, R.; Germano, F., **"Reengenharia Orientada a Objetos de um Sistema Contábil Implementado em Cobol para Java"**, Documento de Trabalho, 2002. Universidade de São Paulo – SP.

[9] Lemos, G. S., **"Garantia de Qualidade no Processo de Reengenharia Orientada a Objetos"**, São Carlos-SP. Dissertação de Mestrado apresentada ao PPGCC-DC. Universidade Federal de São Carlos, em Agosto/2002.

[10] Magalhães, L. F.; Soares, C. L., **"SALV – Reengenharia de um Sistema de Automação de Locadora de Vídeo de Visual Basic para Java"**, Documento de Trabalho, PPGCC-DC. Universidade Federal de São Carlos, 2000.

[11] Penteado, R. A. D., **"Um Método para Engenharia Reversa Orientada a Objetos",** São Carlos, 1996. 237 p. Tese (Doutorado em Física Computacional) - Instituto de Física de São Carlos, Universidade de São Paulo.

[12] Penteado, R., Germano, F., Masiero, P. C., **"An Overall Process Based on Fusion to Reverse Engineering Legacy Code",** In: Working Conference Reverse Engineering, 3, 1996, Monterey-California. Anais. IEEE, p. 179-188.

[13] Penteado, R., Braga, R.T.V., Masiero, P.C., **"Improving the Quality Legacy Code by Reverse Engineering",** Trabalho submetido ao 4th International Conference on Information Systems Analysis and Synthesis, ISAS/98, a ser realizado em Julho/1998, Orlando-Flórida.

[14] Recchia, E. L.; Lemos, G. S.; Deo, M. A., **"Locadora de Vídeo – Engenharia Reversa e Reengenharia"**, Documento de Trabalho, PPGCC-DC. Universidade Federal de São Carlos, 2000.

[15] Recchia, E. L., **"Engenharia Reversa e Reengenharia Baseadas em Padrões"**, São Carlos-SP. Dissertação de Mestrado apresentada ao PPGCC-DC. Universidade Federal de São Carlos, em Junho/2002.

[16] Recchia, E. L.; Penteado, R. – **Avaliação da Aplicabilidade da Linguagem de Padrões de Engenharia Reversa de Demeyer a Sistemas Legados Procedimentais**, Artigo apresentado no The Second Latin American Conference on Pattern Languages of Programming – Software Pattern Applications. (SugarLoafPLoP–SPA), Itaipava-RJ, Agosto/2002.

[17] Tavares, D. P. D.; Marucci, R. A., **"Engenharia Reversa e Reengenharia de um Sistema Legado em CLIPPER para CLIPPER OO"**, Documento de Trabalho, PPGCC-DC. Universidade Federal de São Carlos, 2000.

[18] Unified Modeling Language, 2002. URL:http//www.rational.com/uml/index.jtmpl. Consultado em 03/2002.

DCDP: A Distributed Component Development Pattern [1]

Eduardo Santana de Almeida[*]
Calebe de Paula Bianchini
Antonio Francisco do Prado
Luis Carlos Trevelin

Computing Departament – Federal University of São Carlos
Rod. Washington Luiz, km 235 – São Carlos/SP - Brazil
P.O box 676 – Zip.Code 13565-905
Phone/Fax: + 55-16-260-8232
{ealmeida, calebe, prado, trevelin}@dc.ufscar.br

Abstract

This paper presents a Distributed Component Development Pattern (DCDP) that integrates different known technologies to support the process of Distributed Component-Based Development. The involved technologies are: the Catalysis method used as a Component-Based Development (CBD) method to define, specify and design the distributed components, through CORBA architecture. The CORBA architecture to support components distribution and accessing, components frameworks for interfaces creation, guide the distribution of the developed problem domain components and facilitate the database access. A CASE tool is the main mechanism to apply this pattern, supporting the code generation of developed components.

1. Context

The proposed pattern is in the Distributed Component-Based Development (DCBD) context, which aims distributed components creation for diferent domain applications.

2. Motivation

In spite of the recent and constant researches in the Component-Based Development (CBD) area, there is still a lack of strategies, metodologies and patterns that effectively support both the development and the components reuse, in certain applications domain. Although different technologies exist to support the CBD, many difficulties are faced when trying to integrate those technologies to cover the whole CBD process, from the components creation to their use in the applications. If considering the distributed components, as they happen in the Internet with client-server platform, the problem turns even worse.

3. Problem

In the software development, the reuse is characterized by the use of software products, in a different situation for which these products were built [1]. The CBD cares about the components creation which can be reused in other applications. As a solution for this problem, the researches [1, 2, 3, 4, 5] show, as a fundamental step, the systematization of the process of analysis and components creation to a certain application domain.

---

[*] This work is supported by Fundação de Amparo à Pesquisa do Estado da Bahia (Fapesb).

In order to make the reuse effective, it must be considered in all the phases of the software development process. Therefore, the Component-Based Development must offer methods, techniques and tools that support from the components identification and specification, in a problem domain level, to their project and implementation in an object-oriented language. Besides, the CBD must use interrelations among components already existing, which have been previously tested, aiming to reduce the complexity and software development costs [2, 4].

4. Solution

Combining the Component-Based Development *Catalysis* method [2] principles, the *CORBA* architecture [6] for distributed component specification, the pattern-based frameworks and the *MVCase* [7, 8] tool, it was defined a *Distributed Component Development Pattern* (DCDP).

The pattern supports the three levels of *Catalysis*, and it is accomplished in four steps: **Define Problem**, **Specify Components**, **Project Components** and **Implement Components**, according to Figure 1.

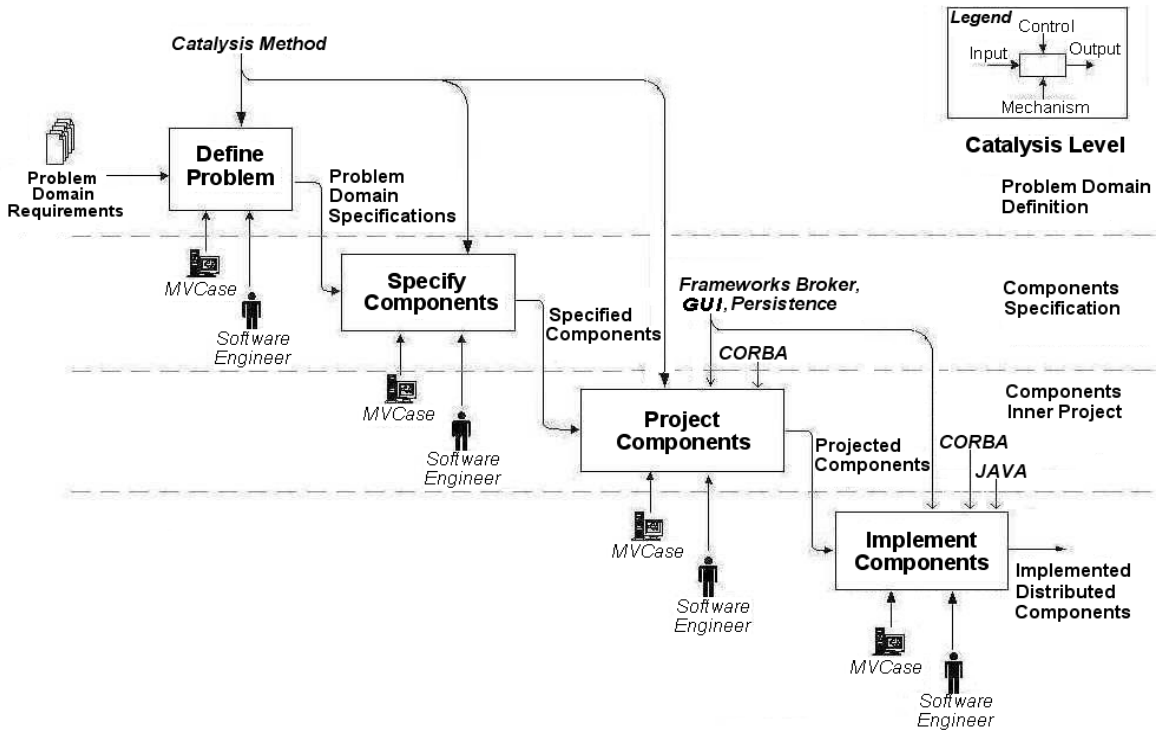


Figure 1 – DCDP: A Distributed Component Development Pattern.

It is followed a presentation of each step of the pattern, used to develop components for a Service Order domain, which belongs to a bigger domain, the Business Resources Administration, as shown in Figure 2. Although the Service Order domain is not complete, the example can show details of used techniques, and the main artifacts generated in each step of the pattern.

Figure 2 – General Structure of Business Resources Administration.

4.1 Define Problem

The Service Order (SO) domain applications are divided in three big modules: the first one, Customers, is responsible for registering and notifying customers of a certain service order; the second one, Employees, is responsible for registering employees and controlling service order tasks; the third one, Reports, is responsible for emitting reports, related to accomplished and pending tasks consultation, service orders of a certain client, and of employees responsible for each task.

In the *first step* of the pattern, the emphasis in the problem understanding, specifying "what" the components must do to solve the problem. Initially, the domain requirements are identified, using techniques as *storyboards* or *mind-maps* [2], aiming to represent the different situations and domain problem sceneries. Next, the identified requirements are specified in *Collaboration Models* [2, 9], representing the action collections and the participant objects. Finally, the collaboration models are refined in *Use Cases Model* [2, 9].

The first step of *DCDP* is summarized in Figure 3, where a *mind-map*, defined in the Service Order domain requirements identification, is specified in an *Collaboration Models*, and, later, refined and partitioned in a *Use Cases Model*, aiming to reduce the complexity and improve the problem domain understanding.

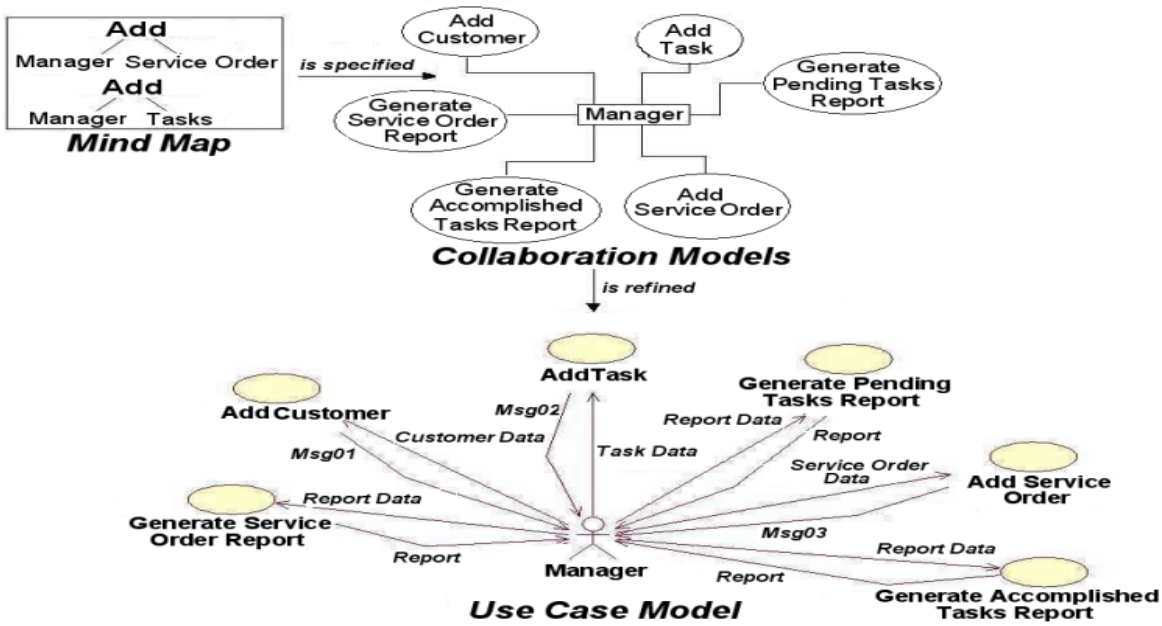


Figure 3 - First Step of Pattern: Define Problem.

4.2 Specify Components

This step supports the *second level* of *Catalysis*, where the system external behavior is described in a non ambiguous way. In the CASE tool, the software engineer refines the previous steps specifications, aiming to obtain the components specifications.

This step begins refining the problem domain models. The *Model of Types* [2] is specified, according to Figure 4, showing attributes and object type operations, without worrying about implementation. Still in this step, the data dictionary can be used to specify each type found, and the *Object Constraint Language* (OCL) [2] to detail the objects behavior, with no ambiguity.

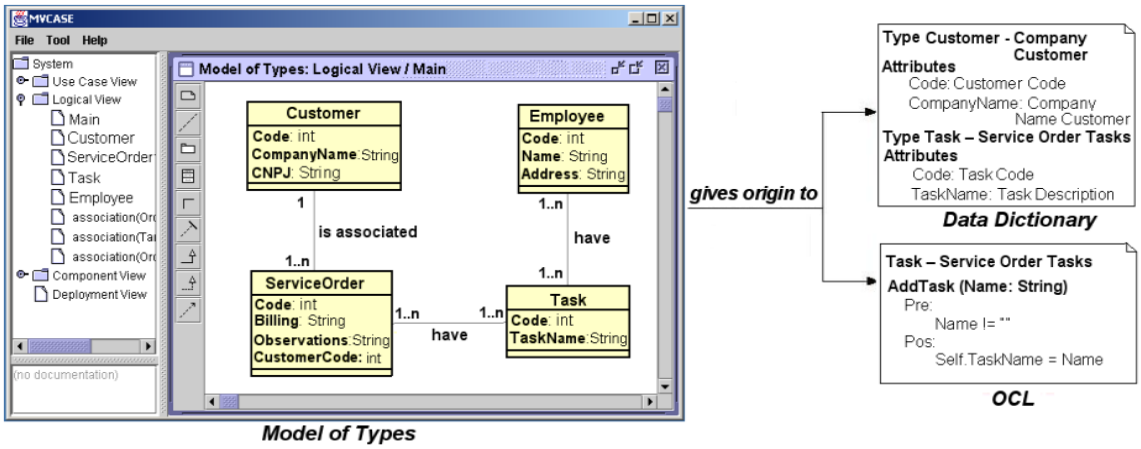


Figure 4 – Model of Types from Specify Components step.

According to Figure 5, the *Model of Types* is organized in the *Model Framework* [2], with their attributes and relationships. The *framework* is reused through the *Framework Application* [2], representing the dependences of the *framework* types, with the extended types in the application. The *Use Cases Models*, from the previous step, are refined into *Interaction Models* represented by sequence diagrams [9] to detail the components scenaries utility in different applications of the problem domain.

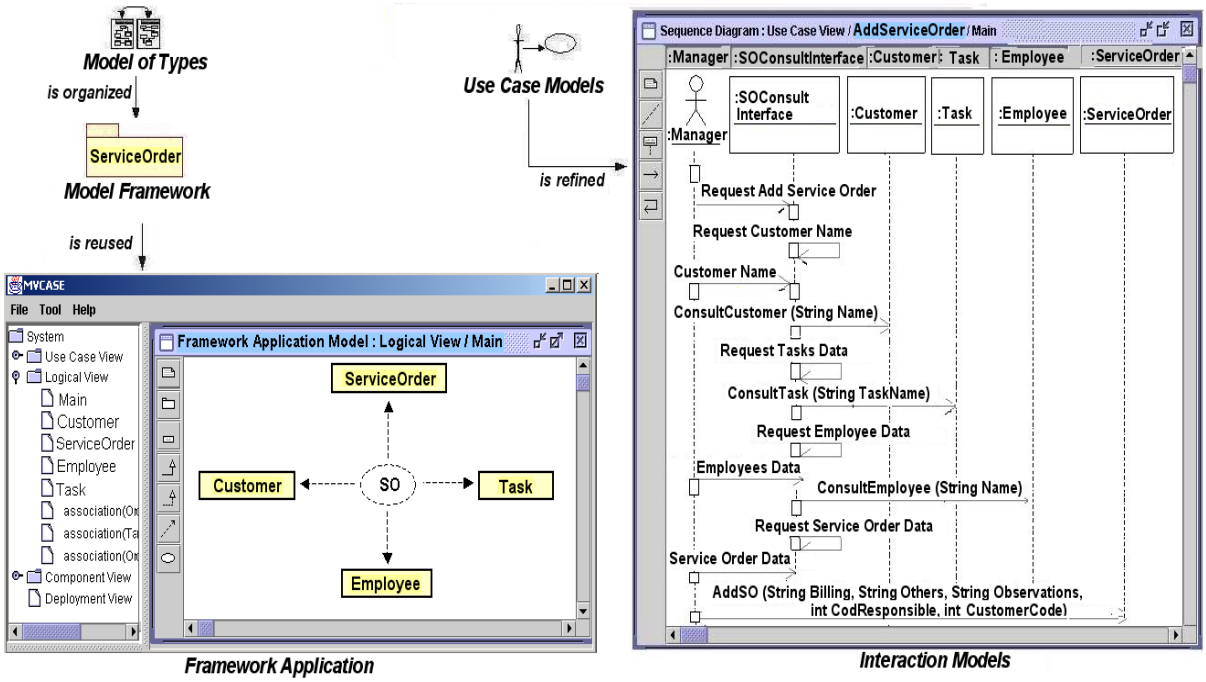


Figure 5 – Second Step of Pattern: Specify Components.

Summarizing, the activities from this step, accomplished by the software engineer, in the *MVCase* tool, include the specifications of:

a) *Model of Types*;
b) *Model Framework*;
c) *Framework Application*, based on the *Model Framework*; and
d) *Interaction Models*, represented by sequence diagrams, based on *Use Cases Model*.

These models are used in the next step of the pattern, to obtain the components inner project.

4.3 Project Components

In this step, the software engineer does the components inner project, according to the third level of *Catalysis*. Now, the implementation details become important, standing out: safety, persistence, distributed architecture and the implementation language.

As a first step, the *Model of Types* are refined into components *Classes Models* [9], where the classes are modeled with their relationships, taking into consideration the components definitions and their interfaces, according to Figure 6. The *Interaction Models*, represented by sequence diagrams techniques are refined to show design details of the methods behavior in each class.

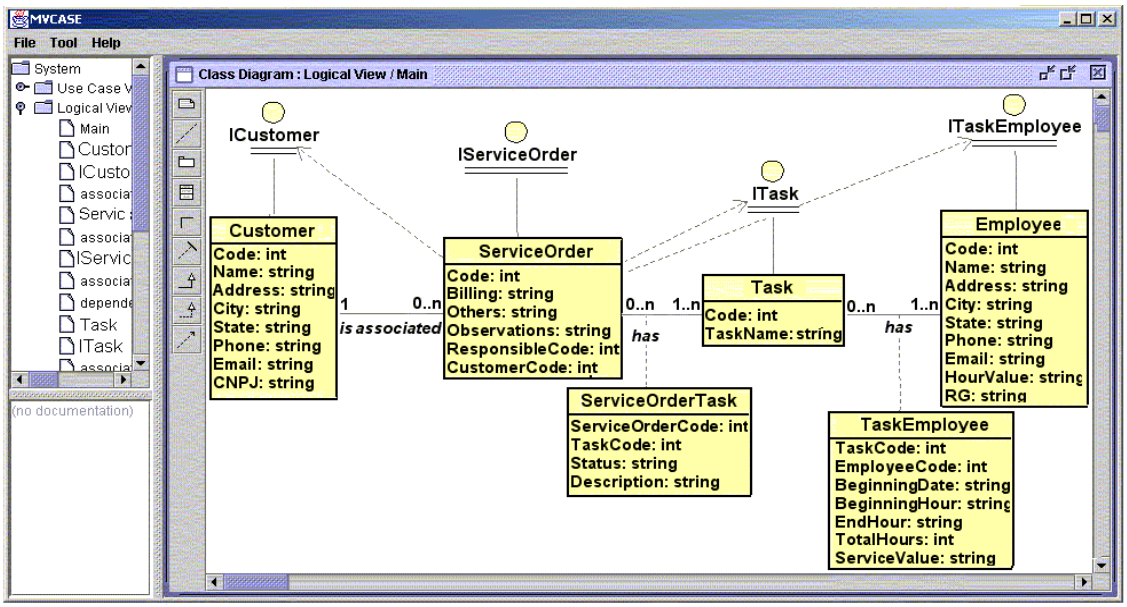


Figure 6 – Class Model obtained from Model of Types.

Starting from *Classes Model*, the *Components Model* may be designed [9], where the organizations and dependencies between components are shown. The *Components Model* may reuse components from other existing frameworks. Thus, components from related domains with non-functional requirements can be reused, as well as interface creation (GUI) [10], database persistence (Persistence) [11] and components distribution (Broker) [12]. Figure 7 shows the *Components Model* obtained from the *Classes Model* of Figure 6, reusing the components of frameworks *GUI*, *Persistence* and *Broker*.

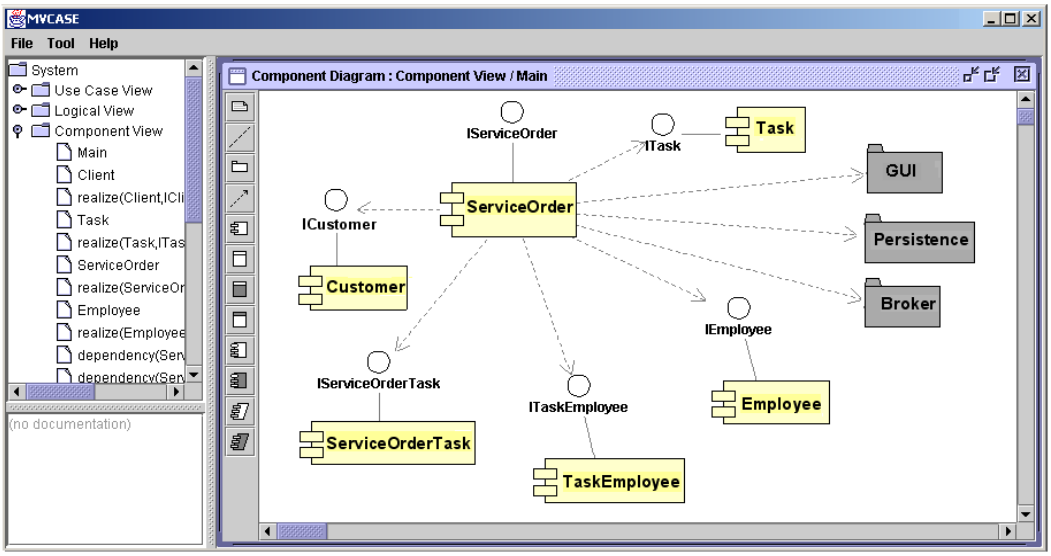


Figure 7 – Components Model.

Figure 8 shows, for example, the *GUI* framework components, which use some Design Patterns from Gamma's catalog [10], which are *Abstract Factory*, *Factory Method*, and *Singleton.* The *AbstractToolkit* component allows to create basic *widgets* components, as *frames, textfields, labels, buttons,* and other elements that compose the user interface. For a better view of the interfaces, there are, also, other *widgets* that compose the user interface, such as: *combobox*, *panel*, and *textarea* that are not in the figure, as well as the *swing* library components [13] from *Java*. The components *LinToolkit* and *WinToolkit* implement the methods defined by the *AbstractToolkit* component for *Linux* and *Windows* platform, respectively, maintaining the application portability for different environments. The *GUI framework,* has interfaces for each type of *widget,* like: *IFrame*, *ITextField, ILabel*, and *IButton.* The components *WinFrame*, *LinFrame*, *WinTextField*, *LinTextField*, *WinLabel*, *LinLabel*, *WinButton,* and *LinButton* implement the *widgets* according to the interface execution platform.

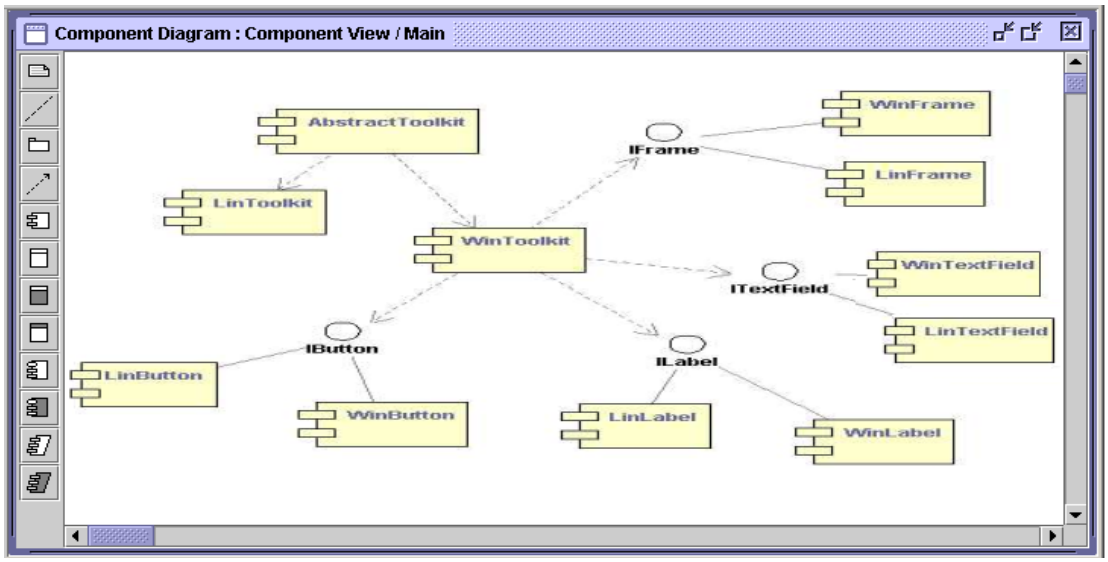


Figure 8 – Framework *GUI*.

In the same way, we have the components of other frameworks, as in the case of *Persistence framework*, which supports the information persistence in a relational database.

They are based on design patterns from Gamma's catalog which are, *Singleton*, *Facade* and others, like *PersistentObject* [11] e *ObjectPool* [11].

Figure 9 shows the *framework Persistence* components. The *ConnectionPool* component, through its *IConnectionPool* interface, does the management and connection with the database used in the application. The *DriversUtil* component, based on *eXtensible Markup Language* (XML) [14], has information from supported database drivers, available through its interface *IDriversUtil*. The *TableManager* component manages the mapping of an object into database tables, making their methods available by the *ITableManager* interface. The persistent component of the *FacadePersistent* structure, through its *IPersistentObject* interface, makes the values which must be added to the database available, passing parameters to the *TableManager* component.

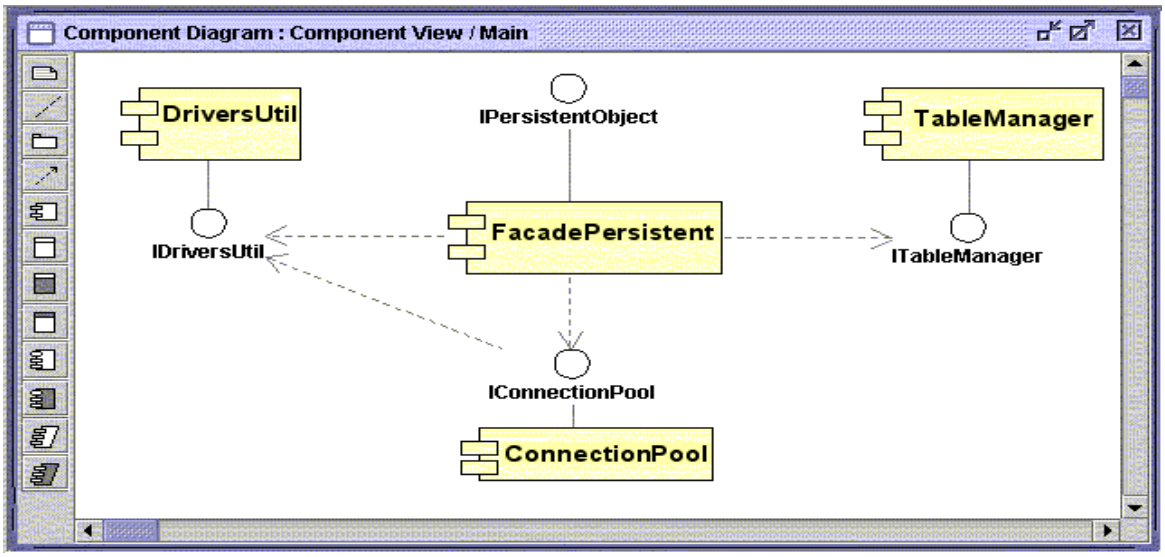


Figure 9 – *Framework Persistence.*

The components in the *Broker framework* use the *Distributed Adapters Pattern* (DAP) [12] to implement remote communication between two components. The technique adopted by *DAP* to offer this functionality is to insert a pair of adapter components, seeking a better component unjoining in a distributed architecture. The adapters, basically, encapsulate the API needed to remote access *Target* components. This way, *Sources* components of an application in relation to the distributed aspects, and any change on this aspects does not cause impact on it autonomous. Figure 10 shows the *Broker framework* structure. The *Source* and *Target* components abstract business rules, from a problem domain. The *TargetInterface* interface abstracts the *Target* component behavior in a distributed scenary. Both this interface, and the *Source* and *Target* components, do not have communication code. These three elements make an independent distribution layer.

The main components of *framework Broker* are *SourceAdapter* and *TargetAdapter*. They are connected to a specific distribution API and encapsulate the communication details. *SourceAdapter* is an adapter that isolates the *Source* component from distributed code. It is located in the same machine than *Source* and works as a proxy to *TargetAdapter*. *TargetAdapter* is located in an other machine, isolating the *Target* component from distributed code. *SourceAdapter* and *TargetAdapter*, usually, are located in different machines, and do not directly interact. *TargetAdapter* implements *RemoteInterface* used to connect with *SourceAdapter*. The *Initializer* component is located in the same computer as *Target* and *TargetAdapter* components, and it is responsible for the creation of *Target* and *TargetAdapter* components [12] as can be seen in the Figure 10.

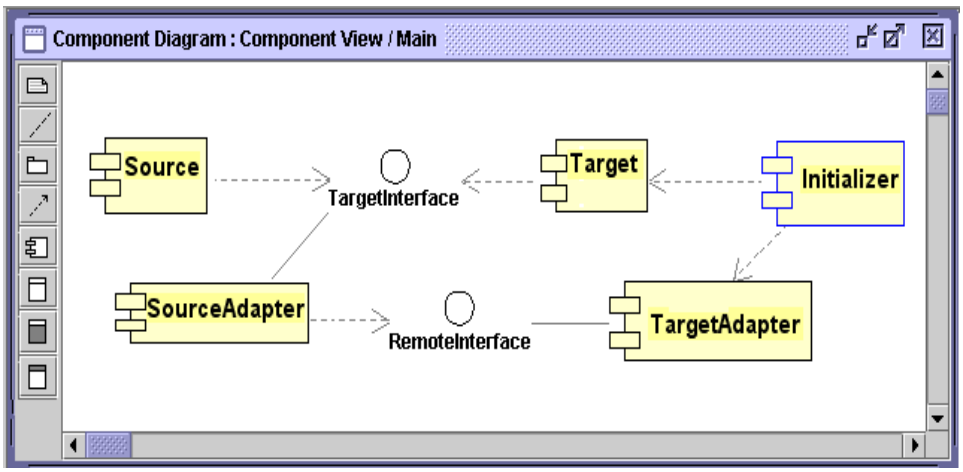


Figure 10 –*Broker* Framework structure.

Figure 11 summarizes the main artifacts and the sequence of the Project Components activities, which include:

a) Refining *Model of Types* into *Classes Models*;
b) Refining the *Interaction Models*; and
c) Creating the *Components Models* reusing existents components.

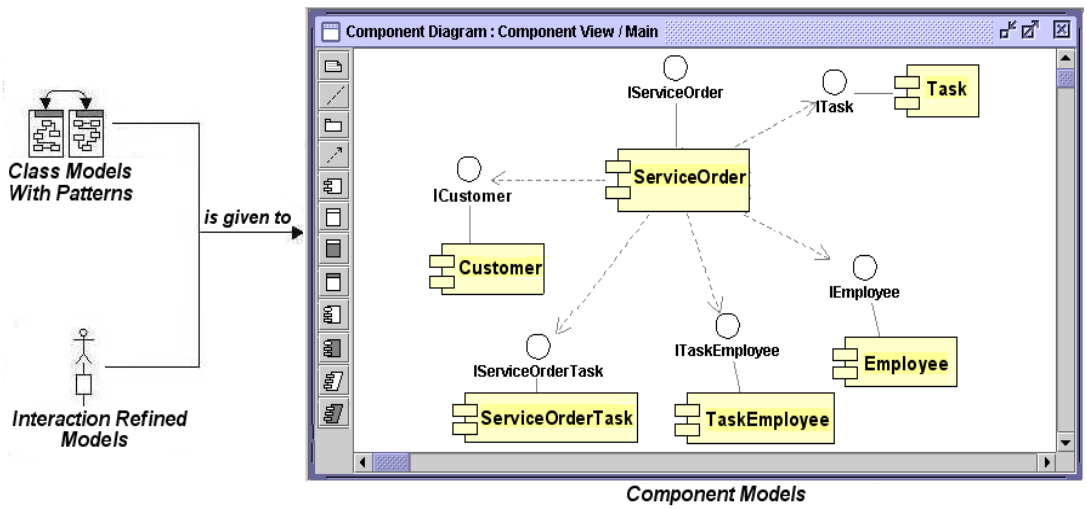


Figure 11 – Third step of Pattern: Project Components.

4.4. Implement Components

At last, the software engineer uses a code generator, from *MVCase* to implement the designed components. As the components are distributed, the code is generated using *CORBA*. For each component, there are the *stubs* and *skeletons* and their interfaces that makes its services avaliable.

Figure 12 shows part of the Service Order domain components designed, and the respective *Interface Definition Laguage* (IDL) [6] and *Java* code generated. The code generation is done through the *idlj* utilitary, native from *Java Development Kit* (JDK) avaliable in the tool.

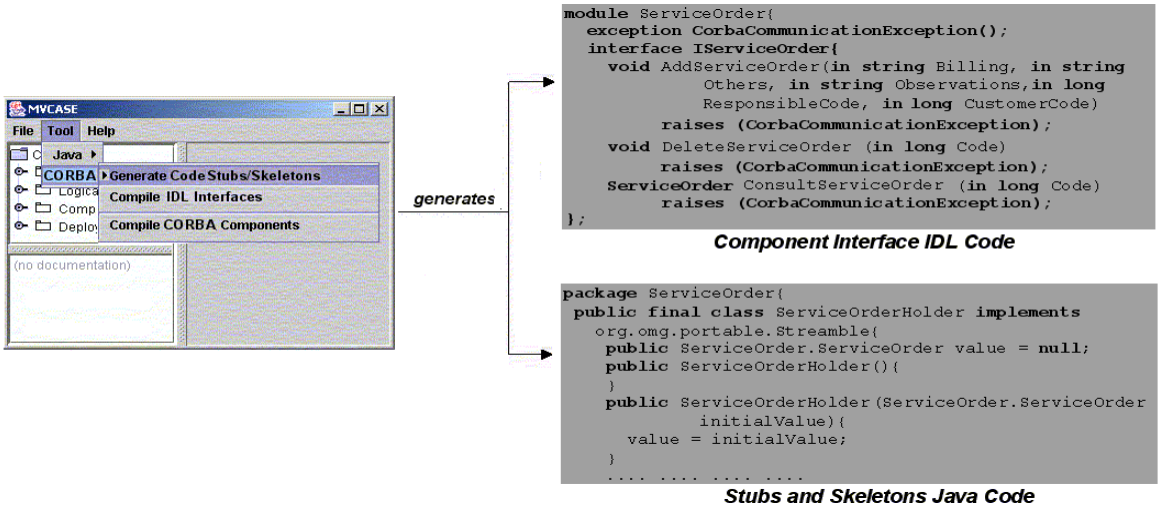


Figure 12 – Fourth step of Pattern: Implement Components.

Summarizing, the main activities of the Implement Components step, accomplished by the software engineer, in the *MVCase* tool, are the generation of the *IDL* an *Java* code the distributed components.

5. Consequences

DCDP offers the following benefits:

- *Modularity*: the pattern allows to separate distribution aspects from interface and database persistence;
- *Reuse*: through the modularity aspects offered, separating the tiers, the developers can reuse the components for several applications of the created domain, reducing the code redundancy;
- *Partial Automation*: with the *MVCase* tool support, great part of the activities posed in the pattern can be executed automatically.

Even having the advantages listed above, the following disadvantages can be presented:

- *Incremental class number* [12]: using the *DAP* pattern, using a pair of adapters, initialization and nomination components are necessary; anyway, these structures can be generated through partial automation, using the *MVCase* tool;
- *Knowledge about other technologies*: using the *Persistence framework*, the software engineer needs to know technologies, like *XML*, for definition of information related to database management systems, as connection port, username, password, and others.

6. Implementation

To support the pattern proposed to Distribution CBD, different methods, techniques and tools are used, which are presented next.

6.1 *Catalysis* Method

The pattern is based on the *Catalysis* method for CBD, which has three levels: **Problem Domain Definition**, where it is put emphasis in the problem understanding, specifying “what” the system must do to solve the problem; **Components Specification**,

where the system behavior is described in a non ambiguous way; and the **Components Inner Project**, where it is defined "how" the specified requirements will be implemented.

Catalysis is based on the principles of *abstraction*, *precision* and *"plug-in" components*. The *abstraction* principle guides the developer in search of essential aspects of the system, sparing details that are not relevant for the context of the system. The *precision* principle has as objective to detect errors and inconsistency in modeling. The *"plug-in" components* principle supports components reuse to construct other systems.

6.2 *Common Object Request Broker Architecture (CORBA)*

In the CBD it is necessary to establish a formal relation among the components and the application that uses them, through well-defined interfaces. To meet these requirements, the *DCDP* is based on the *CORBA* [6] architecture, which is a pattern established by *Object Management Group* (OMG) to support distributed objects. *CORBA* presents well-defined interfaces and independent of applications, through the *IDL* [6], that fits perfectly in the CBD context.

Other aspects that motivated the use of *CORBA* were: the programming language independence, due to the possibility of mapping from *IDL* to several languages; the portability among computational environments and services of Safety, Nomination and Notification, offered by the specification.

6.3 Frameworks and Patterns

To construct software components safer, more reliable, easier to maintain and use, the CBD uses frameworks techniques based on Patterns [10]. Framework is a set of related classes that make reuse of a project for specific software classes [10] The use of patterns in complex software systems allows previously tested solutions to be reused, making the system more comprehensible, flexible, easy to develop and maintain. The objective of using software patterns is the spread of the software developing solutions already existing.

6.4 MVCase Tool

CASE tools have been used, with success, in the project and re-project of systems to be reconstructed. Among the CASE tools, stands out MVCase [7, 8], which supports system specification using UML techniques, generates code, automatically, in an object-oriented programming language, starting from high-level specifications, using distributed components specified in *IDL*.

MVCase implements a three-tier architecture to construct the components. The three tiers allow that the software engineer separate the client applications from the remote client (thin client) interface, business rules, and from the database storing services oranother way of storing.

The components of these tiers can be distributed in different platforms, supporting client-server applications, which can be executed by the Internet.

7. Related Patterns

- *Abstract Factory, Factory Method* and *Singleton.* The *GUI* framework is implemented using the Design Patterns [10] *Abstract Factory, Factory Method* and *Singleton.*
- *Broker* and *Trader*. Well known patterns for structuring distributed systems already exist. The *Broker* [15] and *Trader* [15] patterns are examples. These are architectural patterns and focus mostly on providing fundamental distribution issues, such as marshalling and message protocols. Therefore, they are mostly tailored to the implementation of distributed platforms, such as *CORBA. DAP* uses these fundamental patterns and provides a higher level of abstraction: distribution API transparency to both clients and servers [12].
- *Wrapper-Facade* [15] *and DAP* have the common goal of minimizing platform-specific variation in application code. However, *Wrapper-Facade* encapsulates existing lowerlevel non-object-oriented APIs (such as sockets, and threads), whereas *DAP* encapsulates object-oriented distribution APIs, such as *RMI* and *CORBA* [12].
- *Facade, PersistentObject and ObjectPool*. *Framework Persistence* is implemented using the Design Patterns *Singleton and Facade*, and, patterns for database persistence [11], like *PersistentObject and ObjectPool.*

8. Acknowledgements

The authors woul like to thank to Shepherd Dr. Jugurta Lisboa Filho for sugestions received during the process and Rosana Teresinha Vaccare Braga for all contribution. This work is supported by Fundação de Amparo à Pesquisa do Estado da Bahia (Fapesb).

9. References

[1] Jacobson, I., Griss, M., Jonsson, P., 1997. *Software Reuse: Architecture, Process and Organization for Business Sucess*, Addison-Wesley. Longman.

[2] D'Souza, D., F., Wills, A., C., 1999. *Objects, Components, and Frameworks with UML, The Catalysis Approach*, Addison-Wesley. USA.

[3] Heineman, G., T., Councill, W., T., 2001. *Component-Based Software Engineering, Putting the Pieces Together*, Addison-Wesley. USA.

[4] Szyperski, C., 1998. *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley. USA.

[5] Cheesman, J., Daniels, J., 2000. *UML Components: A Simple Process for Specifying Component-Based Software.* Addison-Wesley. USA, 1nd edition.

[6] The Common Object Request Broker Architecture, 1996. Object Management Group. Avaliable in 10/04/2001, URL: http:// www.omg.org.

[7] Almeida, E., S., Bianchini, C., P., Prado, A., F., Trevelin, L., C., 2002. MVCase: An Integrating Technologies Tool for Distributed Component-Based Software Development. In *APNOMS'2002, The Asia-Pacific Network Operations and Management Symposium,Poster Session*. Proceedings of IEEE.

[8] Almeida, E., S., Lucrédio, D., Bianchini, C., P., Prado, A., F., Trevelin, L., C., 2002. MVCase Tool: An Integrating Technologies Tool for Distributed Component Development *(in portuguese)*. In *SBES'2002, 16th Brazilian Symposium on Software Engineering, Tools Session*.

[9] Rumbaugh, J., et al., 1998. *The Unified Modeling Language Reference Manual*, Addison-Wesley. USA.

[10] Gamma, E., et al., 1995. *Elements of Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley.

[11] Yoder, J., Johnson, R., E., Wilson, Q., D., 1998. *Connecting Business Objects to Relational Databases.* In *PLoP'1998, Pattern Language of Progamming.*

[12] Alves, V., Borba, P., 2001. Distributed Adapters Pattern (DAP): A Design Pattern for Object-Oriented Distributed Applications. In *SugarLoafPlop'2001, The First Latin American Conference on Pattern Languages of Programming.*

[13] Horstmann, C., S., Cornell, G., 2002. *Core Java 2: Volume II, Advanced Features,* Prentice Hall.
[14] eXtensible Markup Language, 2000. World Wide Web Consortium. Avaliable in 10/04/2001, URL: http:// www.w3.org/xml.
[15] Buschmann, F., et al, 1996. *Pattern Oriented Software Architecture: A System of Patterns.* John Wiley & Sons.

O Uso de Padrões na Integração de Visões Modeladas com UML

Vânia M. P. Vidal
Departamento de Computação
Universidade Federal do Ceará – UFC
Campus do Pici Bloco 960
Fortaleza, CE – Brasil
vvidal@lia.ufc.br

Fabiana G. Marinho*
Instituto Atlântico
Rua Chico Lemos 946
Cidade dos Funcionários
Fortaleza, CE – Brasil
fabiana@atlantico.com.br

Resumo

Durante o projeto conceitual de um banco de dados, as visões dos usuários são abstraídas e representadas. Em seguida, essas visões são integradas em um esquema conceitual global (esquema integrado) que satisfaz os requisitos de toda a organização. Neste artigo, apresentamos uma metodologia para integração de visões modeladas com UML[1]. O processo de integração proposto em nossa metodologia usa o catálogo de padrões desenvolvido por Marinho[8].

Abstract

During database conceptual project, user views are represented and integrated in a global conceptual schema (integrated schema) that completely satifies the organization requirements. In this article, we discuss an view integration methodology using UML[1]. The integration process proposed in our methodology uses the patterns catalog developed by Marinho[8].

1 Introdução

O projeto de um banco de dados é composto de duas fases principais - projeto conceitual e projeto lógico. No projeto conceitual, o projetista especifica o banco de dados em termos do modelo semântico adotado. O resultado dessa fase constitui o esquema conceitual da aplicação. No projeto lógico, o esquema conceitual é traduzido em um dos modelos tradicionais de implementação, resultando no esquema lógico da aplicação.

Durante o projeto do esquema conceitual, as visões dos vários usuários são abstraídas e representadas. Essas visões, além de proteger o acesso aos dados, ajudam a alcançar um certo grau de independência lógica, uma vez que é possível alterar o esquema do banco de dados sem alterar uma visão. As visões são então integradas em um esquema conceitual global que satisfaz os requisitos de toda a organização, denominado esquema integrado.

Observamos que pouco trabalho tem sido feito com UML no sentido de aprimorar sua capacidade de construir esquemas conceituais globais que representem adequadamente os requisitos dos usuários e aplicações. Definir uma metodologia que facilite e que, de certa forma, padronize o processo de integração desses esquemas é necessário.

Neste trabalho, propomos uma metodologia para integração de visões modeladas com UML. O processo de integração de visões proposto está baseado no uso de padrões de modelagem. Esses padrões têm como objetivo auxiliar os projetistas nas atividades que compõem o processo de integração de visões, de modo a assegurar que essa tarefa seja realizada de forma segura e eficiente.

A motivação para o desenvolvimento desse estudo foi encontrada na ineficácia existente nas atuais práticas de integração de visões, quando aplicadas a situações específicas do mundo real. A

*Este trabalho foi suportado pelo Instituto Atlântico (*www.atlantico.com.br*).

UML foi utilizada por se tratar de uma linguagem de modelagem unificada, tendo mostrado todos os sinais de se tornar a linguagem de modelagem padrão para especificar, visualizar, documentar e construir aplicações orientadas a objeto. Nosso estudo tomou como ponto de partida vários trabalhos realizados para integração de visões e modelagem conceitual dos dados ([14], [7], [11], [3], [9], [2], [10], [5], [4], [12]).

Neste artigo, também formalizamos algumas restrições de integridade e elementos do modelo de objetos da UML. Esse formalismo é necessário para validar as soluções propostas no catálogo de padrões adotado.

Nas seções a seguir descrevemos nossa abordagem para o processo de integração de visões e como o catálogo de padrões é utilizado nesse processo. Na seção 2, descrevemos a metodologia desenvolvida para a integração de visões. Na seção 3, formalizamos o significado dos elementos do modelo de objetos da UML. Na seção 4, apresentamos as restrições de integridade utilizadas na validação dos padrões descritos. Na seção 5, apresentamos alguns dos padrões que compõem o catálogo de padrões. Na seção 6, apresentamos as conclusões e direcionamentos futuros do nosso trabalho.

2 Características Gerais da Metodologia

Nesta seção, descrevemos as características gerais da metodologia proposta para tratar o processo de integração de visões utilizando o diagrama de classes da UML.

Conforme ilustrado na Figura 1, nosso enfoque consiste em 4 fases:

(i) Primeiramente, cada grupo de usuários analisa seus requisitos de dados e especifica as visões dos dados na forma de um esquema conceitual.

(ii) Em seguida, essas visões individuais são combinadas em um esquema conceitual global e a definição conceitual das visões é criada. A definição conceitual das visões consiste dos esquemas das visões originais e de um conjunto de assertivas de correspondência (ACs) que especifica como cada visão é definida em termos do esquema conceitual global obtido. Usamos as assertivas de correspondência para especificar o relacionamento das visões dos usuários com o esquema integrado. Esse passo é bastante complexo devido às diversas representações utilizadas pelos projetistas na modelagem das visões. Todo o processo de integração das visões proposto em nossa metodologia está baseado no uso de padrões de modelagem.

(iii) Após a integração das visões, o esquema conceitual global é mapeado para o esquema lógico. Esse esquema lógico é representado em termos do modelo de dados específico do sistema de gerenciamento de banco de dados adotado.

(iv) Neste último passo, as definições conceituais das visões são traduzidas em definições lógicas em termos do esquema lógico obtido. As ACs também são mapeadas do esquema conceitual para o esquema lógico. Esse passo não é abordado em nosso trabalho.

2.1 O Processo de Integração de Visões

O processo de integração de visões proposto em nossa metodologia consiste em 5 etapas. Cada etapa está baseada no uso de padrões de modelagem. Nesta seção, descrevemos como esses padrões são usados em cada passo do processo de integração de visões.

(i) **Decomposição:** certas formas de dependências funcionais (DFs) sugerem a presença de redundâncias no esquema conceitual. O padrão **P1** remove essas DFs indesejáveis, de modo a minimizar a redundância do esquema conceitual.

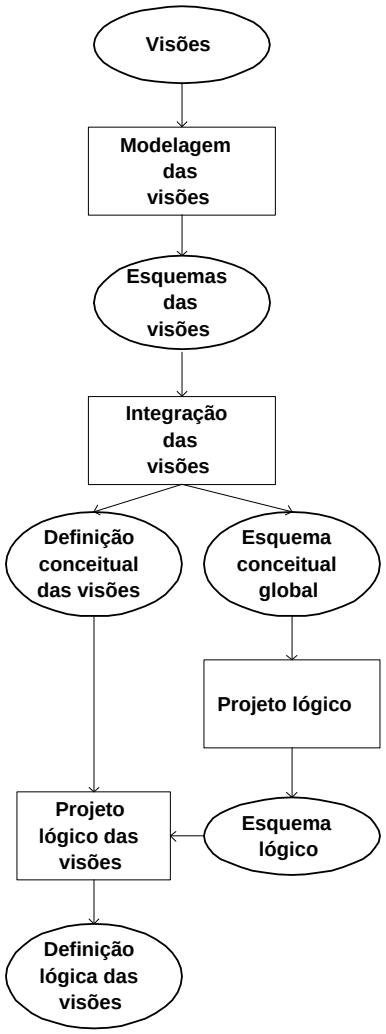


Figura 1: Projeto de Banco de Dados Através da Integração de Visões

(ii) **Combinação I:** durante a etapa de combinação I, as várias visões dos usuários são analisadas e comparadas para determinar correspondências entre extensões de classes. O resultado desse processo é o esquema combinado inicial, o qual contém todas as visões originais e um conjunto de ACs de extensão. Os padrões **P2** e **P3** são usados para auxiliar na identificação dessas ACs.

(iii) **Otimização I:** uma vez identificada a equivalência entre extensões das classes, aplicamos ao esquema combinado inicial uma série de transformações, de modo a obter um esquema global otimizado. Os padrões **P10**, **P11** e **P21** têm como objetivo reduzir o tamanho do esquema através da fusão de classes e atributos comuns.

(iv) **Combinação II:** o objetivo dessa segunda etapa de combinação é identificar relacionamentos semânticos entre classes de associação e entre associações. O resultado da fase de combinação II é um novo esquema combinado composto por todas as visões originais e um novo conjunto de ACs que expressam o relacionamento entre classes de associação e associações. Os padrões **P4**, **P5**, **P6**, **P7**, **P8** e **P9** são utilizados para identificar os relacionamentos semânticos citados.

(v) **Otimização II:** nessa etapa, os padrões **P12**, **P13**, **P14**, **P15** ,**P16**, **P17**, **P18**, **P19** e **P20** reestruturam os esquemas novamente, de forma a eliminar redundâncias identificadas na etapa (iv).

3 Os Elementos de Modelagem da UML

Nesta seção formalizamos alguns elementos de modelagem que compõem o diagrama de classes da UML. Esse formalismo é necessário porque possibilita a validação das soluções propostas nos padrões adotados.

Os elementos básicos do diagrama de classes são classe, atributo, associação e classe de associação. A seguir apresentamos as extensões propostas para o diagrama de classe da UML envolvendo as definições de *ligações monovaloradas* e *ligações multivaloradas*, *caminho* e *classe de associação*.

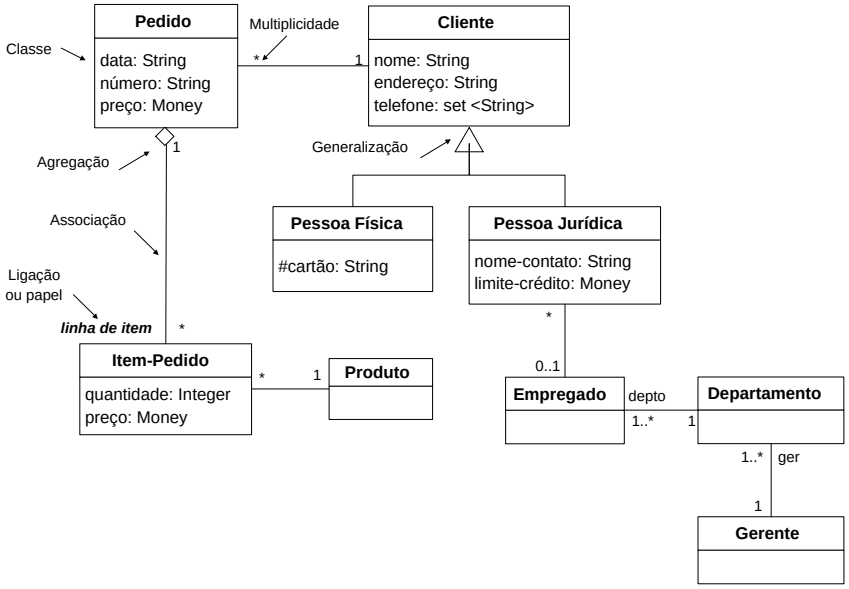


Figura 2: Exemplo - Diagrama de Classes da UML

Para cada direção de navegação de uma associação é associada uma *ligação* ou papel. Usaremos a notação $l : \mathbf{A} \rightarrow \mathbf{B}$ para indicar que as instâncias da classe **A** estão relacionadas com instâncias da classe **B** através da ligação l. Neste caso, dizemos que l é uma ligação da classe **A**. Por exemplo, na Figura 2, temos que as instâncias da classe **Item-Pedido** estão associadas a uma instância da classe **Pedido** através da ligação *linha de item* (*linha de item*: **Item-Pedido** $\rightarrow$ **Pedido**).

As ligações também possuem uma multiplicidade. Se o valor máximo da multiplicidade especificada para uma ligação for 1, então dizemos que a ligação é *monovalorada*; caso seja maior que 1, a ligação é *multivalorada*.

Dado uma classe de associação **C** entre as classes $\mathbf{C_1}$, ..., $\mathbf{C_n}$, então existe uma ligação monovalorada $l_i : \mathbf{C} \rightarrow \mathbf{C_i}$, para $1 \leq i \leq n$. Em geral, nomeamos a ligação l_i com o nome da classe $\mathbf{C_i}$. Por exemplo, no esquema da Figura 3, a classe de associação **Gerência** captura o fato de que um empregado **e** trabalha em um projeto **p** que é gerenciado pelo gerente **g**. Existe uma ligação da classe de associação **Gerência** para cada uma de suas classes participantes cuja multiplicidade é 1.

Instâncias de uma classe podem estar associadas com instâncias de outras classes através da composição de duas ou mais ligações. Considere, o diagrama da Figura 2. De acordo com as ligações definidas entre as classes **Empregado**, **Departamento** e **Gerente**, temos que um empregado está relacionado com o gerente do seu departamento de forma indireta através da composição das ligações *depto* e *ger*, ou seja, existe um *caminho* entre as classes **Empregado** e **Gerente** ($depto \bullet ger$). Uma definição formal de caminho é apresentada a seguir.

Definição 3.1 *(Caminho) Sejam* $\mathbf{C_1}, \mathbf{C_2}, \ldots \mathbf{C_{n+1}}$ *classes de um esquema tais que existe uma ligação* l_i *de* $\mathbf{C_i}$ *para* $\mathbf{C_{i+1}}$ *(*$l_i : \mathbf{C_i} \rightarrow \mathbf{C_{i+1}}$*),* $1 \leq i \leq n$*. Assim sendo,* $\delta = l_1 \bullet l_2 \bullet ... \bullet l_n$ *é um caminho de* $\mathbf{C_1}$*.*

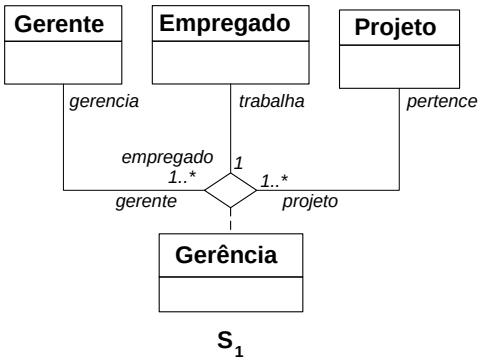


Figura 3: Exemplo - Classe de Associação

Um caminho é *monovalorado* se todas as ligações que o compõem são monovaloradas. Um caminho é *multivalorado* se pelo menos uma das ligações que o compõe é multivalorada.

Definição 3.2 *(Estado de Classe e de Objeto) Suponha o diagrama de classes $\mathcal{S} = (\mathcal{C}, \mathcal{I})$, onde $\mathcal{C}$ representa um conjunto finito de classes e $\mathcal{I}$ um conjunto de restrições de integridade definidas sobre as classes em $\mathcal{C}$. O estado de uma classe $\mathbf{C}$ é o conjunto de objetos que pertencem à extensão de $\mathbf{C}$, ou seja, o conjunto de objetos que são instâncias de $\mathbf{C}$ em um determinado instante. O estado de um objeto é definido pelos valores de suas propriedades (atributos e ligações) em um determinado instante. Considere $\mathbf{c}$ uma instância da classe $\mathbf{C}$. Para qualquer atributo $\mathbf{a}$ de $\mathbf{C}$, $\mathbf{c.a}$ retorna o valor do atributo $\mathbf{a}$ para a instância $\mathbf{c}$ no estado corrente. Para qualquer ligação l: $\mathbf{C} \rightarrow \mathbf{C'}$, $\mathbf{c}.l$ retorna as instâncias de $\mathbf{C'}$ que estão associadas com $\mathbf{c}$ através da ligação l no estado corrente. Para qualquer caminho $\delta = l_1 \bullet l_2 \bullet ... \bullet l_n$ de $\mathbf{C}$, onde $l_1 : \mathbf{C} \rightarrow \mathbf{C_1}$ e $l_i : \mathbf{C_{i-1}} \rightarrow \mathbf{C_i}$, $2 \leq i \leq n$, $\mathbf{c}.\delta$ retorna as instâncias de $\mathbf{C_n}$ que estão associadas com $\mathbf{c}$ através do caminho δ no estado corrente.*

4 Restrições de Integridade em UML

As restrições de integridade capturam a semântica do mundo real representada nos esquemas conceituais. A UML, no entanto, não define uma sintaxe explícita para descrever essas restrições. Nesta seção, definimos uma notação específica para representar restrições de chave, restrições de dependência funcional, restrições de dependência existencial e assertivas de correspondência.

Uma restrição de chave especifica a unicidade nos valores dos atributos de uma classe. A seguir definimos formalmente a restrição de chave.

Definição 4.1 *(Chave) Seja $\mathbf{C}$ uma classe e $\mathbf{a_1}$, $\mathbf{a_2}$, ..., $\mathbf{a_n}$ atributos de $\mathbf{C}$. A restrição de chave $\mathbf{a_1}$, $\mathbf{a_2}$, ..., $\mathbf{a_n} \in$ Chaves($\mathbf{C}$) especifica que para quaisquer instâncias $\mathbf{e_1}$ e $\mathbf{e_2}$ de $\mathbf{C}$ se $\mathbf{e_1.a_i} = \mathbf{e_2.a_i}$, $1 \leq i \leq n$, então $\mathbf{e_1} \equiv \mathbf{e_2}$ ($\mathbf{e_1}$ e $\mathbf{e_2}$ são semanticamente equivalentes, isto é, representam o mesmo objeto do mundo real).*

O diagrama de classes da UML não possui uma definição precisa para restrições de dependência funcional. A seguir, propomos uma notação formal para esse tipo de restrição.

Definição 4.2 *(Dependência Funcional) Seja $\mathbf{C}$ uma classe de $\mathcal{S}$ e $\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_n}$, $\mathbf{q}$ propriedades de $\mathbf{C}$. A dependência funcional $\mathbf{C}[\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_n} \rightarrow \mathbf{q}]$ especifica que para quaisquer instâncias $\mathbf{c_1}$ e $\mathbf{c_2}$ de $\mathbf{C}$, se $\mathbf{c_1.p_i} = \mathbf{c_2.p_i}$, $1 \leq i \leq n$, então $\mathbf{c_1.q} \equiv \mathbf{c_2.q}$.*

As restrições de dependência existencial (DEs) são importantes para dar suporte à reestruturação de esquemas conceituais, pois permitem expressar formalmente a equivalência semântica de componentes de esquemas. A seguir, definimos formalmente alguns tipos de DEs utilizadas nos padrões apresentados neste artigo. Suponha $\mathbf{C_1}$ e $\mathbf{C_2}$ classes, $\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_n}$ propriedades de $\mathbf{C_1}$ e $\mathbf{q_1}, \mathbf{q_2}, ..., \mathbf{q_n}$ propriedades de $\mathbf{C_2}$.

Definição 4.3 *(DE de Subconjunto) A restrição de DE* $\mathbf{C_1}[\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_n}] \subset \mathbf{C_2}[\mathbf{q_1}, \mathbf{q_2}, ..., \mathbf{q_n}]$ *especifica que para qualquer instância* $\mathbf{c_1}$ *de* $\mathbf{C_1}$ *existe uma instância* $\mathbf{c_2}$ *de* $\mathbf{C_2}$ *tal que* $\mathbf{c_1.p_i} = \mathbf{c_2.q_i}$, $1 \leq i \leq n$.

Definição 4.4 *(DE de Equivalência) A restrição de DE* $\mathbf{C_1}\ [\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_n}] \equiv \mathbf{C_2}\ [\mathbf{q_1}, \mathbf{q_2}, \ldots, \mathbf{q_n}]$ *especifica que* $\mathbf{C_1}\ [\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_n}] \subset \mathbf{C_2}\ [\mathbf{q_1}, \mathbf{q_2}, \ldots, \mathbf{q_n}]$ *e* $\mathbf{C_2}\ [\mathbf{q_1}, \mathbf{q_2}, \ldots, \mathbf{q_n}] \subset \mathbf{C_1}\ [\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_n}]$.

As assertivas de correspondência são tipos especiais de restrições de integridade usadas para especificar a correspondência entre componentes de esquemas. Existem vários tipos de ACs, dependendo dos elementos envolvidos e da natureza da correspondência. No presente trabalho, consideramos apenas as *assertivas de correspondência de extensão* (ACEs) e *assertivas de correspondência de caminhos* (ACCs).

As ACEs representam os diferentes tipos de relacionamentos existentes entre as extensões das classes de um ou mais esquemas. Em Marinho[8] são definidos nove tipos de ACEs. A seguir definimos os dois tipos de ACEs mais comuns e que são usados nos padrões apresentados neste artigo. Suponha $\mathbf{C_1}$ e $\mathbf{C_2}$ classes de um esquema.

Definição 4.5 *(ACE de Subconjunto) A ACE de subconjunto* $\mathbf{C_1} \subset \mathbf{C_2}$ *especifica que para qualquer instância* $\mathbf{e_1}$ *de* $\mathbf{C_1}$, *existe uma instância* $\mathbf{e_2}$ *em* $\mathbf{C_2}$, *tal que* $\mathbf{e_1} \equiv \mathbf{e_2}$.

Definição 4.6 *(ACE de Equivalência) A ACE de Equivalência* $\mathbf{C_1} \equiv \mathbf{C_2}$ *especifica que* $\mathbf{C_1} \subset \mathbf{C_2}$ *e* $\mathbf{C_2} \subset \mathbf{C_1}$.

As ACCs especificam relacionamentos entre caminhos de classes semanticamente relacionadas. As classes $\mathbf{C_1}$ e $\mathbf{C_2}$, são semanticamente relacionadas quando suas instâncias podem representar objetos semanticamente equivalentes. A seguir definimos os dois tipos de ACCs mais comuns e que são usadas nos padrões apresentados neste artigo.

Definição 4.7 *(ACC de Equivalência) Sejam* δ_1 *e* δ_2 *caminhos (monovalorados ou multivalorados) das classes* $\mathbf{C_1}$ *e* $\mathbf{C_2}$ *respectivamente, onde* $\mathbf{C_1}$ *e* $\mathbf{C_2}$ *são classes semanticamente relacionadas. A ACC de equivalência* $\mathbf{C_1}.\delta_1 \equiv \mathbf{C_2}.\delta_2$, *especifica que para quaisquer* $\mathbf{e_1}$, $\mathbf{e_2}$ *instâncias de* $\mathbf{C_1}$ *e* $\mathbf{C_2}$, *respectivamente, se* $\mathbf{e_1} \equiv \mathbf{e_2}$ *então* $\mathbf{e_1}.\delta_1 \equiv \mathbf{e_2}.\delta_2$.

Definição 4.8 *(ACC de Subconjunto) Sejam* δ_1 *e* δ_2 *caminhos multivalorados das classes* $\mathbf{C_1}$ *e* $\mathbf{C_2}$ *respectivamente, onde* $\mathbf{C_1}$ *e* $\mathbf{C_2}$ *são classes semanticamente relacionadas. A ACC de subconjunto* $\mathbf{C_1}.\delta_1 \subset \mathbf{C_2}.\delta_2$, *especifica que para quaisquer* $\mathbf{e_1}$, $\mathbf{e_2}$ *instâncias de* $\mathbf{C_1}$ *e* $\mathbf{C_2}$, *respectivamente, se* $\mathbf{e_1} \equiv \mathbf{e_2}$ *então* $\mathbf{e_1}.\delta_1 \subset \mathbf{e_2}.\delta_2$.

5 O Catálogo de Padrões

Nesta seção descrevemos brevemente o catálogo de padrões para integração de visões modeladas com UML. Os padrões estão agrupados em três categorias principais, classificadas de acordo com a etapa do processo de integração de visões que eles tratam (ver seção 2.1): *Padrão de Decomposição*, *Padrões de Combinação* e *Padrões de Otimização*.

Padrão de Decomposição

P1. *Decompondo Classes para Eliminar DFs Indesejáveis* - identifica no esquema conceitual a presença de classes embutidas representadas por restrições de dependência funcional.

Padrões de Combinação

P2. *Identificando Correspondências entre Extensões de Classes* - identifica ACs entre extensões de classes de um esquema conceitual.

P3. *Identificando Associações Ocultas* - identifica associações ocultas a partir da análise do esquema conceitual.

P4. *Identificando Correspondências entre Classes de Associação* - identifica restrições de DE entre classes de associação.

P5. *Identificando Correspondências entre uma Classe de Associação e uma Classe Participante da Classe de Associação* - identifica ACs entre extensões de uma classe de associação e uma das classes participantes da classe de associação.

P6. *Identificando Correspondências entre Associações* - identifica restrições de DE entre associações.

P7. *Identificando Associações Derivadas* - verifica se uma associação é derivável da composição de duas ou mais associações.

P8. *Identificando Correspondências entre uma Classe de Associação e uma Associação* - identifica DEs entre uma classe de associação e uma associação definida entre duas classes participantes da classe de associação.

P9. *Identificando Classes de Associação Derivadas* - verifica se uma classe de associação é derivável da composição das associações existentes entre as classes participantes da classe de associação.

Padrões de Otimização

P10. *Integrando Classes Equivalentes* - reestrutura o esquema conceitual de modo a capturar as ACEs de equivalência.

P11. *Eliminando Redundâncias Capturadas por ACEs de Subconjunto* - reestrutura o esquema conceitual de modo a capturar as ACs de subconjunto entre extensões das classes.

P12. *Representando Correspondências entre uma Classe de Associação e uma Classe Participante da Classe de Associação* - reestrutura o esquema conceitual a partir das correspondências semânticas identificadas entre extensões de uma classe de associação e uma das classes participantes da classe de associação.

P13. *Integrando Associações Equivalentes* - reestrutura o esquema conceitual capturando as ACs de equivalência entre associações.

P14. *Eliminando ACs de Subconjunto entre Associações* - reestrutura o esquema conceitual capturando as ACs de subconjunto entre associações.

P15. *Removendo Associações Derivadas* - reestrutura o esquema conceitual de modo a remover associações deriváveis da composição de duas ou mais associações.

P16. *Removendo Classes de Associação Derivadas* - reestrutura o esquema conceitual para remover classes de associação deriváveis da composição das associações existentes entre as classes participantes da classe de associação.

P17. *Integrando Classes de Associação Equivalentes* - reestrutura o esquema conceitual visando remover restrições de DE de equivalência entre classes de associação.

P18. *Eliminando Redundâncias Capturadas por DEs de Subconjunto entre Classes de Associação* - reestrutura o esquema conceitual visando remover restrições de DE de subconjunto entre classes de associação.

P19. *Integrando Classes de Associação e Associações Equivalentes* - reestrutura o esquema conceitual para capturar as restrições de DE de equivalência entre classes de associação e associações.

P20. *Eliminando DEs de Subconjunto entre uma Classe de Associação e uma Associação* - reestrutura o esquema conceitual para capturar as restrições de DE de subconjunto entre classes de associação e associações.

P21. *Adicionando Associações Ocultas* - reestrutura o esquema conceitual adicionando associações ocultas identificadas.

A Figura 4 apresenta a classificação geral dos padrões, assim como seus inter-relacionamentos. As elipses representam respectivamente as categorias dos problemas abordados. Os padrões são representados por retângulos. As setas implicam na existência de um relacionamento entre os padrões, ou seja, implicam que um padrão usa ou depende dos resultados obtidos com a aplicação do outro.

Devido à limitação de espaço, selecionamos alguns padrões mais relevantes do catálogo de padrões utilizado. Os padrões **P2** e **P11** estão relacionados com a integração de classes semanticamente relacionadas. Os padrões **P4** e **P18** focalizam a integração de classes de associação. Já os padrões **P7** e **P15** abordam a remoção de associação derivadas do esquema conceitual. Nosso objetivo é mostrar como os padrões podem contribuir para a qualidade das atividades presentes no processo de integração de visões de um banco de dados.

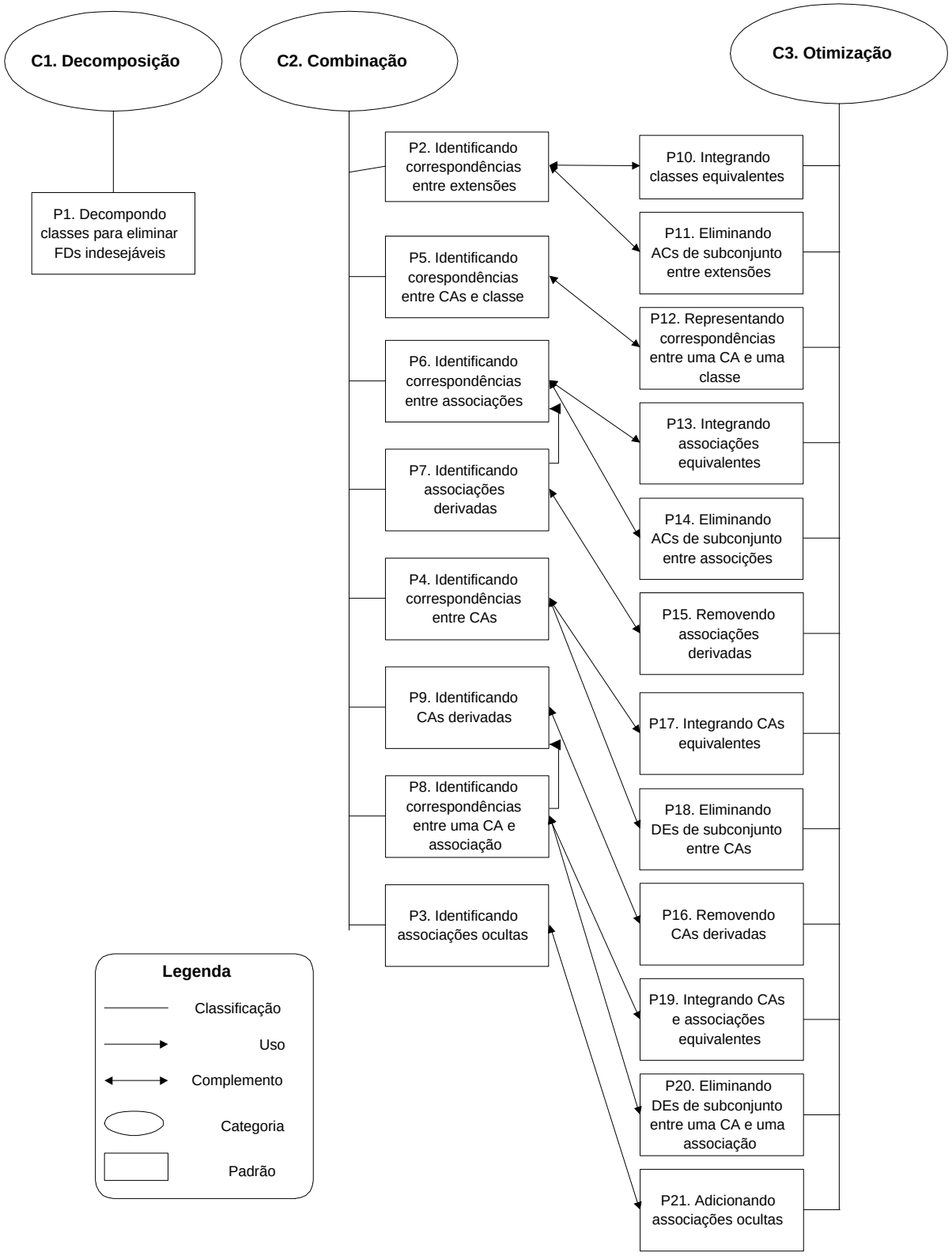


Figura 4: Classificação dos Padrões para Integração de Visões com UML

P2. Identificando Correspondências entre Extensões de Classes

O padrão **P2** é usado durante a fase de combinação I e identifica uma situação de possível correspondência entre extensões de classes.

Contexto: Considere o esquema $\mathbf{S_1}$ mostrado na Figura 5.

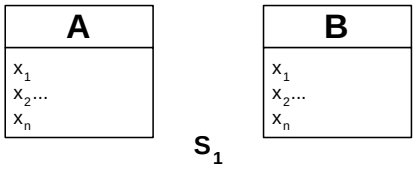


Figura 5: Classes com Atributos em Comum

Sejam **A** e **B** classes. Suponha que $\mathbf{x_1}$, $\mathbf{x_2}$,..., $\mathbf{x_n}$ sejam atributos em comum das classes **A** e **B**.

Problema: Verificar se existe um relacionamento semântico entre as extensões das classes **A** e **B**.

Solução: A semântica da aplicação deve ser analisada para verificar se:

1. existe uma ACE de subconjunto dada por $\mathbf{A} \subset \mathbf{B}$ ou
2. existe uma ACE de equivalência dada por $\mathbf{A} \equiv \mathbf{B}$.

Exemplo P2.1: Considere o esquema $\mathbf{S_2}$ mostrado na Figura 6. Analisando as classes **EMPREGADO** e **GERENTE**, verificamos que elas possuem um conjunto de atributos em comum: os atributos **cpf**, **nome** e **salário**. Essa situação sugere a existência de um relacionamento semântico entre as extensões das classes **EMPREGADO** e **GERENTE**.

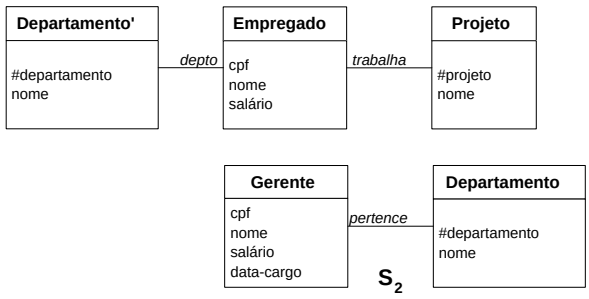


Figura 6: Exemplo - Classes com Atributos Similares

Analisando a semântica associada ao esquema $\mathbf{S_2}$, identificamos que todo gerente é um empregado. Isso significa que para toda instância **g** de **GERENTE** existe uma instância **e** de **EMPREGADO**, tal que $\mathbf{g} = \mathbf{e}$. Esse relacionamento é formalmente capturado pelas ACs abaixo:

Assertiva de Correspondência de Extensão:

(i) `GERENTE` $\subset$ `EMPREGADO`

Assertivas de Correspondência de Caminho:

(ii) `GERENTE.cpf` $\equiv$ `EMPREGADO.cpf`

(iii) `GERENTE.nome` ≡ `EMPREGADO.nome`

(iv) `GERENTE.salário` ≡ `EMPREGADO.salário`

(v) `GERENTE.`*pertence* ≡ `EMPREGADO.`*depto*

Usos Conhecidos: Uma discussão relacionada à identificação de correspondências semânticas entre extensões pode ser encontrada em Storey[13].

Padrões Relacionados: No padrão **P11**, propomos a reestruturação dos esquemas conceituais de forma a capturar as ACEs identificadas.

P4. Identificando Correspondências entre Classes de Associação

O padrão **P4** é utilizado durante a fase de combinação II e identifica situações de possíveis correspondências entre classes de associação.

Contexto: Suponha o esquema **S** mostrado na Figura 7.

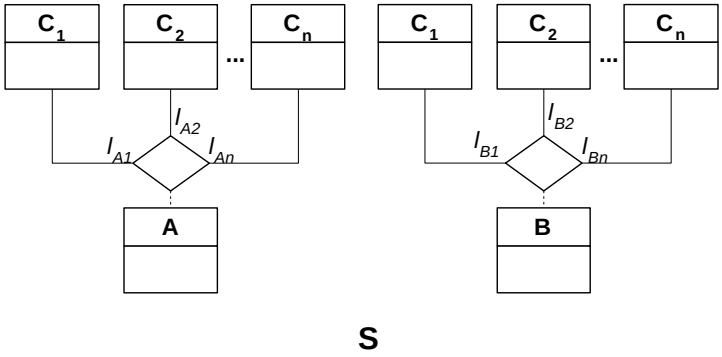


Figura 7: DE entre Classes de Associação

Sejam **A** e **B** classes de associação. Suponha que $\mathbf{C_1}$, $\mathbf{C_2}$, ..., $\mathbf{C_n}$ sejam classes em comum participantes das classes de associação **A** e **B**. Considere $l_{Ai} : \mathbf{A} \rightarrow \mathbf{C_i}$ e $l_{Bi} : \mathbf{B} \rightarrow \mathbf{C_i}$, $1 \leq \mathrm{i} \leq \mathrm{n}$.

Problema: Verificar se existe uma restrição de dependência existencial entre as classes de associação **A** e **B**.

Solução: A semântica deve ser analisada para verificar se:

1. existe uma restrição de DE de subconjunto entre **A** e **B**, dada por **A** $[l_{A1}, l_{A2}, ..., l_{An}] \subset$ **B** $[l_{B1}, l_{B2}, ..., l_{Bn}]$ ou
2. existe uma restrição de DE de equivalência entre **A** e **B**, dada por **A** $[l_{A1}, l_{A2}, ..., l_{An}] \equiv$ **B** $[l_{B1}, l_{B2}, ..., l_{Bn}]$.

É importante observar que as ACEs são casos especiais de DEs. No caso de **A** $[l_{A1}, l_{A2}, ..., l_{An}] \subset$ **B** $[l_{B1}, l_{B2}, ..., l_{Bn}]$ e $[l_{A1}, l_{A2}, ..., l_{An}]$ conter a chave de **A** e $[l_{B1}, l_{B2}, ..., l_{Bn}]$ conter a chave de **B**, então existe uma ACE de subconjunto entre **A** e **B** dada por **A** $\subset$ **B**.

No caso de **A** $[l_{A1}, l_{A2}, ..., l_{An}] \equiv$ **B** $[l_{B1}, l_{B2}, ..., l_{Bn}]$ e $[l_{A1}, l_{A2}, ..., l_{An}]$ conter a chave de **A** e $[l_{B1}, l_{B2}, ..., l_{Bn}]$ conter a chave de **B**, então existe uma ACE de equivalência entre **A** e **B** dada por **A** $\equiv$ **B**.

Exemplo P4.1: Considere os esquemas $\mathbf{S_1}$ e $\mathbf{S_2}$ da Figura 8. A existência de uma instância na classe de associação **MATRÍCULA** associando uma disciplina **d** e um professor **p**, requer a existência de uma instância na classe de associação **OFERTA** associando **d** e **p**. Essa restrição é capturada pela assertiva de DE de subconjunto `MATRÍCULA[disciplina, professor]` $\subset$ `OFERTA[disc, prof]`.

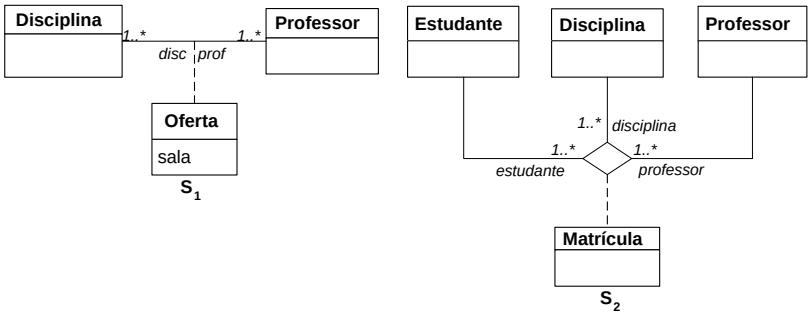


Figura 8: Exemplo - DE de Subconjunto entre Classes de Associação

Exemplo P4.2: Suponha os esquemas $\mathbf{S_3}$ e $\mathbf{S_4}$ da Figura 9. As classes de associação **ALOCAÇÃO** e **MATRÍCULA** relacionam um mesmo conjunto de classes: as classes **SEMESTRE**, **DISCIPLINA** e **PROFESSOR**.

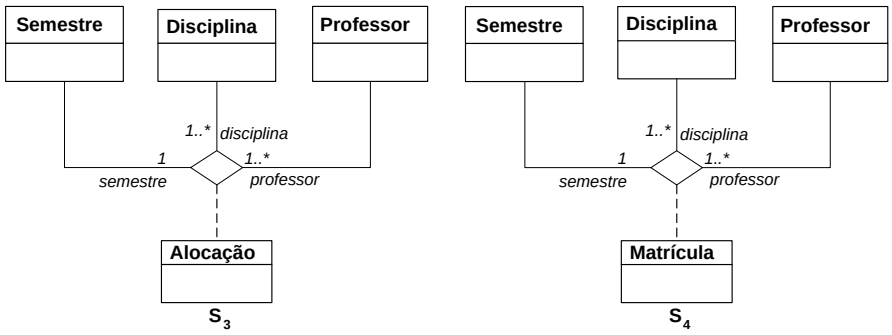


Figura 9: Exemplo - DE de Equivalência entre Classes de Associação

A existência de uma instância na classe de associação **ALOCAÇÃO** associando um semestre **s**, uma disciplina **d** e um professor **p**, requer a existência de uma instância na classe de associação **MATRÍCULA** associando **s**, **d** e **p** e vice-versa. Essa restrição é formalmente capturada pela assertiva de DE de equivalência `ALOCAÇÃO[semestre, disciplina, professor]` $\equiv$ `MATRÍCULA[semestre, disciplina, professor]`.

Usos Conhecidos: A identificação de restrições de dependência existencial a partir da análise dos esquemas conceituais pode ser encontrada em Vidal[14].

Padrões Relacionados: Certas formas de restrições de dependência existencial capturam redundânciacs no esquema conceitual. O padrões **P13** e **P14** reestruturam os esquemas de modo a remover essas redundâncias.

P7. Identificando Associações Derivadas

O padrão **P7** é usado durante a etapa de combinação II e ajuda a identificar associações derivadas no esquema conceitual. Uma associação derivada (ou redundante) é aquela cujas instâncias podem ser inferidas a partir de instâncias de outras associações.

Contexto: Suponha o esquema **S** da figura 10.

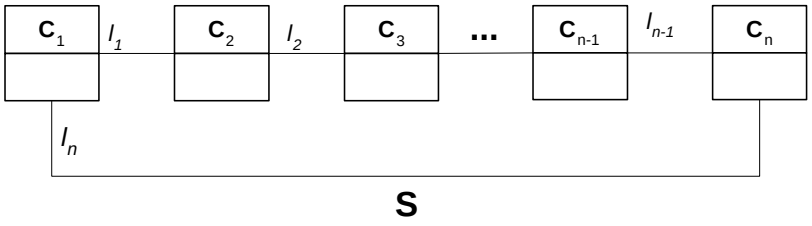


Figura 10: Associação Derivada

Existem dois caminhos ligando as classes $\mathbf{C_1}$ e $\mathbf{C_n}$: o caminho l_1 e o caminho $l_1 \bullet ... \bullet l_{n-1}$. Identificamos, portanto, a presença de um ciclo. Existe um ciclo em um esquema quando existe mais de um caminho ligando duas classes no esquema conceitual.

Problema: Verificar se a ligação l_n é derivável da composição das ligações $l_1 \bullet ... \bullet l_{n-1}$.

Solução: Um ciclo no esquema conceitual sugere a presença de uma associação derivada. Ao detectar a existência de um ciclo, é necessário verificar a compatibilidade das multiplicidades que compõem esse ciclo. As multiplicidades de dois caminhos são compatíveis se os valores mínimo e máximo das multiplicidades desses caminhos são iguais. Além disso, a semântica deve ser analisada. Pela análise da semântica associada ao esquema **S**, temos que l_n é derivada da composição dos caminhos $l_1 \bullet ... \bullet l_{n-1}$ se para qualquer instância $\mathbf{c_1}$ de $\mathbf{C_1}$, então $\mathbf{c_1}.l_n = \mathbf{c_1}.(l_1 \bullet ... \bullet l_{n-1})$. Nesse caso, existe uma ACC de equivalência dada por $l_n \equiv (l_1 \bullet ... \bullet l_{n-1})$.

Exemplo P7.1: Considere o esquema $\mathbf{S_1}$ mostrado na figura 11. Existem dois caminhos ligando as classes **PEÇA** e **FORNECEDOR**: o caminho *é-fornecida* e o caminho *está-contida* • *é-preenchido*. Identificamos, portanto, a presença de um ciclo.

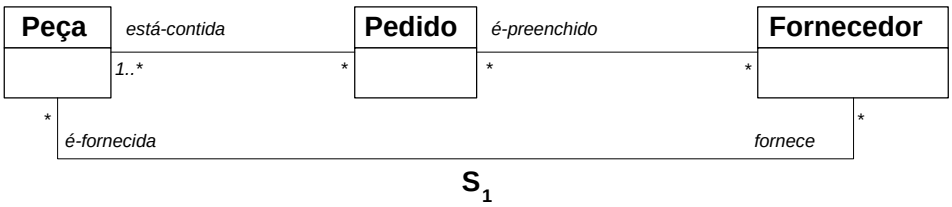


Figura 11: Exemplo - Associação derivada

As multiplicidades do ciclo da figura 11 são compatíveis, pois uma peça pode estar associada a zero ou mais fornecedores pelo caminho *é-fornecida* e pode estar associada a zero ou mais fornecedores pelo caminho *está-contida* • *é-preenchido*.

É necessário agora verificarmos se os caminhos representam o mesmo fato do mundo real. No contexto do exemplo mostrado na figura 11, temos que o caminho *é-fornecida* é derivável da composição dos caminhos *está-contida* e *é-preenchido*, pois para qualquer instância **p** de **PEÇA**, então $\mathbf{p}.$*é-fornecida* $= \mathbf{p}.$(*está-contida* • *é-preenchido*). Esta restrição é formalmente capturada pela ACC de equivalência `é-fornecida ≡ está-contida • é-preenchido`.

Usos Conhecidos: Uma discussão relacionada à existência de derivação em esquemas conceituais pode ser encontrada em Fowler[6].

Padrões Relacionados: No padrão **c** mostramos como remover associações derivadas do esquema conceitual.

P11. Eliminando Redundâncias Capturadas por ACEs de Subconjunto

O padrão **P11** é utilizado durante a fase de otimização I e propõe uma reestruturação do esquema conceitual de forma a eliminar redundâncias capturadas por ACEs de subconjunto.

Contexto: Suponha o esquema **S** mostrado na Figura 12.

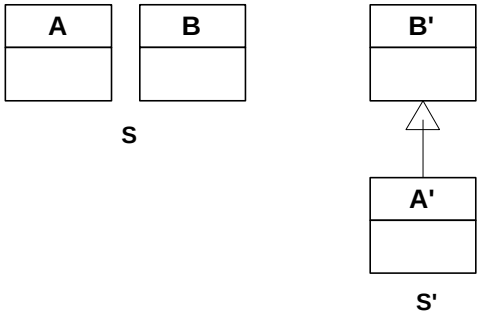


Figura 12: Eliminando AC de Subconjunto entre Extensões de Classes

Sejam **A** e **B** classes, `At(A)` o conjunto de atributos da classe **A**, `At(B)` o conjunto de atributos da classe **B**, `Lg(A)` o conjunto de ligações da classe **A** e `Lg(B)` o conjunto de ligações da classe **B**. Existe uma ACE de subconjunto entre **A** e **B**, dada por `A` $\subset$ `B`.

Problema: Remover do esquema conceitual a redundância capturada pela ACE de subconjunto `A` $\subset$ `B`.

Solução: O esquema **S** deve ser transformado no esquema **S'** como mostrado na Figura 12. No esquema **S'**, a classe **B'** é equivalente à classe **B** e a classe **A'** é definida como uma subclasse da classe **B'**. Os atributos e ligações de **A'** são distribuídos como segue:

- `At(A') = At(A) - At(B)`, onde - (diferença de conjuntos) significa que os atributos da classe **A'** são aqueles atributos de **A** que não possuem atributos semanticamente equivalentes em **B**.
- `Lg(A') = Lg(A) - Lg(B)`, onde - (diferença de conjuntos) significa que as ligações da classe **A'** são aquelas ligações de **A** que não possuem ligações semanticamente equivalentes em **B**.

O mapeamento entre o esquema original **S** e o esquema transformado **S'** é formalmente especificado pelas ACs abaixo:

(i) `A` $\equiv$ `A'`

(ii) `B` $\equiv$ `B'`

(iii) os atributos de **B** são mapeados diretamente nos atributos de **B'**

(iv) os atributos de **A** que não possuem atributos semanticamente equivalentes em **B** são mapeados diretamente em atributos de **A'**

Exemplo P11.1: Suponha o esquema $\mathbf{S_1}$ mostrado no exemplo **P2.1**.

De acordo com a AC (i), verificamos uma generalização implícita entre as classes **GERENTE** e **EMPREGADO** que não está devidamente representada no esquema $\mathbf{S_1}$. Nesse caso, propomos reestruturar o esquema $\mathbf{S_1}$ como sugerido no esquema $\mathbf{S_2}$ da Figura 13. No esquema $\mathbf{S_2}$, definimos duas novas classes **EMPREGADO'** e **GERENTE'** e definimos uma generalização entre essas classes. Os atributos e ligações da nova classe **EMPREGADO'** correspondem aos atributos e ligações da classe **EMPREGADO**. Os atributos da classe **GERENTE'** correspondem aos atributos da classe **GERENTE** que não possuem atributos semanticamente equivalentes em **EMPREGADO**. As ligações de **GERENTE'** correspondem às ligações de **GERENTE** que não possuem ligações semanticamente equivalentes em **EMPREGADO**.

O mapeamento entre o esquema original $\mathbf{S_1}$ e o esquema transformado $\mathbf{S_2}$ é formalmente especificado pelas assertivas de correspondência abaixo:

Assertivas de Correspondência de Extensão:

(i) `EMPREGADO` $\equiv$ `EMPREGADO'`

(ii) `GERENTE` $\equiv$ `GERENTE'`

Assertivas de Correspondência de Caminho:

(iii) `GERENTE.cpf` $\equiv$ `EMPREGADO'.cpf`

(iv) `GERENTE.nome` $\equiv$ `EMPREGADO'.nome`

(v) `GERENTE.salário` $\equiv$ `EMPREGADO'.salário`

(vi) `GERENTE.data-cargo` $\equiv$ `GERENTE'.data-cargo`

(vii) `GERENTE.`*pertence* $\equiv$ `EMPREGADO'.`*depto*

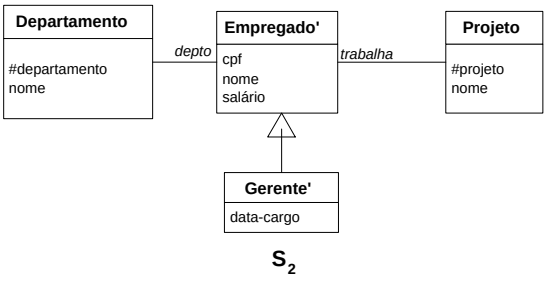


Figura 13: Exemplo - Eliminando AC de Subconjunto entre Extensões de Classes

Usos Conhecidos: Uma discussão relacionada a reestruturação de esquemas a partir das ACEs pode ser encontrada em Vidal[14].

Padrões Relacionados: No padrão **P2** mostramos como identificar ACEs de subconjunto.

P15. Removendo Associações Derivadas

O padrão **P15** é utilizado durante a fase de otimização II e propõe uma reestruturação do esquema conceitual de forma a eliminar redundâncias capturadas pela presença de associações derivadas no esquema conceitual.

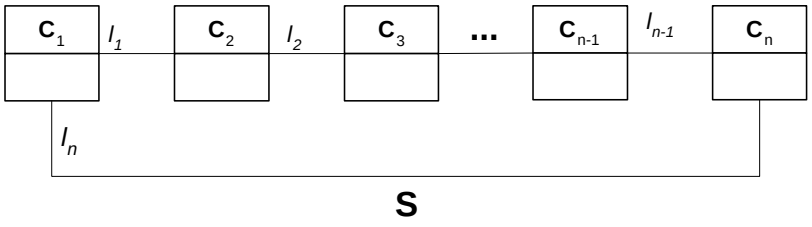


Figura 14: Associação Derivada

Contexto: Suponha o esquema **S** da figura 14.

Pela análise da semântica de **S**, temos que l_n é derivada da composição dos caminhos $l_1 \bullet ... \bullet l_{n-1}$, isto é, $l_n \equiv l_1 \bullet ... \bullet l_{n-1}$.

Problema: Reestruturar o esquema conceitual de modo a remover a associação derivada l_n.

Solução: O esquema **S** deve ser transformado no esquema **S'** como mostrado na figura 15.

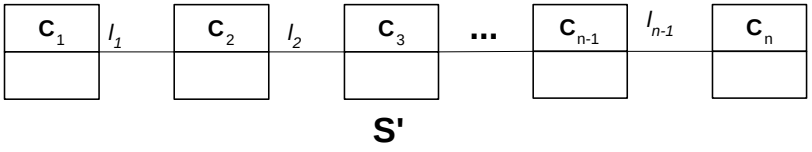


Figura 15: Removendo Associação Derivada

O mapeamento entre o esquema original **S** e o esquema transformado **S'** é formalmente especificado pelas assertivas de correspondência abaixo.

(i) $C_i \equiv C'_i$, $\texttt{1} \leq \texttt{i} \leq \texttt{n}$

(ii) $C_1.l_n \equiv C'_1.(l_1 \bullet ... \bullet l_{n-1})$

(iii) $C_i.l_i \equiv C'_i.l_i$, $\texttt{1} \leq \texttt{i} \leq \texttt{n}$

Exemplo P15.1: Considere o esquema $\mathbf{S_1}$ do exemplo **P7.1**. De forma a remover a redundância identificada, sugerimos reestruturar esse esquema como mostrado no esquema $\mathbf{S_2}$ da figura 16. No esquema $\mathbf{S_2}$, removemos a associação *é-fornecida*, já que esta pode ser obtida a partir da composição das associações *está-contida* e *é-preenchido*. Vale a pena ressaltar que o projetista pode decidir permanecer com a associação derivada no esquema durante a implementação do banco de dados por motivos relacionados ao desempenho das operações a serem executadas sobre o banco de dados.

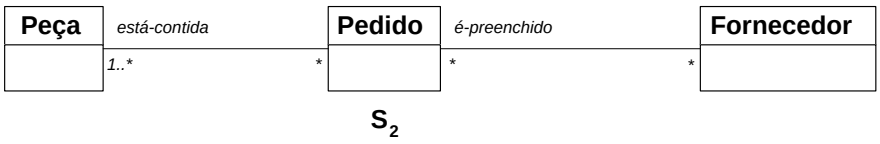


Figura 16: Exemplo - Removendo Associação Derivada

O mapeamento entre o esquema original $\mathbf{S_1}$ e o esquema transformado $\mathbf{S_2}$ é formalmente especificado pelas assertivas de correspondência abaixo:

Assertivas de Correspondência de Extensão:

(i) `PEÇA` $\equiv$ `PEÇA'`

(ii) `PEDIDO` $\equiv$ `PEDIDO'`

(iii) `FORNECEDOR` $\equiv$ `FORNECEDOR'`

Assertivas de Correspondência de Caminho:

(iv) `PEÇA`.*é-fornecida* $\equiv$ `PEÇA'.(`*está-contida* $\bullet$ *é-preenchida*)

(v) `PEÇA`.*está-contida* $\equiv$ `PEÇA'`.*está-contida*

(vi) `PEDIDO`.*é-preenchido* $\equiv$ `PEDIDO'`.*é-preenchido*

Usos Conhecidos: Fowler[6] aborda o problema de associações derivadas no esquema conceitual.

Padrões Relacionados: No padrão **P7** discutimos como identificar a presença de associações derivadas no esquema conceitual.

P18. Eliminando Redundâncias Capturadas por DEs de Subconjunto entre Classes de Associação

O padrão **P18** é utilizado durante a fase de otimização II e propõe uma reestruturação do esquema conceitual de forma a eliminar redundâncias capturadas por DEs de subconjunto entre classes de associação.

Contexto: Suponha o esquema **S** mostrado na Figura 17.

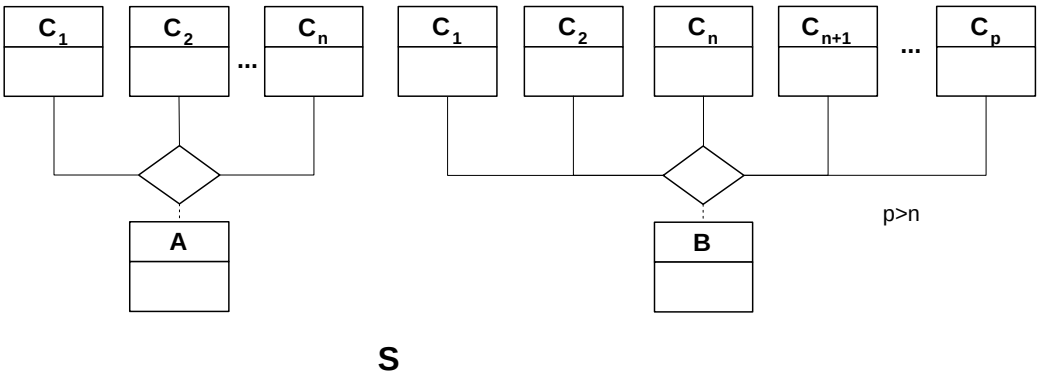


Figura 17: DE de Subconjunto entre Classes de Associação

Sejam **A** e **B** classes de associação, `At(A)` o conjunto de atributos de **A**, `At(B)` o conjunto de atributos de **B**, `Lg(A)` o conjunto de ligações de **A** e `Lg(B)` o conjunto de ligações de **B**. Existe uma restrição de DE entre **A** e **B** dada por texttt**A** $[l_{A1}, l_{A2}, ..., l_{An}] \subset$ **B** $[l_{B1}, l_{B2}, ..., l_{Bp}]$.

Problema: Remover do esquema conceitual a redundância causada pela restrição de DE de subconjunto existente entre as classes de associação **A** e **B** dada por **A** $[l_{A1}, l_{A2}, ..., l_{An}] \subset$ **B** $[l_{B1}, l_{B2}, ..., l_{Bp}]$.

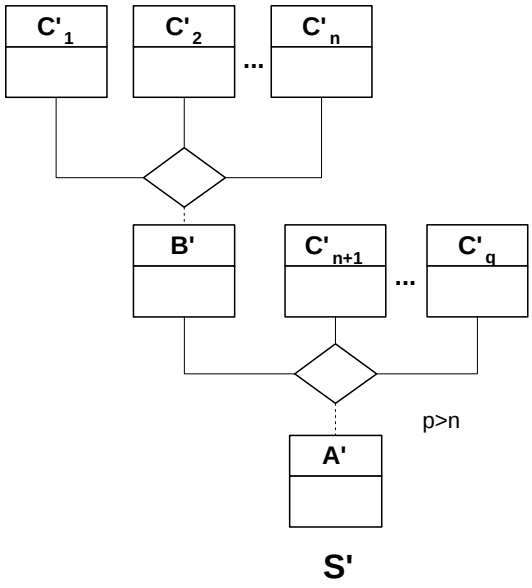


Figura 18: Removendo DE de Subconjunto entre Classes de Associação

Solução: O esquema **S** deve ser transformado no esquema **S'** como mostrado na Figura 18. No esquema **S'**, a classe **B'** é equivalente à classe **B** e a classe **A'** é modelada como uma associação entre as classes **B'**,$\mathbf{C'_{n+1}}$ e $\mathbf{C'_p}$.

Os atributos e ligações da nova classe de associação **A'** são distribuídos como segue:

- `At(A') = At(A)`

- `Lg(A') = Lg(A) - Lg(B)`, onde - (diferença de conjuntos) significa que as ligações da classe **A'** são aquelas ligações de **A** que não possuem ligações semanticamente equivalentes em **B**.

O mapeamento entre o esquema original **S** e o esquema transformado **S'** é formalmente especificado pelas assertivas de correspondência abaixo:

(i) `A` $\equiv$ `A'`

(ii) `B` $\equiv$ `B'`

(iii) $C_i \equiv C'_i$, `1` $\leq$ `i` $\leq$ `n`

(iii) os atributos de **A** são mapeados diretamente em atributos de **A'**

(iv) os atributos de **B** são mapeados diretamente em atributos de **B'**

Exemplo P18.1: Considere os esquemas $\mathbf{S_1}$ e $\mathbf{S_2}$ do exemplo **P4.1**.

Sugerimos reestruturar os esquemas $\mathbf{S_1}$ e $\mathbf{S_2}$ como mostrado no esquema $\mathbf{S_3}$ da Figura 19. No esquema integrado $\mathbf{S_3}$, definimos as classes de associação **MATRÍCULA'** e **OFERTA'**. Os atributos e ligações da classe de associação **OFERTA'** correspondem aos atributos e ligações da classe de associação **OFERTA**. Os atributos da classe de associação **MATRÍCULA'** correspondem aos atributos de **MATRÍCULA**. As ligações de **MATRÍCULA'** correspondem às ligações de **MATRÍCULA** que não possuem ligações semanticamente equivalentes em **OFERTA**.

O mapeamento entre o novo esquema $\mathbf{S_3}$ e os esquemas originais $\mathbf{S_1}$ e $\mathbf{S_2}$ é formalmente especificado pelas assertivas de correspondência abaixo:

Assertivas de Correspondência de Extensão:

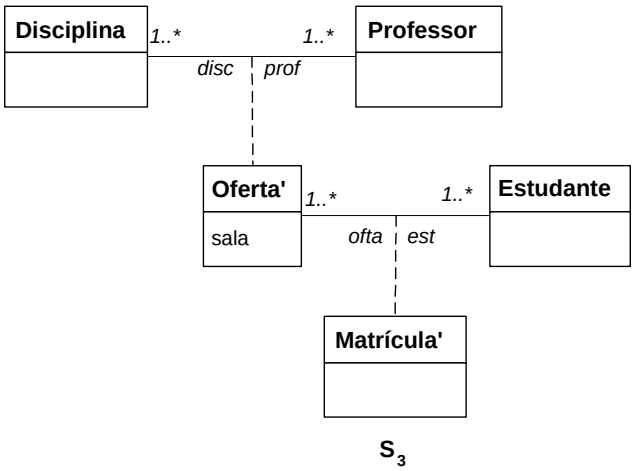


Figura 19: Exemplo - Representando DE de Subconjunto entre Classes de Associação

(i) `OFERTA` $\equiv$ `OFERTA'`

(ii) `MATRÍCULA` $\equiv$ `MATRÍCULA'`

Assertivas de Correspondência de Caminho:

(iii) `MATRÍCULA`.*estudante* $\equiv$ `MATRÍCULA'`.*est*

(iv) `MATRÍCULA`.*disciplina* $\equiv$ `MATRÍCULA'`.*ofta'*.*disc*

(v) `MATRÍCULA`.*professor* $\equiv$ `MATRÍCULA'`.*ofta'*.*prof*

Usos Conhecidos: Restrições de dependência existencial são abordadas em Vidal[14].

Padrões Relacionados: No padrão **P4** mostramos como identificar, a partir da análise da semântica do mundo real, as restrições de dependência existencial entre classes de associação.

6 Conclusões

Segundo Navathe[12], dois motivos justificam a necessidade de integração de visões durante o projeto de um banco de dados: (i) a estrutura de um banco de dados para grandes aplicações é bastante complexa de ser modelada por um único projetista em uma única visão; (ii) em geral, os grupos de usuários trabalham de forma independente nas organizações e possuem seus próprios requisitos dos dados, que podem conflitar com os interesses de outros grupos. Neste artigo, abordamos o problema da integração de visões modeladas com UML durante o projeto conceitual de um banco de dados.

O processo de integração proposto em nossa metodologia usa o catálogo de padrões definido em Marinho[8]. A idéia associada a esse catálogo é propor uma série de soluções individuais para problemas de integração que, em conjunto, mostram como construir um esquema conceitual global de forma correta.

Como trabalhos futuros, pretendemos incorporar o catálogo de padrões proposto a um tutorial para ensino de modelagem conceitual utilizando o diagrama de classes da UML. Esse tutorial possibilitará, através de exemplos, questionamentos e textos explicativos, o aprendizado de forma didática de importantes técnicas de modelagem conceitual, auxiliando os projetistas de banco de dados no desenvolvimento de projetos conceituais consistentes, mesmo em situações não facilmente capturadas por um esquema conceitual.

Agradecimentos: Agradecimento especial ao shepherd, Jerffeson Teixeira de Souza, pela cooperação e tempo dispensados na realização deste trabalho e ao Instituto Atlântico pelo apoio financeiro que viabilizou a apresentação deste artigo na SugarloafPLoP'02.

Referências

[1] Grady Booch, Ivar Jacobson, and James Rumbaugh. The UML specification document. Technical report, Rational Software Corporation, 1997. Disponível em www.rational.com.

[2] M. Casanova and Vânia Maria Ponte Vidal. Towards a sound view integration methodology. In *2nd ACM SIGACT/SIGMOD Conference on Principles of Database Systems*, 1983.

[3] E. Codd. Further normalization of the data base relational model. In *Data Base Systems*, 1972.

[4] Deb Dey, Veda Catherine Storey, and Terence M. Barron. Improving database design through the analysis of relatioships. Draft Only, Maio 1997.

[5] Martin Fowler. *Analysis Patterns. Reusable Object Models.* Object Technology Series. Addison-Wesley, 1997.

[6] Martin Fowler. *UML Distilled.* Object Technology Series. Addison-Wesley, 1997.

[7] T. W. Ling. A normal form for entity-relationship diagrams. In *International Conference on the Entity-Relationship Approach*, 1985.

[8] Fabiana Gomes Marinho. Padrões para integração de visões modeladas com uml. Master's thesis, Universidade Federal do Ceará, 2001.

[9] R.J. Miller, Y. E. Ioannidis, and R. Ramakrishman. The use of information capacity in schema integration and translation. In *19th International Conference on Very Large Databases*, 1993.

[10] S.B. Navathe and S.G. Gadgil. A methodology for view integration in logical database design. In *8th International Conference on Very Large Data Bases*, 1982.

[11] Michael Schrefl. A comparative analysis of view integration methodologies. In *GI-Fachtagung EMISA*, 1987.

[12] R. Shamkant Navathe, Elmasri and J. Larson. Integrating user views in database design. *Data Engineering*, 1986.

[13] Veda C. Storey. Relational database design based on the entity-relationship model. *Data e Knowledge Engineering*, 1991.

[14] Vânia Maria Ponte Vidal. *Preservando a Semântica de Atualizações na Integração de Visões.* PhD thesis, Universidade Federal do Rio de Janeiro, 1994.

Special Session on Software Pattern Applications

SPA is dedicated to explore applications that involve software patterns. It provides a forum for researchers and practitioners in the area to meet and exchange research ideas and results.

We want to spread the use of patterns in Latin America, stimulating not only new patterns to be written, but also disseminating the culture of patterns among our software developers. This can be obtained if we show that patterns are a powerful reuse technique that has evolved in the last decade and is being used more and more in concrete projects.

The SPA Session Dynamics

SPA sessions were about 30 minutes each, with 25 minutes for authors' presentation and 5 minutes for questions.

A tool and a formalism to design and apply patterns[1]

Agnès Conte[1], José-Celso Freire Junior[1,2], Jean-Pierre Giraudin[1],
Ibtissem Hassine[1], Dominique Rieu[1]

[1]LSR-IMAG, SIGMA
BP 72, 38402 SAINT MARTIN D'HERES CEDEX – France

[2]UNESP/FEG/DEE
12500-000 - CP 205 - Guaratingueta/SP Brazil B

E-mail : Agnes.Conte@imag.fr, Jose-Celso.Freire@imag.fr,
Jean-Pierre.Giraudin@imag.fr, Ibtissem.Hassine@imag.fr,
Dominique.Rieu@imag.fr

Abstract

Pattern systems are becoming more and more numerous. They offer product patterns or process patterns of varied range and cover (analysis, design or implementation patterns, and general, domain or enterprise patterns). New application development environments have been developed together with these pattern-oriented approaches. These tools address two kinds of actors: patterns engineers who specify pattern systems, and applications engineers who use these systems to specify information systems. Most of the existing development environments are made for applications engineers; they offer few functionalities allowing definition and organization of pattern systems. This paper presents AGAP, a development environment for defining and using patterns, which distinguishes pattern formalisms from pattern systems. Not only does AGAP address applications engineers, but it also allows patterns engineers to define pattern systems. The same formalisms or items of existing formalisms may either be used in order to facilitate the engineering of pattern systems or to increase the level of reuse. We illustrate the use of AGAP by the presentation of P-Sigma, a common formalism for pattern representation. P-Sigma expresses a semantics common to most of the existing formalisms and standardizes the expression of product patterns and process patterns. It allows to clarify the patterns selection interface and facilitates the organization of pattern systems. Two pattern systems developed during industrial experimentation validate the P-Sigma formalism and were implemented in AGAP.

Keywords: pattern, pattern system, reuse, product pattern, process pattern, pattern formalism, pattern-based development environment.

1 Introduction

A wide variety of reusable component models have already been proposed to integrate reuse in all applications development processes: business objects [20], generic domain models [16], analysis patterns [8] [9], design patterns [11], frameworks [15], etc. In all cases, a component

is considered as a tested and accepted solution to a problem which occurs frequently in information systems development.
In this paper, particular interest is given to the pattern approach [1]. Different criteria may be used to compare component models [10] [7]. Some of them are particularly interesting to characterize patterns.

- ***Type of knowledge***. A pattern capitalizes ***products*** – a product corresponds to a goal to reach - or capitalizes ***processes*** - a process corresponds to a way to reach a result.
- ***Coverage***. A pattern coverage may be ***generic*** (resp. ***domain, enterprise***) if it solves a problem frequently occurring in several domains (resp. in an application domain, in a particular enterprise).
- ***Range***. The pattern range may be ***analysis***, ***design*** or ***implementation*** depending on the stage of the engineering process it addresses.

The best known pattern systems (P. Coad [8], E. Gamma [11], etc.) propose ***general product patterns*** dedicated to only one stage of the development process (***analysis*** for P. Coad's patterns, ***design*** for E. Gamma's patterns). S.W. Ambler's patterns [2] are ***general process patterns*** covering all the stages of the engineering process. They provide collections of techniques, actions and/or tasks for software development. Finally, the pattern system proposed by L. Gzara [13] concerns a specific ***domain*** (the Product Information Systems[2] – PIS- or the Product Data Management). Patterns cover the stages of ***analysis*** and ***design*** of the PIS development and integrate ***product patterns*** and ***process patterns***. In this pattern system, process patterns aim to specify a PIS development process which makes easier the use of product patterns (selection, application, composition, etc.).
Whatever their coverage, range or type may be, patterns have to be integrated in development environments. Pattern-based tools are dedicated to two kinds of actors:

- The applications engineer's goal is to apply patterns and to combine these patterns applications in order to model an information system.
- The patterns engineer's goal is to define new patterns by identifying recurrent problems and solutions to these problems.

The majority of existing tools only integrates E. Gamma's pattern system [GAM 95] and don't take into account the patterns engineers needs ([4] [19] [17] [18] [21] [6] [12]). Section 2 presents AGAP, a development environment, which combines the needs of the applications engineers and the needs of patterns engineers. Section 3 introduces the P-Sigma formalism, a pattern representation formalism which intends to resolve some limits of the existing pattern formalisms. P-Sigma aims to express a common semantics for the majority of existing formalism, to standardize the expression of product patterns [8] [11] [9] and process patterns [2] [GZA13R00], to make explicit the patterns selection interface to facilitate their reuse, and finally to propose patterns relationships to better organize pattern systems and to increase their reuse. To conclude, section 4 presents two industrial experiences using AGAP and P-Sigma and proposes future prospects of our research.

[2] Product Information Systems (PIS) support all types of engineering data used to define, manufacture and support products. They may include definitions, specifications, CAD drawings, manufacturing process plans and routings, project plans, control records, etc.

2 AGAP

2.1. Actors

AGAP (in french « Atelier de Gestion et d'Application de Patrons ») offers solutions to combine the various needs of two kinds of actors: applications engineer and patterns engineer.

2.1.1. Patterns engineer

The patterns engineer's goal is to define pattern formalisms and pattern systems described according to these formalisms. Several use cases are allocated to him (see Figure 1): for example, a patterns engineer must be able to create, modify, validate and visualize a pattern formalism or a pattern system. The validation of a pattern formalism (respectively a pattern system) implies that it is not modifiable, but allows it to be used to create pattern systems (respectively information systems). The patterns engineer is also authorized to manipulate applications domains and targeted technologies: a pattern system is applied to a given domain (for example banking IS) according to a given technology (for example, relational or object-oriented technology). Finally, AGAP allows a classification of items: the patterns engineer can define, modify and visualize the types of the fields (text, UML diagrams, etc.).

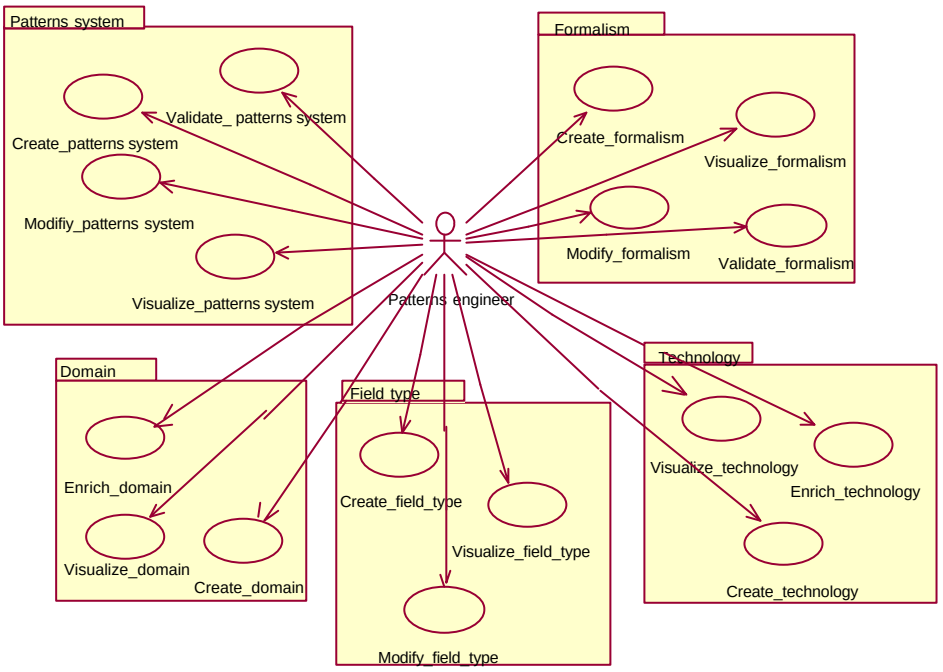


Figure 1: Patterns engineer's use cases

2.1.2. Applications engineer

The goal of the applications engineer is to apply patterns in order to model and design information systems. This application is based on one or more pattern systems available in AGAP. The main needs of the applications engineer can be summarized by UML use cases (see Figure 2): an applications engineer must be able to create or modify an information system, to visualize a pattern system and to visualize an information system trace. An information system trace shows the patterns selection and patterns application processes by

preserving the patterns applications, the patterns from which they result, and the successive integrations of patterns applications.

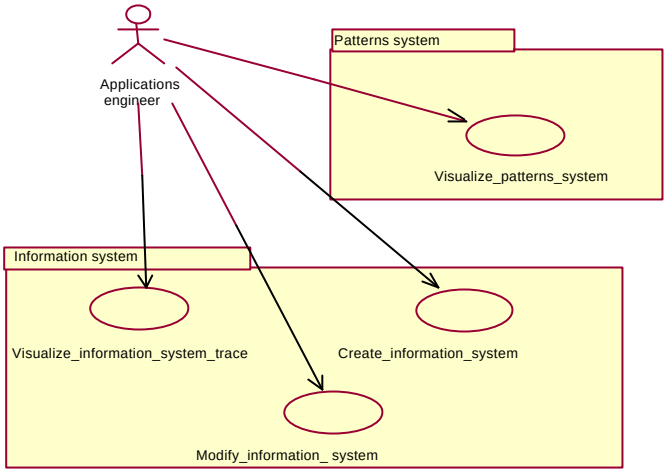


Figure 2: Applications engineer's use cases

2.2. Components

AGAP is composed of 7 business components: Tool, Information System, Pattern system, Domain, Technology, Formalism and Field-type. Only two components will be described here (Formalism and Pattern system).

The structure of each component conforms the business components structuring of the Symphony process [14]. This structuring is inspired by CRC (Class-Responsibility-Collaboration) [24]. According to this method, a business component is modeled by a package composed of three parts (see Figure 3): an interface part (*what I can do*), a structural part (*what I am*) and a collaboration part (*what I use*). The three parts of a component are represented by four object types stereotyped by:

- **Master object**: it is the main object of the component for which the services are carried out. It is identifiable by any external actor.
- **Part object**: the part-object is complementary to the master object. It is identified by its attachment with the master object to which it is linked by a relation of composite type.
- **Role object**: it is an object servant. All the services of the other components are called through this object.
- **Interface**: it represents the services contract of the component. It supports the operations of the component responsibility.

2.2.1. Component « Formalism »

A formalism (see Figure 3) contains one or more items which can be shared with other formalisms. Some of them are mandatory, and others are optional. Three types of items exist: *Interface item* in order to facilitate patterns selection, *Realization item* in order to express patterns solution and *Relations item* in order to organize pattern systems. Each item is composed of one or more fields which may be optional.

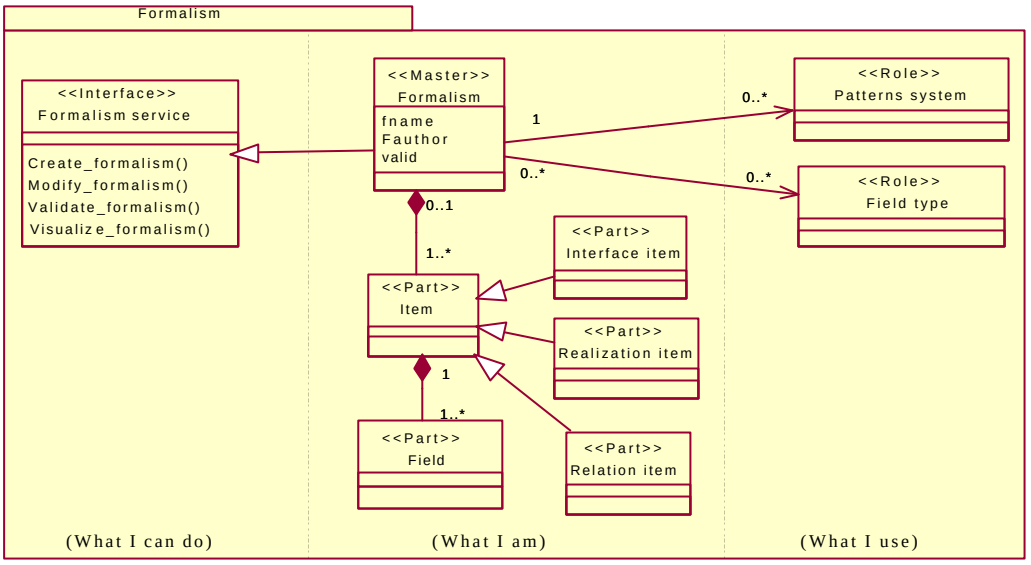


Figure 3: Component «Formalism»

2.2.2. Component « Pattern system »

A pattern system (see Figure 4) is composed of patterns. Its representation is described in a given formalism. Each pattern has a given number of items whose fields are defined in the associated formalism. A domain (banking IS, geographic IS, etc.) and a technology (object-oriented, relational, etc.) are associated to each pattern system.

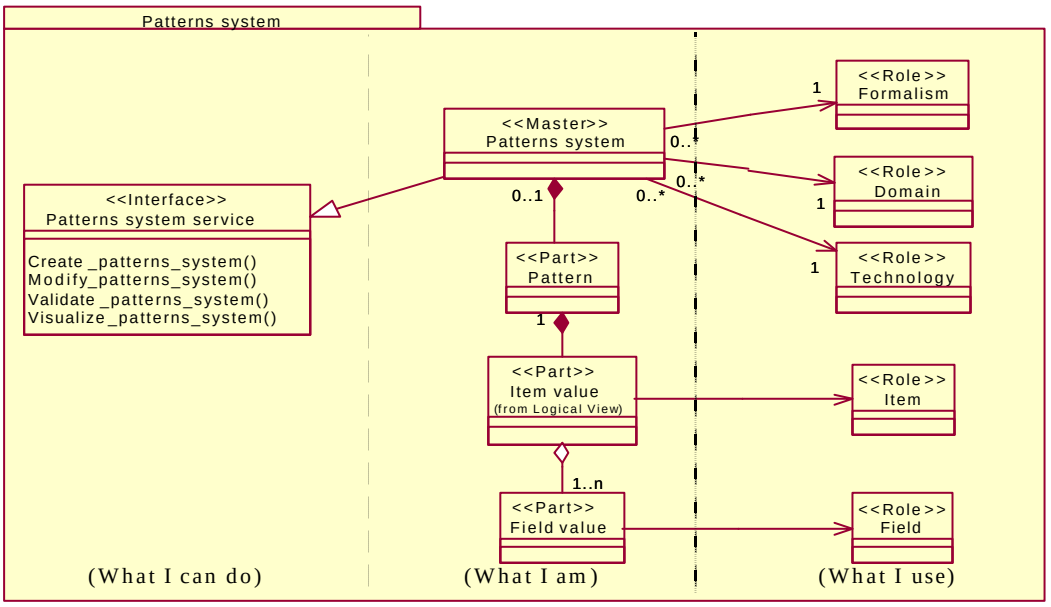


Figure 4: Component « Pattern system »

We describe the use case « Visualize_patterns_ system » with a scenario (see Figure 5).

Scenario: ***Visualize_ patterns_ system***

1- Search for the pattern systems
2- Choose a pattern system
3- Display information (name, formalism, domain, technology, etc.) of the chosen pattern system
4- Optionally visualize patterns and the relationships between them
5- Optionally select a pattern and visualize its information (items values).
6- Terminate the scenario or go to 5.

Figure 5: A scenario of the use case « Visualize_ patterns_system »

Figure 6 displays the screen proposed to the user to visualize a pattern system. The formalism, the domain and the technology associated to a pattern system are showed (for example in Figure 6, for the Symphony pattern system).

The button « See » allows the user to reach all the patterns composing this pattern system: he can then visualize the relationships between the patterns of the chosen pattern system (see Figure 7). The pattern system is represented by a graph whose nodes symbolize the different patterns of the system and whose arcs represent the relationships between patterns. The user can then select a pattern and get its information. Finally, when the user clicks on a pattern, he reaches the information on the pattern itself (see Figure 8).

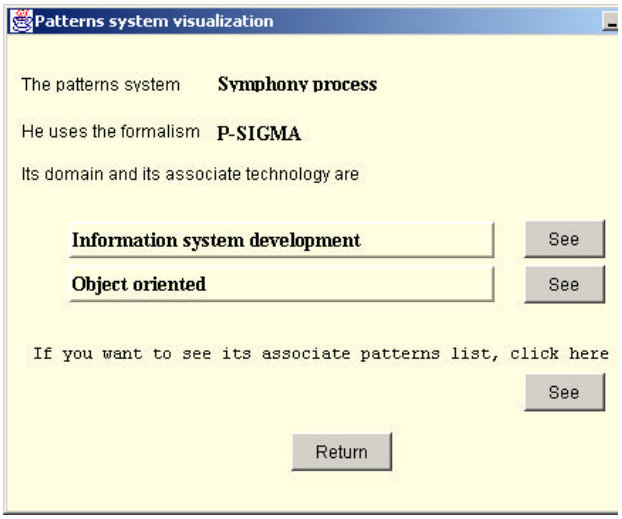


Figure 6: Visualizing a pattern system

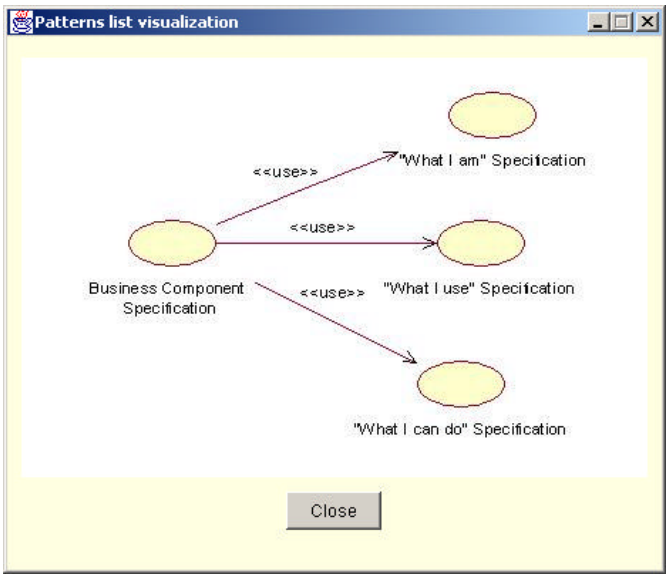


Figure 7: Visualizing the relationships between the patterns of a pattern system

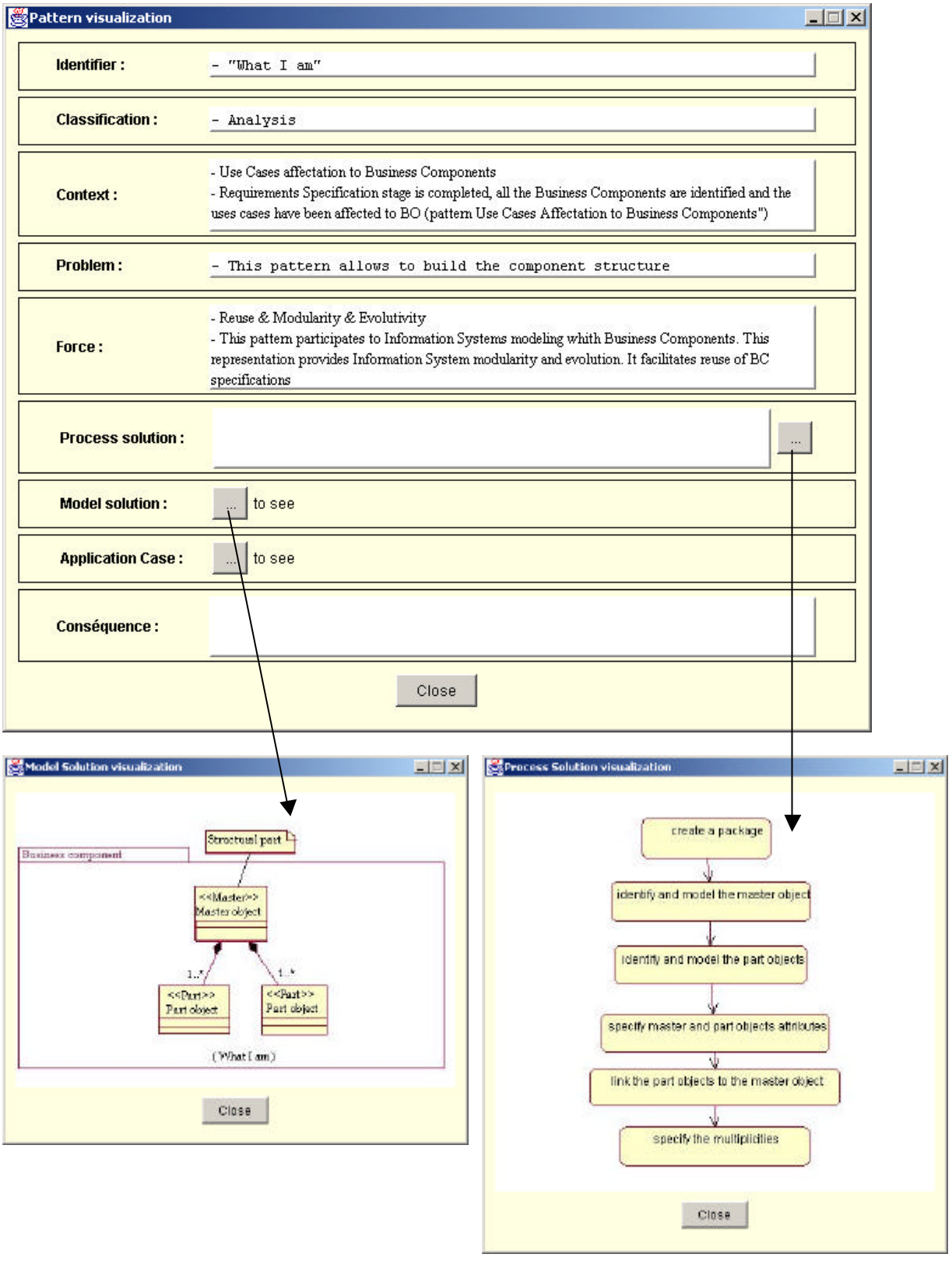


Figure 8 : Visualizing the information on a pattern in AGAP

2.3. Meta-Model and implementation

Building on the seven business components described above, we propose a model containing the essential information on the patterns managed by AGAP. This model is considered as the meta-model of AGAP (see Figure 9).

AGAP is implemented in Java with the RAD JBuilder[3]. Human interfaces are designed using the graphical library Swing. XML files are manipulated by the IBM parser. The communication between AGAP and Rational Rose is done by Microsoft COM technology.

The meta-model of AGAP is thereafter instantiated to produce pattern systems description formalisms and pattern systems. We present in the next section the formalism P-Sigma, a common pattern representation formalism, and an instantiation of the meta-model of AGAP.

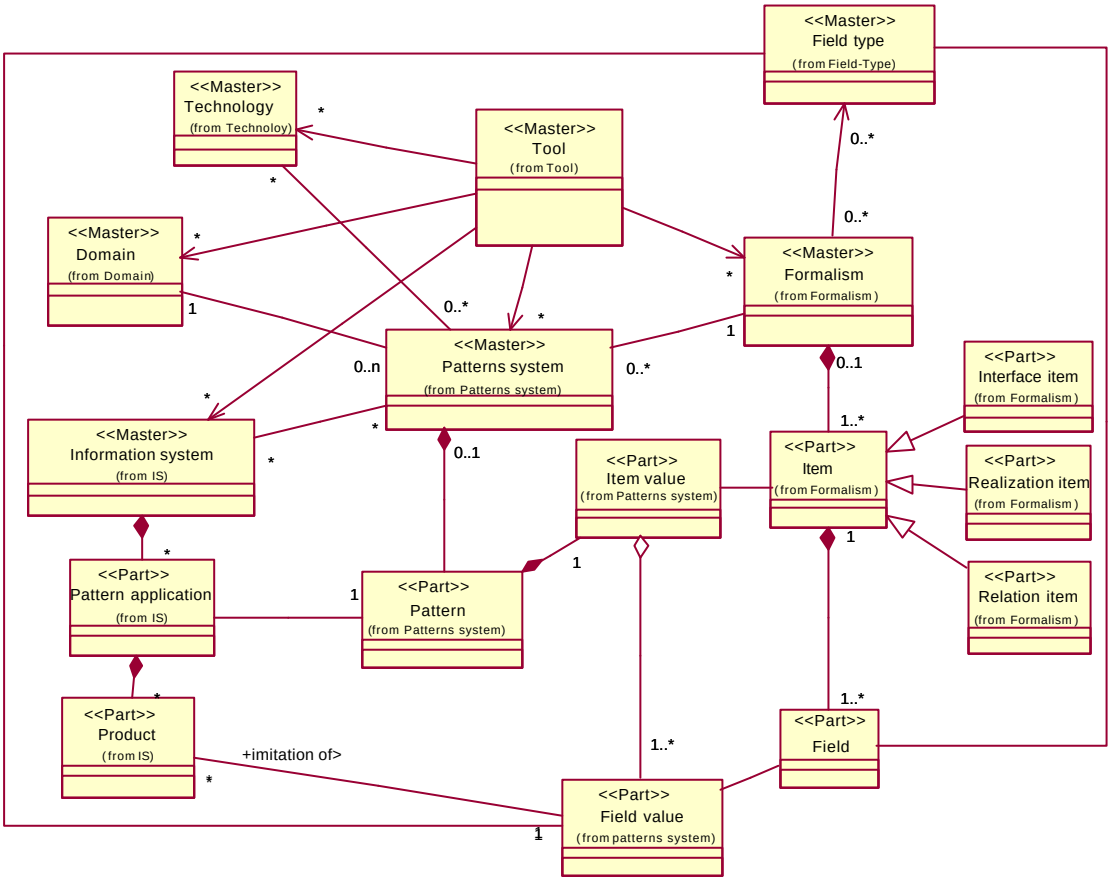


Figure 9: AGAP's meta-model

3 The formalism P-Sigma

3.1. Goals

P-Sigma main objectives are:

- **Standardization of product and process pattern representation.**

[3] http://www.borland.fr/produits/jbuilder/

A single pattern system must integrate product and process patterns and therefore must offer a unique formalism to facilitate the combined expression of model and process solutions. A pattern expressed in P-Sigma can then:

- ✓ Offer only a Model Solution. It is the case when this model adaptation does not require any specific methodological assistance. The pattern's application is then obtained by cloning and adapting the proposed model.
- ✓ Offer only a Process Solution. It is the case of patterns whose objective is to decompose a process into elementary fragments, also described by patterns. The pattern's application consists in performing the proposed process.
- ✓ Offer a model (Model Solution) as a solution to the given problem as well as the way to obtain such a model (Process Solution). The Process Solution consists then in a methodological guide assisting the designer to adapt the model. The pattern's application consists in performing the process proposed by the Process Solution in conformity with the model proposed by the Model Solution.

Globally, all patterns could be expressed in a more homogeneous way, therefore facilitating its communication and its combination.

- **Better formalization of the pattern's selection interface**

Contrary to the existing representation formalisms, where the items allowing pattern's selection are not explicit, P-Sigma distinguishes five items helping to select patterns. These items are grouped in the formalism Interface part.

- **Pattern system organization**

Most pattern formalisms express all types of relationships in a unique item. As an example, the "Combination" item of P. Coad's formalism [8] gives the possible combinations of the pattern with other patterns. The "Related Patterns" item of E. Gamma's formalism [11] gives patterns using or used by the given pattern. S.W. Ambler's "linked patterns" item [2] includes patterns composing the studied pattern, patterns composed of the studied pattern and patterns associated with it. The semantics of these items is therefore not clearly defined. P-Sigma formalism aims to make explicit the different relations among patterns. The Relation part enables to organize a pattern system thanks to clear relations: uses, requires, alternative, refines, etc.

3.2. General Structure

P-Sigma is composed of three parts: Interface, Realization and Relation. Interface part contains all elements allowing pattern's selection. Realization part gives the solution in terms of Model Solution and Process Solution. Finally, Relation part organizes relationships between patterns.

Each part contains a certain number of items (see Figure 10). Each item is composed of one or several typed fields (text, UML diagram, keywords logical expression, etc.). Figure 10 underlines for each item the number and the type of its different fields. It also shows the mandatory items: Identification, Classification, Context, Problem and Solution (Process or Model).

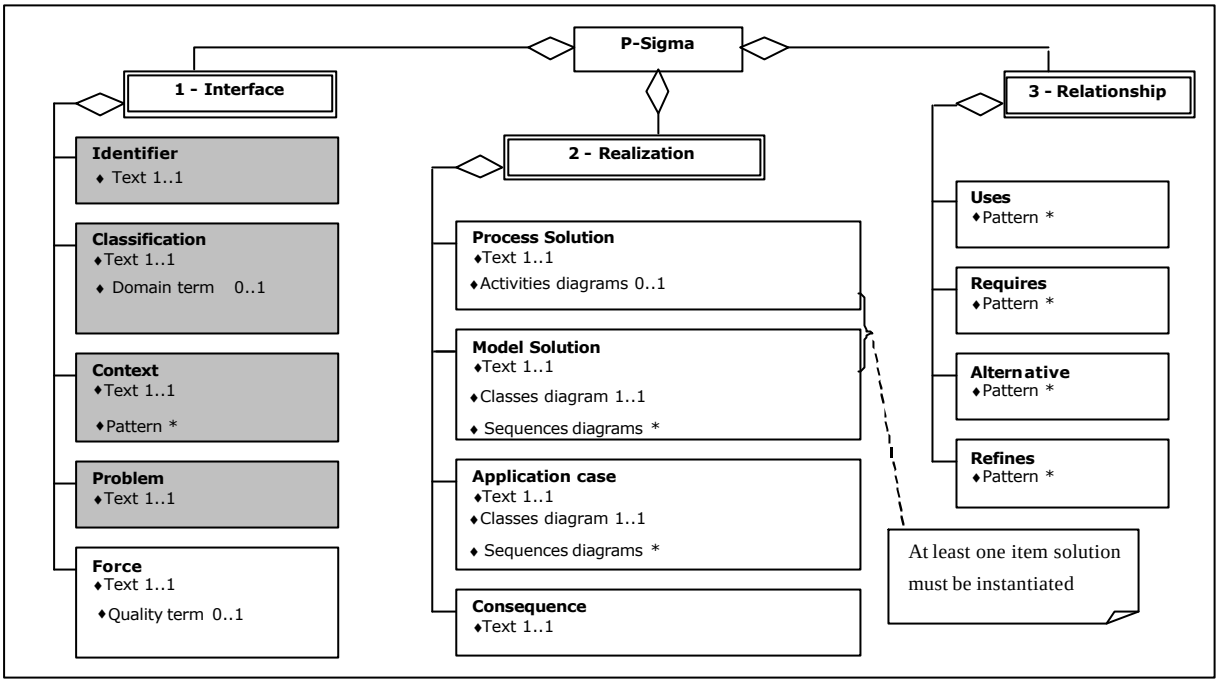


Figure 10: General Structure of P-Sigma formalism

Figure 8 partially illustrates the interface part and the solution part of P-Sigma by the pattern *"What I am" Specification* of the Symphony pattern system. In the following, *Relation* part is detailed.

3.3. Relation part

The Relation part is composed of four items corresponding to the four types of relationships between patterns: Uses, Refines, Requires and Alternative. In P-Sigma, each relation is expressed by an item giving the patterns linked to the pattern described (see Figure 10).The meaning of each relation is based mainly on the items of the Interface part.

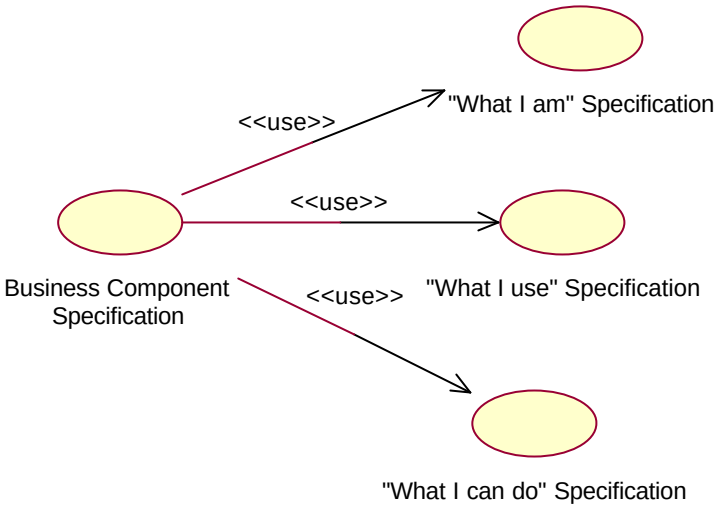
- **Uses:** If a pattern P1 uses a pattern P2, then:
  - ✓ P1's Process Solution must be expressed using P2.
  - ✓ P2's Classification may be enriched with respect to P1's one: new keywords may be added in P2's Classification.
  - ✓ P2's Context may be enriched with respect to P1's one.

P1 uses P2	P1	P2
Classification	M1	M1 ∧ M2
Context	C1	C1 ∧ C2
Process Solution	***Apply P2***	

Example*:*

Business Component Specification **Uses** {« What I Am » Specification, « What I use » Specification, « What I can do » Specification}

The Business Component Specification pattern uses 3 patterns in order to model the BC.



- **Refines:** If a pattern P1 refines a pattern P2, then:
  - ✓ P1's Problem must be a specialization of P2's one.
  - ✓ P1's Classification may be enriched with respect to P2's one.
  - ✓ P1's Force may be enriched with respect to P2's one..
  - ✓ P1's Context may be enriched with respect to P2's one.

P1 refines P2	**P1**	**P2**
Classification	M1 ∧ M2	M2
Context	C1 ∧ C2	C2
Force	F1 ∧ F2	F2
Process Solution	***Pb1 is a specialization of Pb2***	Pb2

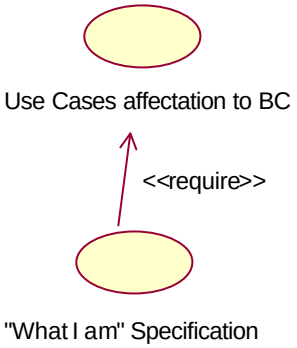
- **Requires:** If a pattern P1 requires a pattern P2, then:
  - ✓ P2's application is required in P1's one.
  - ✓ P2 must appear in P1's Context.

  P2 must have been executed before executing P1.

P1 requires P2	**P1**	**P2**
Context	***P2 must have been applied***	C2

Example:

"What I am" Specification **Requires** {Use Cases affectation to BC}



- **Alternative:** A pattern P1 is an alternative of a pattern P2 if P1 and P2 have different force items justifying different solutions to the same problem:
  - ✓ P1 and P2 have the same Classification, the same Context and the same Problem.
  - ✓ Only the Force of the two patterns is different.

P1 alternative of P2	P1	P2
Classification	M	M
Context	C	C
Problem	P	P
Force	***F1 ¹ F2***	F2

4 Conclusion

This article presented AGAP, a development environment suited to two types of actors, applications engineers and patterns engineers. AGAP addresses therefore two types of processes:

- a process by reuse allowing the applications engineer to define information systems by selecting, applying and integrating patterns applications,
- a process for reuse allowing the patterns engineer to define and to organize pattern systems.

AGAP clearly establishes a distinction between formalisms and pattern systems. It is therefore possible to define several pattern systems by using the same formalism or by reusing items of existing formalisms. An expected improvement of the meta-model consists in managing the synonymies between items having a similar meaning in different formalisms (for example, E. Gamma's Intention item and P-Sigma's Problem item). AGAP enables as well to capitalize item fields types. For example, specifying P-Sigma implies the definition of a great number of types. Thus, the application of class diagrams and sequence diagrams is achieved and can be reused to define items fields of others formalisms.

However, several use cases mentioned in the article have not yet been implemented (for example the patterns selection use cases) because the specificity of the "Interface" items is not currently taken into account in the tool. An ongoing study based on information research techniques should resolve it [3]. Another problem is due to the use of traces linked to patterns application. Patterns applications are indeed "naturally" preserved and keep a link with the

patterns from which they are issued. These imitations are represented by instances of AGAP's "Information System" component classes. These traces are nevertheless not yet exploited to facilitate the evolution and the maintenance of IS specifications.

AGAP was used to specify two formalisms (P-Sigma and E. Gamma's formalism), and two pattern systems written in P-Sigma. These pattern systems result from applied researches on two projects in collaboration with industrial companies. The first one focuses on the engineering of Product Information Systems (PIS) of industrial enterprises [13] and was developed in collaboration with Schneider Electric company (project CNRS PROSPER-POSEIDON). The goal was to propose an engineering process based on reuse of specifications of industrial products. Patterns technology offers a framework to consider a PIS engineering process as a set of process fragments, in order to design PIS models thanks to a set of model fragments. A pattern system was developed and validated; it is a significant evolution of existing pattern systems for it explicitly introduces a combination of process patterns and product patterns as well as a differentiation of inter-patterns relationships semantics [7].

The second pattern system developed in AGAP concerns the specification of Symphony, a development process based on business components proposed by the UMANIS company. UMANIS uses a pragmatic development process for business component oriented information systems. Coupled with this process, UMANIS wishes to develop an environment in which the engineers could be efficiently guided in their design activities, while taking into account different specific situations of targeted information systems as well as former expertise. In a first step, this goal led us to finalize P-Sigma in order to get patterns capitalizing knowledge and consensual expertise in engineering field, where decision and process aspects are important. P-Sigma's "Interface" part was in particular improved thanks to the selection of patterns to be applied according to the running context. A pattern system was then developed in which patterns are essentially process patterns modeling process fragments. This development process was also used to specify AGAP.

From these first validated results, other research works were initiated to facilitate reuse in information systems engineering field and to guaranty a traceability between design choices and software products resulting from the design. These works require either improvements of P-Sigma, or the definition of a meta-process to combine use of several formalisms. These formalisms would take into account others patterns, architectures and components forms: situational patterns [23] [22], frameworks [5], distributed objects on a CORBA bus, etc.

We would like to underline that these industrial experimentations showed the difficulty to enrich such pattern systems. In these experimentations, new patterns were found and formalized by researchers and not by application domain experts. One meets the same difficulties in the domain of knowledge bases building.

References:

[1] C. Alexander, *The Timeless Way of Building*, Oxford University Press, 1979.

[2] S.W. Ambler, *Process Patterns building Large Scale Systems using Object technology*, SIGS Books, Cambridge University Press, December 1998.

[3] C. Berrut, A. Front-Conte, *Patterns retrieval system: first attempt*, 5th International Conference on Applications of Natural Language to Information Systems (NLDB'2000), Versailles, June 2000.

[4] I. Borne, N. Revault, *Comparaison d'outils de mise en oeuvre de design patterns*, Object-oriented Patterns, Vol5, num2, 1999.

[5] F. Buschmann, R. Meunier & al., *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley & Sons, 1996.

[6] C. Casati, S Castano, M.G. Fugini, I. Mirbel, B. Pernici , *WERDE: a pattern-based tool for exception design in workflows*, proceedings of SEBD 98, Ancona, 1998.

[7] C. Cauvet, D. Rieu, P. Ramadour, et A. Front-Conte, *Réutilisation dans l'ingénierie des systèmes d'information*, Chapitre de l'ouvrage *Ingénierie des systèmes d'information* du *Traité IC2 – Information – Commande – Communication*, Hermès, Février 2001.

[8] P. Coad, D North et M Mayfield, *Object Models – Strategies, Patterns and Application,* Yourdon Press Computing Series, 1995.

[9] M. Fowler, *Analysis Patterns – Reusable Object Models*, Addison-Wesley, 1997.

[10] A. Front-Conte, J.P. Giraudin, D. Rieu, C. Saint-Marcel, *Réutilisation et patrons d'ingénierie,* Chapitre de l'ouvrage *Génie Objet: Analyse et Conception de l'Evolution*, Editeur M. Oussalah, Hermès, 1999.

[11] E. Gamma, R. Helm, R.E. Johnson, J. Vlissides, *Design patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[12] Dennis Gruijs: *A Framework of Concepts for Representing Object-Oriented Design and Design Patterns*, Masters Thesis, Utrecht University, CS Dept., INF-SCR-97-28, November.

[13] L. Gzara, D. Rieu, M. Tollenaere, *Pattern Approach To Product Information Systems Engineering,* Requirements Engineering Journal, Editors: Peri Loucopoulos & Colin Potts, Springer- Verlag, London, LTD., 2000.

[14] I. Hassine, D. Rieu, F. Bounaas, O. Seghrnouchni, Symphony : a Conceptual Model based on Business Component, IEEE SMC'02, Hammamet, Tunisia, October 2002.

[15] R.E. Johnson, *Documenting Frameworks using Patterns*, OOPSLA'92, 1992.

[16] N. Maiden, A. Sutcliffe, C. Taylor, D. Till, *A set of formal problem abstractions for reuse during requirements engineering*, ISI, Hermes, vol. 2, n° 6, pp. 679-698, 1994.

[17] M. Meijers, *Tools Support for Object-Oriented Design Patterns*, Master's Thesis, Utrecht University, 1996.

[18] T.D. Meijler, S. Demeyer, R. Engel, *Making design patterns explicit in face,* in European Software Engineering Conference (ESEC/FSE 97), 1997.

[19] L'Objet, *Numéro spécial Patrons orientés objet*, Coordonnateurs D. Rieu et J-P. Giraudin, Vol. 5, n° 2, Hermès, 1999.

[20] OMG Business Object Concept, BODTF, White Paper, Fred Cummings eds, BOM/99-01-01.

[21] B. Pagel, M Winter, *Toward pattern-based tools*, EuroPLoP'96,1996.

[22] C. Rolland, N. Prakash, A. Benjamen, *A multi-model view of process modeling*, Requirements Engineering Journal, pp 169-187, 1999.

[23] R.J. Welke, K. Kumar, *Method Engineering: a proposal for Situation-specific Methodology Construction*, in Systems Analysis and Design: a Research Agenda, Cotterman and Senn (eds), Wiley, pp 257-268, 1992.

[24] R. Wirfs-Brock, B. Wilkerson, L. Weiner, *Designing Object-Oriented Software* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

Avaliação da Aplicabilidade da Linguagem de Padrões de Engenharia Reversa de Demeyer a Sistemas Legados Procedimentais

Edson Luiz Recchia
DC - Universidade Federal de São Carlos / Universidade Anhembi Morumbi
erecchia@terra.com.br

Rosângela Penteado
DC - Universidade Federal de São Carlos
rosangel@dc.ufscar.br

Resumo

A Linguagem de Padrões de engenharia reversa existente na literatura conduz esse processo no contexto de orientação a objetos. Sua aplicação a sistemas legados procedimentais não fornece resultados expressivos para todos os padrões aplicados. Assim, para que a engenharia reversa de sistemas legados procedimentais seja realizada plenamente há necessidade de aprimorar a Linguagem de Padrões existente. Este trabalho tem por objetivo analisar cada um dos padrões de Demeyer quanto à sua aplicabilidade para sistemas legados procedimentais. Ressalta-se, entretanto, que essa análise foi necessária para que outros padrões pudessem ser elaborados de forma a atender a reengenharia orientada a objetos a partir de sistemas legados procedimentais.

Abstract

The Pattern Language for reverse engineering found in the literature conducts this process in an object-oriented context. Its application to procedural oriented legacy systems does not provide expressive results for all the patterns. Thus, it is necessary to improve this Pattern Language in order to realize wholly the reverse engineering process for procedural legacy systems. In this paper, the Pattern Language written by Demeyer is analyzed in a procedural context. However this analysis was necessary so that others patterns could be elaborated in order to became possible the reengineering object-oriented process from the procedural legacy systems.

1. Introdução

O interesse em transformar um sistema legado procedimental em um orientado a objetos, pelo processo de reengenharia, existe em diversas empresas. Isso ocorre devido à necessidade de alterar o ambiente e/ou a linguagem de programação para mais atual, alterar a interface para que se tenha maior usabilidade e o interesse em melhorar a manutenibilidade do sistema. Com essa preocupação surgem diversas abordagens, ferramentas e métodos para auxiliar o engenheiro de software no processo de reengenharia.

A Linguagem de Padrões de Demeyer [1] foi projetada para auxiliar no processo de engenharia reversa de sistemas orientados a objetos, porém suas idéias podem ser adaptadas para permitir a engenharia reversa de sistemas legados procedimentais, conforme proposto na Família de Padrões de Reengenharia - FaPRE/OO [3].

Assim, este trabalho discute a Linguagem de Padrões de Demeyer aplicada a sistemas legados procedimentais, apresentando o comportamento de cada um dos padrões que a compõem, quando aplicados para a realização da engenharia reversa de um sistema legado

procedimental. Essa discussão fornece os argumentos que incentivaram a criação da FaPRE/OO.

Este trabalho está organizado da seguinte forma: na Seção 2 apresenta-se a Linguagem de Padrões de Engenharia Reversa de Demeyer, na Seção 3 avalia-se cada padrão dessa linguagem de padrões para sistemas procedimentais, na Seção 4 a FaPRE/OO é brevemente descrita e são apresentadas as considerações finais.

2. Linguagem de Padrões de Engenharia Reversa proposta por Demeyer

A linguagem de padrões de engenharia reversa proposta por Demeyer [1] é o resultado do trabalho realizado com apoio do Governo Suíço sob o Projeto no. NFS-2000-46947.96 e BBW-96.0015 e, também, com o apoio da União Européia sob o programa ESPIRIT Projeto no. 21975 (FAMOOS). Essa linguagem resume a experiência no processo de engenharia reversa obtida como parte do Projeto FAMOOS, que teve como objetivo principal investigar técnicas de engenharia reversa e de reengenharia de sistemas orientados a objetos.

Essa linguagem de padrões tem como objetivo apoiar diferentes fases quando se realiza engenharia reversa em um sistema de software, desde quando não há familiaridade com o sistema até quando se está preparando a reengenharia em si. Essa linguagem de padrões foi dividida em *clusters* (Figura 1), cada qual resumido nas sub-seções seguintes.

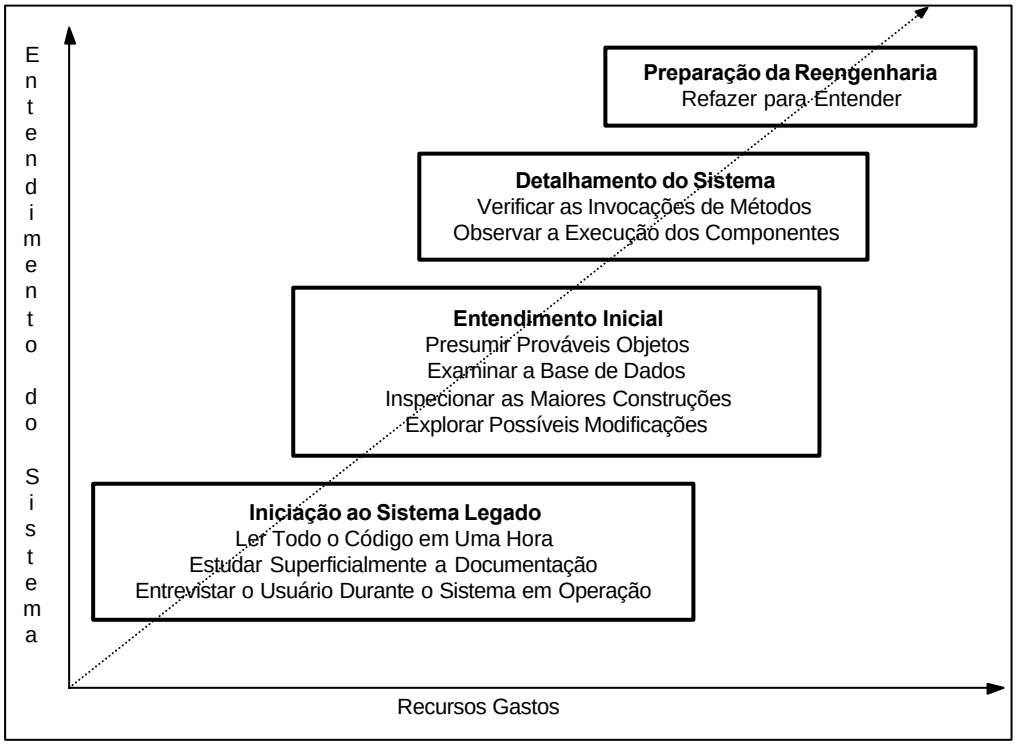


Figura 1: Linguagem de Padrões de Engenharia Reversa (extraída de Demeyer [1])

2.1 - *Cluster*: Iniciação ao Sistema Legado (*First Contact*)

Esse *cluster* agrupa os padrões que mostram o que fazer quando se tem o primeiro contato com um sistema de software. É composto dos seguintes padrões: Ler Todo o Código em Uma

Hora, Estudar Superficialmente a Documentação e Entrevistar o Usuário Durante o Sistema em Operação.

1. Nome: Ler Todo o Código em Uma Hora (*Read All the Code in one Hour*)
Intuito: Fazer uma avaliação inicial da condição do sistema através da leitura do código num tempo limitado.
Problema: Precisa-se de uma avaliação inicial da condição interna do sistema para planejar os esforços da engenharia reversa.
Contexto: **(a)** A condição interna do sistema varia muito, dependendo dos engenheiros de software envolvidos no desenvolvimento e na sua manutenção; **(b)** Em sistemas com milhares de linhas de código, existe muita informação a ser inspecionada, tornando-se difícil uma avaliação correta; **(c)** A falta de familiaridade do engenheiro de software com o sistema, dificulta filtrar o que é de fato necessário; **(d)** Tem-se o código fonte disponível, sendo uma informação confiável; **(e)** O engenheiro de software têm boa habilidade com a linguagem de implementação usada no sistema legado, assim pode-se identificar regras de negócio em trechos de códigos.
Solução: Ler o código fonte sem ser interrompido (sem atender telefonemas, sem barulho de colegas, etc.). Leia o código fonte por aproximadamente uma hora. Anote pouca coisa para maximizar seu contato com o código. Após esse tempo de leitura, gaste o mesmo tempo (uma hora) para produzir um relatório contendo suas constatações, incluindo: **(1)** Entidades importantes (classes, packages); **(2)** Linguagem do código; **(3)** Estilo de código de difícil interpretação; **(4)** Elabore esse relatório dando nomes às entidades, de acordo como essas são mencionadas no código fonte.
Padrões Relacionados: Os padrões Ler Todo o Código em Uma Hora e Estudar Superficialmente a Documentação maximizam a chance de se obter uma visão coerente do sistema. Para melhor compreender a documentação do sistema, pode-se preceder esse padrão com o padrão Entrevistar o Usuário Durante o Sistema em Operação.

2. Nome: Estudar Superficialmente a Documentação (*Skim the Documentation*)
Intuito: Supor, inicialmente, a funcionalidade do sistema por meio da leitura da sua documentação existente, num espaço limitado de tempo.
Problema: Planejar os esforços necessários para a realização da engenharia reversa a partir da idéia inicial do sistema.
Contexto: **(a)** A funcionalidade do sistema muda com o passar do tempo e muitas dessas mudanças podem não estar documentadas; **(b)** Em sistemas com milhares de linhas de código, existe muita informação a ser inspecionada, tornando-se difícil uma avaliação correta; **(c)** A falta de familiaridade do engenheiro de software com o sistema, dificulta filtrar o que é de fato necessário; **(d)** Tem-se a documentação disponível, assim há, pelo menos, uma descrição correta de como o sistema se comportou no passado; **(e)** O engenheiro de software é capaz de interpretar especificações formais (por exemplo, statecharts) e semi-formais (por exemplo, use cases) contidas na documentação, então ele é capaz de compreender o sistema.
Solução: Ler a documentação sem ser interrompido. Estude a documentação num curto espaço de tempo (aproximadamente uma hora). Faça pouca anotação para maximizar o contato com a documentação. Após esse tempo de leitura, gaste o mesmo tempo (uma hora) para produzir um relatório contendo suas constatações, incluindo: **(1)** Requisitos importantes; **(2)** Características importantes; **(3)** Limitações importantes; **(4)** Referências para informações relevantes do projeto.

Padrões Relacionados: O padrão Entrevistar o Usuário Durante o Sistema em Operação pode ajudar a coletar uma lista de entidades que se deseja analisar na documentação.

3. Nome: Entrevistar o Usuário Durante o Sistema em Operação (*Interview During Demo*)
Intuito: Obter a idéia inicial da funcionalidade do sistema observando-o em operação e entrevistando a pessoa que o está demonstrando.
Problema: Dimensionar os esforços necessários para a realização da engenharia reversa a partir dos cenários típicos de uso e das características principais do sistema.
Contexto: (a) A variação do uso de cenários entre diferentes usuários dificulta o seu entendimento; **(b)** Obter, a partir do usuário, o que há de errado com o sistema é difícil; **(c)** O engenheiro de software tem acesso a pessoas chaves na organização (usuários, gerentes e aqueles que dão manutenção no sistema), os quais podem demonstrar e explicar os cenários do sistema.
Solução: Observar o sistema em operação através de sua demonstração e entrevistar o usuário que o está demonstrando. Após essa demonstração produza um relatório contendo suas constatações, incluindo: **(1)** Alguns cenários típicos; **(2)** Características principais oferecidas pelo sistema e se elas são apreciadas ou não; **(3)** Componentes (encapsulamento) do sistema e suas responsabilidades.
Padrões Relacionados: Para se obter um resultado satisfatório deve-se aplicar esse padrão diversas vezes, com diferentes tipos de usuários. Dependendo da complexidade do sistema, pode-se exercitar esse padrão antes, depois ou durante o uso dos padrões Ler Todo o Código em Uma Hora e Estudar Superficialmente a Documentação.

2.2 - *Cluster*: Entendimento Inicial (*Initial Understanding*)

Esse *cluster* agrupa os padrões que descrevem como obter um entendimento inicial de um sistema de software documentado, principalmente, com diagramas de classes. É composto dos seguintes padrões: Presumir Prováveis Objetos, Examinar a Base de Dados, Inspecionar as Maiores Construções e Explorar Possíveis Modificações.

4. Nome: Presumir Prováveis Objetos (*Speculate about Domain Objects*)
Intuito: Refinar, progressivamente, um modelo de objetos de acordo com o código fonte, definindo hipóteses sobre quais objetos devem ser representados no sistema.
Problema: Não se sabe como os conceitos do negócio estão mapeados em classes no código fonte.
Contexto: (a) Existem muitos conceitos no sistema, assim há várias maneiras de representá-los na linguagem de programação utilizada; **(b)** Muito das linhas de código não tem relação com a representação dos conceitos do sistema, mas sim com temas relacionados à solução de implementação (interface do usuário, banco de dados, etc.); **(c)** Entende-se a funcionalidade do sistema, portanto pode-se ter uma idéia sobre o que o sistema representa; **(d)** Devido à experiência do engenheiro de software, ele pode imaginar como modelar o sistema.
Solução: Idealizar um modelo hipotético de classes que represente o sistema. Refinar esse modelo inspecionando se os nomes das classes ocorrem no código fonte e adaptando-o adequadamente. Repita o processo até que o modelo se estabilize.

1. Com o entendimento dos requisitos e os cenários de uso, desenvolver um modelo de classes que sirva como hipótese inicial do que se espera do código fonte. Dar nomes às classes, operações e atributos tomando por base sua experiência e nas convenções de nomenclatura adotadas.

2. Relacionar os nomes num diagrama de classes e tentar encontrá-los no código fonte, usando qualquer ferramenta que esteja disponível. Tomar cuidado com nomes existentes no código fonte, eles nem sempre representam o conceito desejado.
3. Manter anotados os nomes que aparecem no código fonte (confirma suas hipóteses) e aqueles que não combinam com o que foi identificado no código fonte (contradiz suas hipóteses). Note que as discordâncias são positivas, elas são motivo para o refinamento.
4. Adaptar o modelo de classes baseado nas discordâncias:
   (a) *renomeando as classes*, quando se descobre que os nomes no código fonte não combinam com suas hipóteses;
   (b) *remodelando as classes*, quando se descobre que a representação do código fonte não corresponde com o que se tem no modelo. Por exemplo, pode-se transformar uma operação em classe, ou um atributo em uma operação.
   (c) *estendendo as classes*, quando elementos importantes são observados no código fonte e não aparecem no diagrama de classes;
   (d) *procurando alternativas*, quando os conceitos de funcionalidade não são encontrados no código fonte. Isso pode implicar em colocar em teste sintomas quando existem algumas contradições, mas podem também implicar em definir um modelo de classes completamente diferente quando existem muitas contradições.

Padrões Relacionados: Todos os padrões do *cluster* Iniciação ao Sistema Legado auxiliam na construção do modelo hipotético de classes. O padrão Observar a Execução dos Componentes, do *cluster* Detalhamento do Sistema, pode ajudar a melhorar esse modelo.

5. Nome: Examinar o Banco de Dados (*Reconstruct the Persistent Data*)
Intuito: Adequar o modelo de objetos, obtido pelo padrão anterior, com o banco de dados do sistema legado.
Problema: Não se conhece quais os objetos que são críticos para a funcionalidade do sistema.
Contexto: Reconstruir o modelo de classes a partir das tabelas do banco de dados relacional.
Solução: Derivar um modelo de classes representando as entidades que estão armazenadas em forma de tabelas no banco de dados do sistema legado. Considerando-se um Banco de Dados Relacional genérico os seguintes passos devem ser aplicados:
1. Construir um modelo de classes, considerando cada tabela como uma classe, preservando o seu nome.
2. Selecionar como atributos os nomes das colunas correspondentes à cada tabela.
3. Selecionar todos os relacionamentos de chave estrangeira entre tabelas, considerando uma associação entre as classes correspondentes.

   Após esses passos, tem-se um modelo de classes que representa as entidades que estão armazenadas num banco de dados relacional. Contudo, banco de dados relacional não pode armazenar relacionamento de herança. Para isso são acrescentados os passos de 4 a 6.
4. Verificar as tabelas onde a chave primária também serve como chave estrangeira de outra tabela. Isso pode identificar um relacionamento um-para-um, indicando um relacionamento de herança. Neste caso, represente esse conjunto de tabelas como uma hierarquia de herança entre classes.
5. Verificar as tabelas com definições de campos semelhantes, indicando que provavelmente a hierarquia de classe está distribuída em várias tabelas. Neste caso, defina uma super-classe movendo os campos comuns para essa super-classe.

6. Verificar as tabelas com atributos opcionais. Isso pode indicar uma situação em que uma hierarquia de classe completa está representada em uma única tabela. Neste caso, transforme essa tabela em uma super-classe e gere várias subclasses para representar esse conjunto de informações.

Padrões Relacionados: O padrão Examinar o Banco de Dados requer um entendimento inicial da funcionalidade do sistema, o qual é obtido com o padrão Presumir Prováveis Objetos.

6. Nome: Inspecionar as Maiores Construções (*Identify the Largest*)
Intuito: Identificar trechos importantes de código utilizando ferramentas de determinação de métricas, inspecionando as maiores construções.
Problema: Não se sabe onde está implementada uma funcionalidade importante em milhares de linhas de código fonte.
Contexto: (a) Não existe uma maneira fácil de saber o que é mais ou menos importante no código fonte; **(b)** Em sistemas com milhares de linhas de código, existem muitos dados a ser inspecionados, tornando-se difícil uma avaliação correta; **(c)** Através da utilização de ferramentas de determinação de métricas é possível quantificar o tamanho das entidades no código fonte e verificar quais são importantes.
Solução: Usar ferramentas de determinação de métricas para coletar um conjunto limitado de medidas sobre as entidades dentro do sistema (por exemplo, hierarquias de herança, *packages*, classes e métodos). Represente os resultados de tal forma que se possa avaliar facilmente diferentes medidas para uma mesma entidade.
Exemplo: Identificar hierarquias de herança:
Identifique as maiores sub-árvores na hierarquia de herança como candidatas potenciais para fornecer funcionalidade importante. Para isso, construa uma lista de classes com as métricas: NDC (*Number of Descendent Class*), HNL (*Hierarchy Nesting Level*), NOM (*Number of Methods for Class*) e NOA (*Number of Attributes for Class*). Valores grandes de NDC, NOM e NOA e valores pequenos (~= 0) de HNL indicam que a funcionalidade importante está na raiz da hierarquia de herança. Valores pequenos de NDC, NOM e NOA (~= 0) e valores grandes de HNL indicam que a funcionalidade importante está nas folhas da hierarquia de herança.
Padrões Relacionados: O padrão Explorar Possíveis Modificações pode ser usado para focar nas partes do sistema que mudaram com as diferentes versões geradas, identificando assim funcionalidade importante.

7. Nome: Explorar Possíveis Modificações (Recover the Refactorings)
Intuito: Reconstruir o processo iterativo da construção do sistema, pela comparação de subseqüentes versões, observando o quanto o sistema cresceu ou decresceu, por meio de trechos de código fonte que foram alterados.
Problema: Recuperar o que os desenvolvedores do sistema alteraram durante o desenvolvimento e subsequentes manutenções.
Contexto: (a) O sistema tem sido modificado através de novas atualizações de versão e a comparação das várias versões é bastante trabalhosa; **(b)** Através de ferramentas é possível percorrer o código fonte, assim pode-se analisar as alterações realizadas nas diferentes versões do sistema; **(c)** Através da utilização de ferramentas de determinação de métricas é possível quantificar o tamanho das entidades no código fonte; **(d)** Devido à experiência do engenheiro de software, ele pode inferir porque certas modificações foram aplicadas.

Solução: Usar ferramentas de determinação de métricas para comparar medidas de subsequentes versões do sistema legado, encontrando entidades cujos valores de medidas aumentaram ou diminuíram. Assim, descobre-se onde funcionalidade foi incluída ou removida, respectivamente.
Padrões Relacionados: O padrão Inspecionar as Maiores Construções pode ser usado para descobrir partes do sistema que mudaram com as diferentes versões geradas.

2.3 - *Cluster*: Detalhamento do Sistema (*Detailed Model Capture*)

Esse cluster agrupa os padrões que mostram como obter um entendimento detalhado de um determinado componente (encapsulamento) em seu sistema de software. Esse *cluster* é composto dos seguintes padrões: Verificar as Invocações de Métodos e Observar a Execução dos Componentes.

8. Nome: Verificar as Invocações de Métodos (*Derive Public Interface*)
Intuito: Saber como uma classe está relacionada com outra verificando os parâmetros definidos nos métodos da interface da classe.
Problema: Obter o relacionamento entre classes no sistema legado.
Contexto: (a) No estágio final da engenharia reversa tem-se uma visão global da funcionalidade do sistema e baseado nesse entendimento pode-se selecionar classes para uma inspeção futura; **(b)** Com uma ferramenta adequada, para percorrer o código, é possível identificar onde o método está definido a partir de sua chamada.
Solução: A partir da inspeção de uma chamada do método, encontrar a sua assinatura na interface da classe aonde ele foi definido, descobrindo informações por meio da conexão de mensagens.
Padrões Relacionados: Os padrões Inspecionar as Maiores Construções e Explorar Possíveis Modificações podem ser usados para descobrir funcionalidade importante a ser avaliada.

9. Nome: Observar a Execução dos Componentes (*Step Through the Execution*)
Intuito: Obter um entendimento detalhado do comportamento de uma parte do código, através da execução de seus componentes (encapsulamento).
Problema: É preciso obter o entendimento detalhado de uma parte encapsulada do código.
Contexto: (a) Baseado no entendimento do sistema pode-se selecionar trechos de código para inspeção; **(b)** Com uma ferramenta para depurar o código, é possível inspecionar estruturas de dados e interagir com a execução, passo a passo, de pedaços de código.
Solução: Alimentar, com um conjunto representativo de massa de teste, a entrada do pedaço de código fonte para se obter uma seqüência normal de operações. Usar um depurador para acompanhar a execução passo a passo e inspecionar o estado interno do pedaço de código.
Padrões Relacionados: Os padrões Inspecionar as Maiores Construções e Explorar Possíveis Modificações podem ser usados para descobrir funcionalidade importante a ser avaliada.

2.4 - *Cluster*: Preparação da Reengenharia (*Prepare Reengineering*)

Esse *cluster* possui um padrão que ajuda a preparar os passos subsequentes de reengenharia: Refazer para Entender.

10. Nome: Refazer para Entender (*Refactor To Understanding*)
Intuito: Obter um melhor entendimento de uma parte específica do código fonte, refazendo-a.

"**Refazer para Entender**" é o processo de modificar um sistema de software, de tal maneira que o comportamento externo do código não seja alterado, mas sua estrutura interna seja melhorada. É uma forma disciplinada de colocar em ordem o código, minimizando as possibilidades de introduzir "bugs". Na essência, quando se realiza esse processo se está melhorando o projeto do código, depois dele ter sido escrito.

Problema: Compreender um particular trecho de código que aparenta ser importante mas é muito difícil de analisá-lo completamente.

Contexto: (a) Código sem documentação é difícil de se ler, então é difícil, também, de se entender; **(b)** Alterar código sem documentação pode causar efeitos colaterais.

Solução: Renomear interativamente e refazer o código para introduzir nomes significativos e se certificar de que a estrutura do código reflete o que o sistema está fazendo de fato.

1. **Remover código duplicado:** Quando se identifica código duplicado, tentar refazê-lo num simples local. Por exemplo, transformar o trecho de código num método e, nos locais onde foi encontrado o código duplicado, substituí-lo por uma chamada do Método.
2. **Substituir trechos condicionais por Métodos:** Quando se encontra grande construção condicional, transformar os trechos de código de cada condição em novos métodos dando a eles nomes baseados nas condições. Continue o processo até que se tenha o entendimento completo da estrutura do código.
3. **Substituir trechos de código longo por Métodos:** Longos trechos de código com comentários separando blocos de código violam a regra de que todos os comandos num simples trecho de código deveriam ter o mesmo nível de abstração. Refazer cada bloco introduzindo um novo método.
4. **Renomear atributos com nomes significativos:** Procurar por atributos com nomes obscuros. Procurar saber sobre o seu contexto e dar nomes significativos.
5. **Renomear métodos com intuito significativo:** Procurar por métodos que não tem nomes relacionados com a sua funcionalidade. Recupere os seus objetivos, investigue todas as chamadas desses métodos de acordo com os seus objetivos.
6. **Renomear classes com propósitos significativos:** Encontrar classes cujos nomes não são representativos com a funcionalidade. Encontrar seus objetivos e então renomeá-las de acordo com esses objetivos.

Padrões Relacionados: Para ajudar a entender a funcionalidade, pode-se usar o padrão Observar a Execução dos Componentes.

3. Avaliação da Aplicabilidade da Linguagem de Padrões de Engenharia Reversa de Demeyer a Sistemas Legados Procedimentais

A seguir é apresentada a avaliação da aplicabilidade de cada padrão da Linguagem de Padrões de Demeyer a sistemas legados procedimentais. O objetivo desta avaliação não é de criticar tais padrões, visto que foram propostos com objetivos diferentes, mas justificar a necessidade da criação da FaPRE/OO [2] [3].

Para a avaliação da aplicação de cada padrão muitos itens do formato apresentado anteriormente não serão novamente apresentados. Serão descritos, apenas, os itens Avaliação (*Trade off*) e Justificativa, conforme sugerido por Demeyer, pois esses devem conter a opinião dos engenheiros de software quando da utilização dos padrões em seus processos de engenharia reversa, além dos itens Nome e Intuito de cada padrão.

3.1 - *Cluster:* **Iniciação ao Sistema Legado** (*First Contact*)

1. Nome: Ler Todo o Código em Uma Hora (*Read All the code in one Hour*)
Intuito: Fazer uma avaliação inicial da condição do sistema através da leitura do código num tempo limitado.
Avaliação (*Trade-off*):

Prós:
O código fonte de sistemas orientados a objetos e, também implementados em linguagens orientadas a objetos, possibilita uma análise rápida devido às próprias características desse contexto. Sendo assim, com a aplicação desse padrão o engenheiro de software obtém algumas informações (classes, objetos, métodos, etc.) para poder dimensionar os esforços que terá que despender durante o processo de engenharia reversa.

Contra:
O código fonte de sistemas procedimentais implementados em linguagens tais como, CLIPPER, COBOL, RPGII, etc., é, em geral, seqüencial e dividido em módulos funcionais. Com a aplicação desse padrão consegue-se obter poucas informações relevantes sobre um sistema legado procedimental.

Justificativa: Esse padrão não é aplicado em sistemas legados procedimentais porque nesses sistemas não se constata o encapsulamento que é uma característica dos sistemas orientados a objetos. Dessa forma pode-se inferir, num curto espaço de tempo, poucas informações tais como, as entidades mais importantes, estilo de programação realizada, etc.

2. Nome: Estudar Superficialmente a Documentação (*Skim the Documentation*)
Intuito: Supor, inicialmente, a funcionalidade do sistema por meio da leitura da documentação existente do sistema, num espaço limitado de tempo.
Avaliação (*Trade-off*):

Prós:
Independente do contexto no qual o sistema legado foi desenvolvido (orientado a objetos ou procedimental), a documentação existente pode contribuir na análise inicial da sua funcionalidade. Em sistemas legados orientados a objetos, a documentação existente pode ser produzida com ajuda de ferramenta *Case*. A partir dessa documentação consegue-se obter a funcionalidade inicial do sistema legado num curto espaço de tempo.

Contra:
Segundo estudos de casos realizados em sistemas legados procedimentais, sabe-se que esses sistemas possuem, em geral, pouca documentação. Na sua maioria o que existe, de fato, são, código fonte, estruturas de arquivos de dados e sistema executável. Com a análise dessa documentação não se consegue obter a funcionalidade inicial do sistema legado num curto espaço de tempo.

Justificativa: Este padrão não é aplicado em sistemas legados procedimentais por não ser possível obter informações relevantes no curto espaço de tempo sugerido.

3. Nome: Entrevistar o Usuário Durante o Sistema em Operação (*Interview During Demo*)
Intuito: Obter a idéia inicial da funcionalidade do sistema observando-o em operação e entrevistando a pessoa que o está demonstrando.
Avaliação (*Trade-off*):

Prós:
A observação do sistema em execução permite ao engenheiro de software o conhecimento de sua funcionalidade.

Contra:

Geralmente, sistemas grandes e integrados envolvem mais do que um usuário, por exemplo, sistemas do tipo ERP – *Entrerprise Resource Planning*, em que existem vários sub-sistemas interagindo entre si. Esse padrão deve ser aplicado a todos os usuários envolvidos para se conseguir a maioria dos cenários típicos. Além disso, deve-se ter a preocupação de obter informações das interfaces de Negócios entre os subsistemas. Na maioria das vezes isso é feito através de reuniões coletivas com todas as áreas envolvidas da organização. Sendo assim, nesse contexto de sistema integrado, não se consegue obter as informações necessárias num curto espaço de tempo.

Justificativa: Esse padrão foi utilizado na elaboração da FaPRE/OO, para os padrões: Construir Diagramas de *Use Cases* e Obter Cenários. Pois, entrevistar usuários quando estão operando o sistema e engenheiros de software que participaram do desenvolvimento do sistema legado é essencial para se obter informações da funcionalidade e dos cenários típicos de uso.

3.2 - *Cluster:* Entendimento Inicial (*Initial Understanding*)

4. Nome: Presumir Prováveis Objetos (*Speculate about Domain Objects*)

Intuito: Refinar, progressivamente, um modelo de objetos de acordo com o código fonte, definindo hipóteses sobre quais objetos devem ser representados no sistema.

Avaliação (*Trade-off*):

Prós:

Este padrão concebe um modelo hipotético de classes, que representa o sistema a partir das classes/objetos, de acordo com a suposição das informações levantadas pelos padrões anteriores. Refina-se esse modelo inspecionando se os nomes dessas classes/objetos ocorrem no código fonte e adaptando-as adequadamente ao modelo.

Contra:

Em sistemas procedimentais não existem classes/objetos claramente definidas no código fonte. Assim, não é possível fazer suposição de um modelo de classes/objetos somente a partir desse código fonte. No entanto, é possível elaborar um modelo inicial a partir dos dados e do código fonte do sistema legado.

Justificativa: Este padrão foi usado duas vezes durante a elaboração da FaPRE/OO, utilizando-se como hipótese de possíveis classes as entidades do MER, Modelo Entidade Relacionamento, da seguinte forma:

1) Com base no código fonte elabora-se a Tabela Detalhes de Implementação, usando o padrão Tratar Anomalias.

2) Com base na Tabela Detalhes de Implementação e no MER foram efetuados refinamentos gerando o diagrama de classes do sistema, por meio do padrão Definir as Classes.

Durante a aplicação desse padrão nos estudos de caso, em sistemas procedimentais, pôde-se aproveitar a essência da concepção desse padrão adaptando-o ao contexto procedimental, da seguinte forma: utilizou-se como hipótese de possíveis classes os arquivos de dados, uma vez que esses arquivos em sistemas procedimentais representam, na sua maioria, entidades importantes do sistema. A partir daí um modelo de dados procedimental, e não de classes, é obtido com refinamentos sucessivos a partir dos dados e do código fonte.

5. Nome: Examinar o Banco de Dados (*Reconstruct the Persistent Data*)
Intuito: Adequar o modelo de objetos, obtido pelo padrão anterior, com o banco de dados do sistema legado.
Avaliação (*Trade-off*):

Prós:

Em sistemas orientados a objetos, implementados em bancos de dados relacionais, existem tabelas em que as regras de acesso são bem definidas. Portanto, consegue-se derivar um modelo de classes representando as entidades que estão armazenadas no banco de dados.

Contra:

Em sistemas procedimentais, em geral, não existem arquivos de dados definidos claramente como as tabelas dos bancos de dados relacionais. Sendo assim, não é possível elaborar um modelo somente com a análise dos dados, tem-se que analisar também o código fonte para encontrar as regras de acesso. Portanto, é possível elaborar um modelo de classes a partir dos dados e do código fonte do sistema.

Justificativa: Este padrão foi usado duas vezes durante a elaboração da FaPRE/OO:

1) Este padrão contribuiu, em parte, para a elaboração dos padrões, Iniciar a Análise dos Dados e Definir Chaves, na construção do MER do sistema legado.

2) Este padrão também contribuiu para a elaboração do padrão Analisar Hierarquias na construção do Diagrama de Classes do sistema.

É comum em sistemas orientados a objetos ter um banco de dados relacional contendo os dados persistentes. Já para sistemas procedimentais têm-se arquivos de dados. No entanto, os conceitos de banco de dados podem ser aplicados nesses sistemas, gerando modelos representativos das informações do Negócio. Durante a aplicação desse padrão nos estudos de caso, em sistemas procedimentais, pôde-se aproveitar a essência da sua concepção adaptando-o ao contexto procedimental, da seguinte forma: como possíveis classes foram utilizados os arquivos de dados, uma vez que, esses nos sistemas procedimentais representam, na sua maioria, entidades importantes do sistema. A partir daí um modelo de pseudo-classes é obtido sendo refinado sucessivamente a partir do código fonte e da análise dos arquivos de dados.

6. Nome: Inspecionar as Maiores Construções (*Identify the Largest*)
Intuito: Identificar trechos importantes de código utilizando ferramentas de determinação de métricas, inspecionando as maiores construções.
Avaliação (*Trade-off*):

Prós:

Em sistemas orientados a objetos pode-se usar ferramentas de determinação de métricas para coletar um conjunto limitado de medidas sobre as entidades do sistema (por exemplo: hierarquia de herança, *packages*, classes, métodos, etc.). Pode-se representar os resultados de tal forma que seja possível avaliar, facilmente, diferentes medidas para uma mesma entidade.

Contra:

Não é possível implementar os conceitos de, hierarquia de herança, *packages*, etc., em sistemas procedimentais.

Justificativa: Este padrão não foi utilizado na elaboração da FaPRE/OO, mas o uso de métricas pode ser explorado para auxiliar a engenharia reversa de sistemas procedimentais.

7. Nome: Explorar Possíveis Modificações (*Recover the Refactorings*)
Intuito: Reconstruir o processo iterativo da construção do sistema, pela comparação de subseqüentes versões, observando o quanto o sistema cresceu ou decresceu, por meio de trechos de código fonte que foram alterados.
Avaliação (*Trade-off*):

Prós:

Para sistemas legados orientados a objetos pode-se usar ferramentas de determinação de métricas, propostas pelo padrão anterior, para comparar medidas de versões subsequentes a fim de encontrar entidades onde funcionalidade pode ter sido incluída ou removida.

Contra:

Não é possível implementar os conceitos de, hierarquia de herança, *packages*, etc., em sistemas procedimentais

Justificativa: Este padrão não foi utilizado na elaboração da FaPRE/OO, mas o uso de métricas pode ser explorado para auxiliar a engenharia reversa de sistemas procedimentais.

3.3 - *Cluster:* Detalhamento do Sistema (*Detailed Model Capture*)

8. Nome: Verificar as Invocações de Métodos (*Derive Public Interface*)
Intuito: Saber como uma classe está relacionada com outra verificando os parâmetros definidos nos métodos da interface da classe.
Avaliação (*Trade-off*):

Prós:

Em sistemas orientados a objetos é possível encontrar informações através da assinatura dos métodos, constantes da interface da classe, descobrindo-se como uma classe está relacionada com outra por meio dos parâmetros (conexão de mensagens).

Contra:

Em sistemas procedimentais a implementação dos conceitos de conexão de mensagens não é como em sistemas orientados a objetos. A comunicação entre arquivos de dados nos sistemas procedimentais pode ser obtida através da análise do código fonte.

Justificativa: Este padrão não é aplicado em sistemas legados procedimentais porque nesses sistemas não se constata os conceitos da orientação a objetos.

9. Nome: Observar a Execução dos Componentes (*Step Through the Execution*)
Intuito: Obter um entendimento detalhado do comportamento de uma parte do código, através da execução de seus componentes (encapsulamento).
Avaliação (*Trade-off*):

Prós:

Esse padrão tem uma proposta muito interessante para se entender parte do código através da execução dos componentes. Seu conceito é também bastante utilizado, na prática, pelos engenheiros de software. Esse processo pode ser comparado à fase de teste nos sistemas procedimentais.

Contra:

Em sistemas procedimentais o encapsulamento, existente em orientação a objetos, não é realizado quando da implementação de sistemas, ou seja, essa implementação é realizada por meio de um conjunto de linhas seqüenciais de código.

Justificativa: Este padrão não foi utilizado devido à formação do código fonte dos sistemas legado procedimentais, uma vez que o padrão Refazer para Entender pode ser melhor aplicado no contexto de sistemas procedimentais.

3.4 - *Cluster:* Preparação da Reengenharia (*Prepare Reengineering*)

10. Nome: Refazer para Entender (*Refactor To Understanding*)
Intuito: Obter um melhor entendimento de uma parte específica do código fonte, refazendo-a.

Avaliação (*Trade-off*):

Prós:

Com este padrão consegue-se modificar partes do código fonte, de tal maneira que o comportamento externo do código não seja alterado, mas a sua estrutura interna seja melhorada.

Contra:

Em sistemas procedimentais esse padrão requer profundo conhecimento da linguagem de programação utilizada na implementação do sistema legado.

Justificativa: Este padrão foi utilizado, em parte, na elaboração do padrão Tratar Anomalias, da FaPRE/OO. Com a aplicação desse padrão, apesar da necessidade do conhecimento da linguagem, é possível eliminar inconsistências (anomalias) comumente encontradas em sistemas procedimentais.

4. Considerações Finais

Este trabalho apresentou, primeiramente, a linguagem de padrões proposta por Demeyer [1] justificando, em seguida, que não é suficiente utilizá-la para processos de engenharia reversa de sistemas legados procedimentais. Para solucionar esse problema, foi construída a FaPRE/OO, contendo padrões para conduzir processos de reengenharia orientada a objetos de sistemas legados procedimentais [2] [3].

A FaPRE/OO é composta de quatro *clusters,* cada um agrupando os padrões relacionados a situações similares da reengenharia, sendo os três primeiros para o processo de engenharia reversa e o último para o processo de engenharia avante:

- *Cluster* 1 - Modelar os Dados do Legado - agrupa os seguintes padrões: Iniciar Análise dos Dados, Definir Chaves, Identificar Relacionamentos e Criar Visão OO dos Dados;
- *Cluster* 2 - Modelar a Funcionalidade do Sistema - agrupa os seguintes padrões: Obter Cenários, Construir Diagramas de Use Cases, Elaborar a Descrição de Use Cases e Tratar Anomalias;
- *Cluster* 3 - Modelar o Sistema Orientado a Objetos - agrupa os seguintes padrões: Definir as Classes; Definir Atributos, Analisar Hierarquias, Definir Métodos e Construir Diagramas de Seqüência; e
- *Cluster* 4 - Gerar o Sistema Orientado a Objetos - agrupa os seguintes padrões: Definir a Plataforma, Converter o Banco de Dados, Implementar os Métodos e Realizar Melhorias na Interface.

Tabela 1 - Padrões de Demeyer utilizados na elaboração dos Padrões para o Processo de Engenharia Reversa da FaPRE/OO

Linguagem de Padrões de Demeyer		**Padrões para o Processo de Engenharia Reversa da FaPRE/OO**	
Clusters	**Padrões**	**Padrões**	***Clusters***
Iniciação ao Sistema Legado	. Ler Todo o Código em uma Hora	—	—
	. Estudar Superficialmente a Documentação	—	—
	. Entrevistar o Usuário Durante o Sistema em Operação	. Construir Diagramas de *Use Cases* . Obter Cenários	Modelar a Funcionalidade do Sistema
Entendimento Inicial	. Presumir Prováveis Objetos	. Definir as Classes	Modelar o Sistema Orientado a Objetos
		. Tratar Anomalias	Modelar a Funcionalidade do Sistema
	. Examinar a Base de Dados	. Iniciar Análise dos Dados . Definir Chaves	Modelar os Dados do Legado
		. Analisar Hierarquias	Modelar o Sistema Orientado a Objetos
	. Inspecionar as Maiores Construções	—	—
	. Explorar Possíveis Modificações	—	—
Detalhamento do Sistema	. Verificar as Invocações dos Métodos	—	—
	. Observar a Execução dos Componentes	—	—
Preparação da Reengenharia	. Refazer para Entender	. Tratar Anomalias	Modelar a Funcionalidade do Sistema

Tabela 2 - FaPRE/OO x Padrões de Demeyer

Padrões para o Processo de Engenharia Reversa da FaPRE/OO		**Linguagem de Padrões de Demeyer**	
Clusters	**Padrões**	**Padrões**	***Clusters***
Modelar os Dados do Legado	. Iniciar Análise dos Dados	. Examinar a Base de Dados	Entendimento Inicial
	. Definir Chaves	. Examinar a Base de Dados	Entendimento Inicial
	. Identificar Relacionamentos	—	—
	. Criar Visão OO dos Dados	—	—
Modelar a Funcionalidade do Sistema	. Obter Cenários	. Entrevistar o Usuário Durante o Sistema em Operação	Iniciação ao Sistema Legado
		. Presumir Prováveis Objetos	Entendimento Inicial
	. Construir Diagramas de *Use Cases*	. Entrevistar o Usuário Durante o Sistema em Operação	Iniciação ao Sistema Legado
	. Elaborar a Descrição de *Use Cases*	—	—
	. Tratar Anomalias	. Presumir Prováveis Objetos . Refazer para Entender	—
Modelar o Sistema Orientado a Objetos	. Definir as Classes	. Presumir Prováveis Objetos	—
	. Definir Atributos	—	—
	. Analisar Hierarquias	. Examinar a Base de Dados	Preparação da Reengenharia
	. Definir Métodos	—	—
	. Construir Diagramas de Seqüência	—	—

A Tabela 1 mostra os padrões de Demeyer utilizados para a elaboração da FaPRE/OO. Por exemplo, do primeiro *cluster* da linguagem de padrões de Demeyer, somente o padrão Entrevistar o Usuário durante o Sistema em Operação é utilizado. Os padrões Inspecionar as Maiores Construções, Explorar Possíveis Modificações, Verificar as Invocações dos Métodos e Observar a Execução dos Componentes não foram utilizados, mas podem ser úteis em futuros refinamentos do processo. Cabe ressaltar que alguns padrões da linguagem de Demeyer foram utilizados, algumas vezes, para a elaboração de mais do que um padrão da FaPRE/OO, como mostra a Tabela 1.

A Tabela 2 complementa a Tabela 1 mostrando o relacionamento existente entre os padrões para o processo de engenharia reversa da FaPRE/OO e a linguagem de padrões de Demeyer.

Referências

[1] Demeyer, S.; Ducasse, S.; Nierstrasz, O., **"A Pattern Language for Reverse Engineering".** Proceedings of the 5th European Conference on Pattern Languages of Programming and Computing, (EuroPLOP'2000), Andreas Ruping(Ed.), 2000.

[2] Recchia, E. L., **Engenharia Reversa e Reengenharia Baseadas em Padrões**, São Carlos-SP, Junho/2002. Dissertação de Mestrado apresentada ao PPGCC - Universidade Federal de São Carlos.

[3] Recchia, E. L.; Penteado, R. – **FaPRE/OO: Uma Família de Padrões para Reengenharia Orientada a Objetos de Sistemas Legados Procedimentais.** Artigo apresentado no SugarloafPLoP2002 – The Second Latin American Conference on Pattern Languages of Programming – Agosto/2002, Itaipava – RJ.

Analyzability and Changeability in Design Patterns[1]

Javier Garzás[1] and Mario Piattini[2]

[1] ALTRAN SDB Senior Consultant
Projects Engineering Research Group
ALTRAN SDB
C/ Ramírez de Arellano, 15, 28043, Madrid - SPAIN
jgarzas@altransdb.com

[2] Alarcos Research Group
Escuela Superior de Informática,
University of Castilla-La Mancha
Ronda de Calatrava, s/n. 13071, Ciudad Real – SPAIN
Mario.Piattini@uclm.es

Abstract

It has been a long time since the appearance of the Object Oriented (OO) paradigm. From that moment, the designers have accumulated much knowledge in the design and construction of OO systems. However, at the present time the exclusive use of patterns is not sufficient to guide a design in a formal way. Two important quality parameters for Object Oriented (patterns) Design exist: Analyzability and Changeability. Principles permit to us to analyze an easier manner in which to introduce design patterns. Indirections provide changeability to the pattern. With all the former, we can obtain a metric to answer how much Changeability we can apply in order not to lose the design's Analyzability.

1. Introduction

In the late eighties, the application of patterns in OO appeared and was consolidated, among others, by the work of Coad (1992), Gamma *et al.* (1995), Buschmann *et al.* (1996), Fowler (1996) and Rising (1998). The motivation was to transfer a type of Object Oriented Design Knowledge (OODK) (Garzás and Piattini, 2001), knowledge accumulated during years of experience. Since then, designers have been reading and using patterns, reaping benefit from this experience.

However, at the present time the exclusive use of patterns is not sufficient to guide a design in a formal way, the designer's experience being necessary to avoid overload, non-application or the wrong use of patterns due to ignorance, or any other problems that may give rise to faulty and counteractive use of the patterns. When patterns are used, several types of problems may occur (Wendorff, 2001; Schmidt, 1995):

- Difficult Application.
- Difficult Learning.

- Temptation to Recast everything as a pattern
- Pattern overload.
- Ignorance.
- Deficiencies in catalogues: Search and Complex Application, High Dependence of the Programming Language, Comparatives, etc.

In principle, using design patterns increments design quality. In this sense, there are many works about metrics and design quality, for example (Genero et al., 2000), (Brito e Abreu and Carapuça, 1994), (Briand et al., 1999), (Henderson-Sellers, 1996), etc. Since design quality can be measured by quality metrics, the use of design patterns should lead to better measurements. However, many common object-oriented design metrics indicate lower quality if design patterns are used. In this sense, Reibing (2001) comments that if we have two similar designs A and B for the same problem, B using design patterns and A not using design patterns, B should have a higher quality than A. However, if we apply "classic" object-oriented design metrics to both designs, the metrics tell us that design A is better – mostly because it has less classes, operations, inheritance, associations, etc. Who is wrong? The metrics or the pattern community? Do we have the wrong quality metrics for object-oriented design? Or does using patterns in fact make a design worse, not better? So what is the cause of the contradiction between the supposed quality improvement by design patterns and the measured quality deterioration? (Reibing, 2001)

First, in the following section, we will analyze the maintenance and design patterns and relationship with analyzability and changeability in more detail. Later, we will show a measurement of the impact of the patterns used. In the last sections, we present acknowledgments, our conclusions and future projects, and references.

2. Maintenance and design patterns

According to the ISO/IEC 9126 – 1999 *"Software Product Evaluation – Quality characteristics and Guidelines for their use"* standard maintainability is subdivided into Analyzability, Changeability, Stability, Testability and Compliance.

Software maintenance consumes the largest part of the overall lifecycle cost (Pigoski, 1997; Bennett and Rajlich, 2000). The incapacity to change software quickly and reliably means that organizations lose business opportunities. Thus, in recent years we have seen an important increase in research directed at addressing these issues.

If we obtain a correct OO design, we will obtain better maintenance. Considering the 9126 standard, two important parameters for quality maintenance of Object Oriented Design (patterns) exist:

- *Changeability* allows a design to be able to change easily, an important requirement at the time of extended functionality into an existing code.
- *Analyzability* allows us to understand the design. This is an essential requisite in order to be able to modify the design in a realistic period of time.

To be specific, at the time of applying patterns to a software design, two opposites forces appear and these forces are directly related to maintainability. On the one hand, we put together a common terminology to the applying patterns, we have proven solutions, but we have the inconvenience that the solution, once obtained, can be very complex, and this means that the design is less comprehensible, and modifying the design is more difficult (Prechelt, 2000). Thus, to continue with the previous concepts, a curious relation between Changeability and Analyzability appears: If we increase the design's Changeability then we will decrease the design's Analyzability, and vice versa.

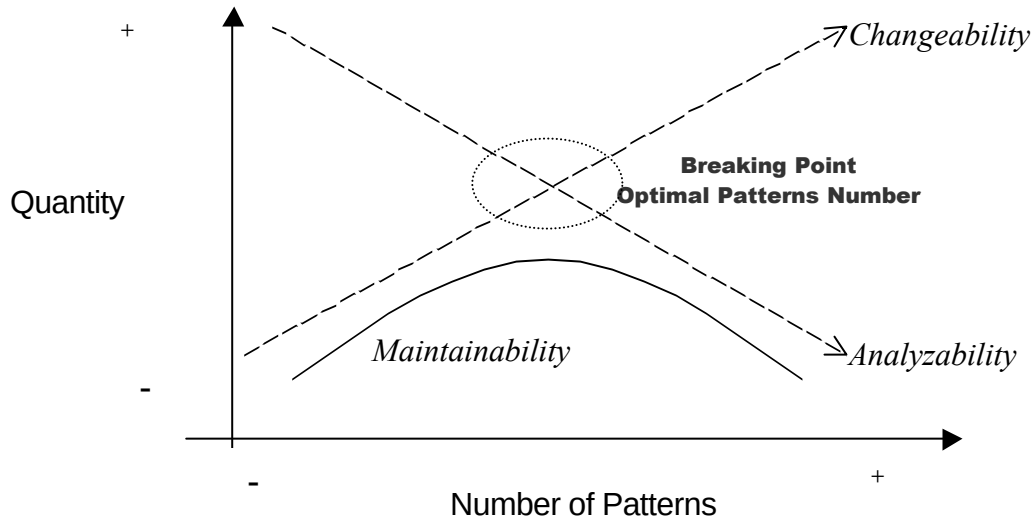


Fig. 1. The Relationship between Changeability and Analyzability. The Breaking Point determines the "Optimal Patterns Number", where the best maintenance is in relation to Patterns.

Figure 1 shows graphically the relationship between Changeability and Analyzability when patterns are applied. As the figure shows:

- If a design has a lot of design patterns this design will have a great amount of Changeability
- If a design has few design patterns this design will have a great amount of Analyzability

2.1 Analyzability issues

In general, the pattern introduction is a complex task. In this epigraph, we show how to introduce design patterns to the design from design principles. This method permits us to analyze an easier way to introduce design patterns.

Garzás and Piattini (2001) comment that an OOD principle can be defined as a set of proposals or truths based on experience that form the foundation of OOD and whose purpose is to control this process. Some principles are the following (other principles apart from those described here may exist but we are limited by the length of this paper):

- **Open-Closed Principle (OCP):** A module should be open for its extension and closed for its modification.
- **Substitution Principle (SP):** The subclasses must be substitutable by their base classes.
- **Dependency Inversion Principle (DIP):** Depend upon abstraction. Do not depend upon specifications.
- **Interface Segregation Principle (ISP):** Many clients specific interfaces are better than one general purpose interface.
- **Default Abstraction Principle (DAP):** Introduces an abstract class that makes the implementation in default of most of the interface operations between the interface and the class that implements it.
- **Interface Design Principle (IDP):** "Program" an interface, not an implementation.
- **Black Box Principle (BBP):** Favor the object composition over class inheritance.
- **Do not Concrete Superclass Principle (DCSP):** Avoid maintaining concrete superclass.

In general, we can state that in order for an OO system to be of a certain quality it should not violate any principles. On the other hand, patterns contribute to an efficient design, but in general the exact relationship between principles and patterns is unknown or more specifically we do not know which principle(s) ensure(s) each pattern.

Therefore, for example, in order to conform to the DIP, one of the strategies could be to use the abstract Factory pattern. The purpose of other patterns such as Prototype, Factory method, etc. is more to perform the Abstract Factory than to directly conform to a principle. Therefore, we can conclude that there are patterns that directly allow a principle to be complied with, whilst other patterns are more related to patterns than to principles. Consequently, patterns could be classified according to the principles they follow. The principles would even enable us to create a different catalogue of patterns to that currently existing (in most cases they are simply presented in alphabetical order). Checklists of principles could also be drawn up which assess the design and offer us solution patterns that ensure that they are complied with. We may specify more and consider their relationship with the patterns, so that the principles can be one or several of the following types:

Type 1, the pattern contributes to a good solution to the resulting model of the application of the principle ("from the principle towards the pattern").	***Type 2***, the pattern completes or contains the principle.	***Type 3***, the principle can improve a solution to which a pattern has previously been applied ("from the pattern towards the principle").

Table 1 shows an analysis of the principles mentioned in the previous epigraphs and their relationship with each pattern of those detailed by Gamma *et al.* (1995) in function of the previous types.

We can observe that the relationship of patterns has been ordered alphabetically. In this way, we can obtain an objective order and later, based on the principles, we will be able to obtain analogies.

Principle		OCP			SP			DIP			ISP			DAP			IDP			BBP			DCSP		
Pattern		1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Abstract F.																									
Adap.	Class																								
	Obje.																								
Bridge																									
Builder																									
Chain R.																									
Command																									
Composite																									
Decorator																									
Facade																									
Factory M.																									
Flyweight																									
Interpreter																									
Iterator																									
Mediator																									
Memento																									
Observer																									
Prototype																									
Proxy																									
Singleton																									
State																									
Strategy																									
Template M.																									
Visitor																									

Table 1. Principles and their relationship with each pattern of those detailed by Gamma *et al.* (1995)

Several considerations, uses and investigation lines can be extracted. Some examples are the following:

- It allows us to break down each one of the patterns into smaller forces, facilitating the study of elements common to all the patterns of their own character: "patterns within the patterns" or "meta-patterns".
- It allow us to guide the use of patterns, since it is easier to know how to apply a pattern that a principle in a correct way, and once the principle is applied, it is easy to arrive at the pattern. This facilitates the pattern's good use. For example, the use of NSCP implies the use of creational patterns and this assures us that our system is written in function of interfaces and not in function of implementations.
- It allows a formal study of micro architectures.
- It allows us to obtain the forces (principles) that conform the pattern and how, depending in its manner of incidence within the pattern (type 1, 2 or 3), this can be of different characteristic. For example:
  - We can observe that Abstract Factory, Builder, Factory Method and Prototype maintain an almost identical kernel of principles while Singleton does not complete any principle. Singleton is not a micro architecture (it only describes one class). Singleton deals with the creation of objects but it does not do this with the same characteristic and the same abstraction as the other four creational patterns, we are according to Buschmann *et al.* (1996) whereon this pattern is an Idiom. With regard to

the four remaining creation patterns, we observe that they complete the same principles with the exception of Builder, since this has the same character as the previous ones but by means of a composition strategy. As we see, the study of the principles that intervene in a pattern allows us, among many other things, a finer and based classification.

- We observe that any micro architecture with some hierarchy whose design pattern we want to consider should complete (type 2) at least the following principles: OCP, SP, DIP, IDP and DCSP.
- We observe that in patterns structurally identical to State and Strategy the same principles are completed and with the same characteristics.
- All patterns that complete OCP, SP, DIP, IDP and DCSP in type 2, ISP and DAP in type 3 and do not fulfill the BBP it is classified (according to Gamma's book) as of behavior.

▪ We will be able to look for and/or to validate new design patterns observing whether they complete certain meta-patterns.

The principles allow us to extract good practical OO, observing how the patterns are based and how they are connected with the design.

2.2 Changeability issues

The element that provides changeability to the pattern is what it is called indirection. Nordberg (2001) comments that "at the heart of many design patterns is an indirection between service provider and service consumer. With objects the indirection is generally via an abstract interface". Unfortunately each level of indirection moves the software farther from the real world or analysis level view of the problem and deeper into relatively artificial mechanism classes that add overhead to both design comprehension and implementation debugging. With respect to the previous factors, we have observed the following:

- Every time that a pattern is introduced at least one indirection appears in the design and these elements are not of dominion or business logic, such as notifications, observer classes, updates methods, etc.
- Every time that we add an indirection the software moves around further away form the analysis. Upon adding indirections or design classes the design becomes less semantic, less comprehensible or less analyzable.
- Every time that an indirection is added it increases the design changeability.

3. Metrics for Optimal Patterns Number

With all the former, we have a problem: how much Changeability can we apply in order not to lose the design's Analyzability? Obtaining metrics to answer the previous question would be a great contribution.

We can define a parameter that quantifies how changeable a design is in relation to indirections:

$$\text{Changeability Number (CN)} = \text{Indirection Classes Number (ICN)} \quad (1)$$

On the other hand, a value that measures the design's analyzability must consider the number of design classes introduced: these are classes that simplify, reusing (such as the subject class into observer pattern) or indirection (such as the observer class into observer pattern). Thus:

$$\text{Analyzability Number (AN)} = \text{Domain Classes Number (DCN)} - \text{Indirection Classes Number (ICN)} - \text{Simplify Classes Number (SCN)} \quad (2)$$

We may observe in the last formula, that when we have an analysis diagram its analyzability is maximum. When we introduce artifacts into the designing phase on the analysis diagram the model's analyzability decreases. We also may observe, as certain patterns will have a larger impact in the analyzability than other ones, depending on the classes that the patterns introduce.

Now, we may calculate the Optimal Patterns Number (OPN) as follow:

$$\text{Changeability Number (CN)} = \text{Analyzability Number (AN)} \quad (3)$$

$$\text{Indirection Classes Number (ICN)} = \text{Domain Classes Number (DCN)} - \text{Indirection Classes Number (ICN)} - \text{Simplify Classes Number (SCN)} \quad (4)$$

$$\text{Indirection Classes Number (ICN)} = [\text{Domain Classes Number (DCN)} - \text{Simplify Classes Number (SCN)}] / 2 \quad (5)$$

Considering in the previous formula that the DCN parameter is a fixed value in design phase, the rest of parameters depend of the kind of pattern or design artifact introduced.

Shortly, we will add to the measures a bigger refinement level considering aspects such as the quality of methods.

3.1. Example

The following example (figure 2) shows us the Metrics for Optimal Patterns Number application (Changeability Number (CN) and Analyzability Number (AN)).

At first, we have three Domain entities (D1, D2 and D3). At this point we have not introduced some patterns, we do not have indirection classes, therefore, CH = 0. With three Domain Classes we have AN = 0 (we do not have Indirection Classes Number (ICN) or Simplify Classes Number (SCN), we only have Domain Classes Number (DCN)).

We introduce the Observer Pattern at a later moment (b in figure). This pattern has an indirection class (observer class), and this introduces one Simplify Class (subject class). With the previous CH = 1 and AN = 3 – 1 – 1 = 1.

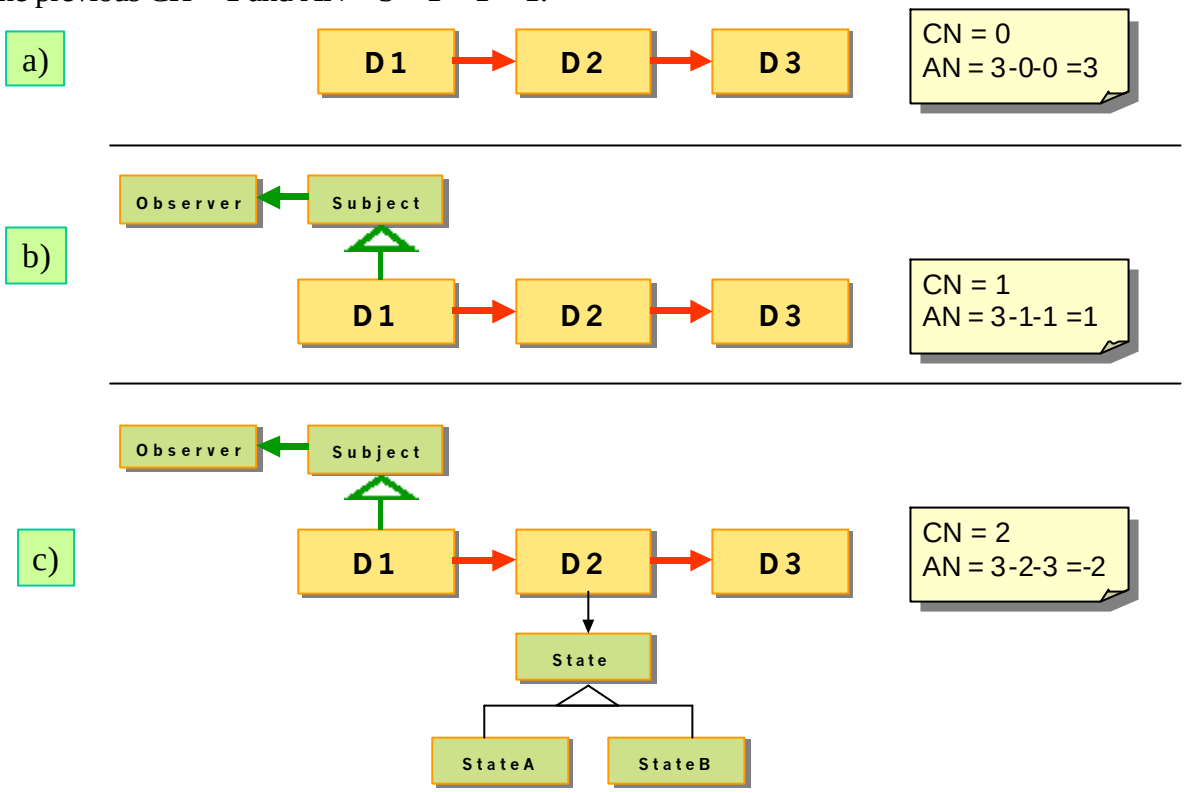


Fig. 2. This example shows the CH and AN variation at the moment of applying patterns

Finally, we introduce the State Pattern and this has an indirection class (State class) and, for our example, it introduces two Simplify Classes (StateA and StateB). With the previous CH = 2 and AN = 3 – 2 – 3 = -2.

At this moment the CH number is bigger than the AN number (figure 3). According to the former, at this moment we should not introduce more patterns if we want to maintain the analyzability of design. Perhaps at a later date it may be essential to add more patterns, but at this moment, if we are only improving the design, in a preventive phase, we do not have an explicit need to introduce a pattern. The relationship between CH and AN gives us a rational way to control the patterns' use.

4. Acknowledgments

This research is a part of the DOLMEN project supported by CICYT (TIC 2000-1673-C06-06). We would also like to thank to ALTRAN SDB for the support shown towards this investigation at all times.

5. Conclusion and future projects

The experts have always used proven ideas. It is in recent years when these ideas, materialized into the pattern concept, have reached their greatest popularity and diffusion, thanks to the concern of the community to discover, to classify and to diffuse all types of patterns.

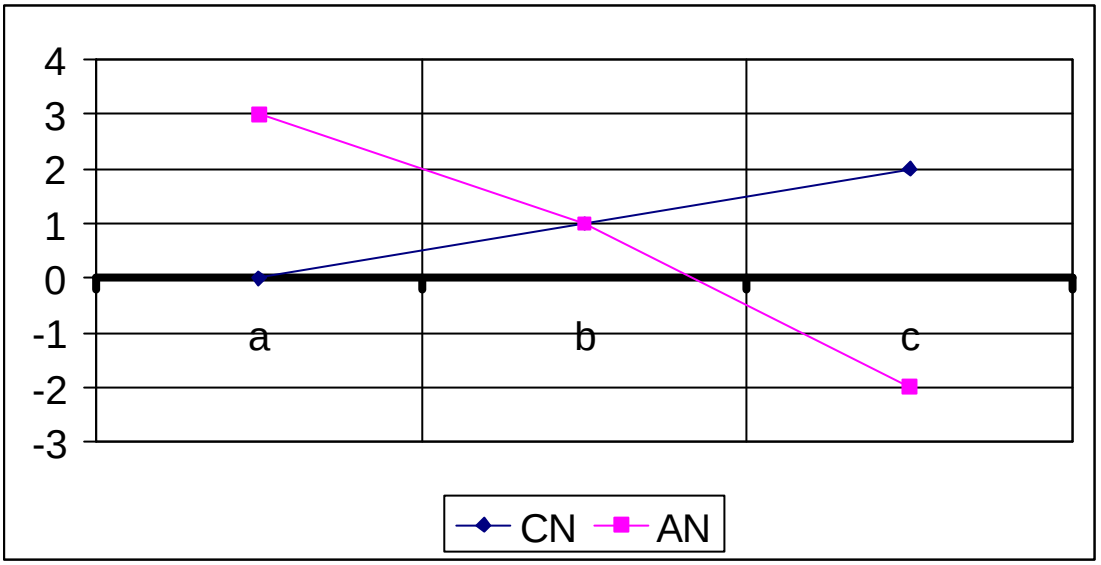


Fig. 3. The graph shows CN and AN evolution

The patterns are useful elements but there are still many elements to be studied if we want to apply them in a rational manner. The first thing that we must make clear is the word quality for design patterns; this word must be used with great care when we apply it to patterns. An appropriate notion of quality should result if the quality definition includes many views. In maintenance quality two appropriate views could be changeability and analyzability.

On the other hand, more knowledge exists apart from that related to patterns, although it would be true to say that this other knowledge is frequently "hidden". We denominate, distinguish and classify the following categories in OODK: principles, heuristic, patterns and refactorings (Garzás and Piattini, 2001). But there is much uncertainty with regards to the previous elements. In fact, the previous knowledge elements have never been studied as a whole, neither its compatibility has been studied nor does a method based on this knowledge exist. There is still a lot of work to be done in order to systematize and offer this OO Design knowledge to designers in such a way that it can be easily used in practical cases.

6. References

Bennet K. H. and Rajlich V. T. Software Maintenance and Evolution: a Roadmap, in Finkelstein A. (Ed.) The future of Software Engineering, ICSE 2000, June 4-11, Limerick, Ireland, pp 75-87.

Brito e Abreu F. and Carapuça R. Object-Oriented Software Engineering: Measuring and controlling the development process. 4th Int Conference on Software Quality, USA, 1994.

Briand L., Morasca S. and Basili V. Defining and Validating Measures for Object-Based high-level design. IEEE Transactions on Software Engineering, 25(5), 722-743, 1999.

Buschmann F., Meunier R., Rohnert H., Sommerlad P. and Stal M., A System of Patterns: Pattern-Oriented Software Architecture, Addison-Wesley, 1996.

Coad P., "Object-Oriented Patterns", Comm. ACM, Vol. 35, No 9, Sep. 1992, pp. 152-159.
Fowler M. Analysis Patterns. Addison Wesley, 1996.

Gamma E, Helm R, Johnson R and Vlissides J. Design patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, 1995.

Garzás J., Piattini M. Principles and Patterns in the Object Oriented Design, OOPSLA 2001 - Workshop "Beyond Design: Patterns (mis) used". Octubre 14-18, 2001. Tampa Bay, Florida, USA.

Genero M., Piattini M. and Calero, C. Early Measures For UML class diagrams. L´Objet. 6(4), Hermes Science Publications, 489-515, 2000.

Henderson-Sellers B. Object-oriented Metrics - Measures of complexity. Prentice-Hall, Upper Saddle River, New Jersey, 1996.

Nordberg M. E. Aspect-Oriented Indirection – Beyond OO Design Patterns. OOPSLA 2001 - Workshop "Beyond Design: Patterns (mis)used". Octubre 14-18, 2001. Tampa Bay, Florida, USA.

Prechelt L., Unger B., Tichy W. Bossler P. A controlled Experiments in Maintenance Comparing Design Patterns to Simpler Solutions. IEEE Transactions on Software Engineering, September 2000.

Pigoski, T. M. Practical Software Maintenance. Best Practices for Managing your Investements. Ed. John Wiley & Sons, USA, 1997.

Reibing R. *The impact of Pattern Use on Design Quality.* OOPSLA 2001 - Workshop "Beyond Design: Patterns (mis)used". Octubre 14-18, 2001. Tampa Bay, Florida, USA.

Rising L., The Patterns Handbook: Techniques, Strategies, and Applications, Cambridge University Press, 1998.
Schmidt D. C., Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software, Communications of the ACM 38,10, October 1995, pp 65-74.

Wendorff P., "Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project", Procedings of the Fifth European Conference on Software Maintenance and Reengineering , CSMR 2001, IEEE Computer Society.

Designing Websites by Using Patterns

Francisco Montero, María Lozano, Pascual González, Isidro Ramos†

LoUISE Research Group
Escuela Politécnica Superior de Albacete
University of Castilla–La Mancha
Campus Universitario
02071 – Albacete – Spain
http://www.info-ab.uclm.es/louise
{fmontero, mlozano, pgonzalez}@info-ab.uclm.es

†Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
E-46071 Valencia - Spain
iramos@dsic.upv.es

Abstract

This paper contains a resumed collection of patterns for designing web sites. Traditionally, the Web is mainly seen as a medium used to exchange information and that is the main reason why most web sites are designed like a book in which you can jump back and forth. Usability criteria should be considered when we are developing a web site [8]. This set of patterns is established under these criteria.

1. Introduction

The World Wide Web has rapidly become the dominant Internet tool, combining hypertext and multimedia to provide a network of multidisciplinary resources. It is important to make sure that all parts of a web site are useful. A user will come to a site expecting to be able to perform a particular task, or read a particular piece of information. When we are designing a web site we want to make sure that the user can find that resource quickly and easily. If they can't find the information quickly then they may leave our site, and proceed to another site where they can find the resource. The Web is a new medium and requires a new approach [8].

We should begin by asking, as in any user interface design process, Who are the users? and What are the tasks? But answer these questions is not easy. Anybody can visit our web site, kids, seniors, older or disabilities people. All information and resources should be accessible to them [3]. Everybody should be able to navigate with no problem. The idea is that, no matter what you're doing, there's a user-centred way of doing it. Users should be considered throughout the web site design process. Usability should not be an afterthought. Testing and fixing a web site *after* it has been built is inefficient and unlikely to produce good results.

Shneiderman [10] commented that "It will take a decade until sufficient experience, experimentation, and hypothesis testing clarify issues" and warned that meanwhile "the paucity of empirical data to validate or sharpen insight mean that some guidelines are misleading". Nevertheless, many sets of web design guidelines have been published. There are many guidelines [14] that can be used to improve the design of our web sites. Most of

these recommendations for web site designers are however not based on research but on intuition. They are based in the designer's experience.

Traditionally, interface design experiences are gathered with guidelines but patterns can be used too. The concept of a pattern language has been developed by Christopher Alexander and his colleagues in architecture and urban design [1, 2]. In brief, a pattern language is a network of patterns of varying scales; each pattern is embodied as a concrete prototype, and is related to larger scale patterns, which it supports, and to smaller scale patterns which support it. The goal of a pattern language is to capture patterns in their contexts, and to provide a mechanism for understanding the non-local consequences of design decisions [4].

2. A Web Design Patterns Language

This patterns language describes the usability-desired result unlike a prescriptive pattern language or guidelines. The patterns in this language are grouped into three levels.

- **Web site** level, people expect a web site to provide information, be interactive and fulfill their requirements. web sites have to be visually pleasing, download quickly, be helpful, easy to use and explore, in addition to presenting a professional appearance. A web site is made up of a collection of documents, images, sounds and other files. These usually reside in a unique place on the Internet. We can find this collection of web pages because they have an address.
- **Web page** level, a web page is a single page on any specific web site. A page is specially important, the home page, due to it identifies the web site.
- **Ornamentation** level. There are several elements that usually are on a web page. These elements give useful support to user for performing his/her tasks. These elements improve usability of the web site.

This taxonomy of three levels is based on [1]. The Alexander's language contains 253 patterns split into three broad categories, towns, buildings and construction. The first 94 patterns are collected together under the heading of "Towns". However, they cover far more than just urban planning and begin by examining the responsibilty of the designer from a global perspective. Moving on from the study of areas and collections of buildings, Alexander examines the buildings which make up the urban landscape and suggests over 100 patterns which define his position on what constitutes good building design. Alexander finishes his collection of patterns by defining the right way to construct buildings. The patterns in this final section of the language show how the need for careful thought and creative effort are required throughout the design process. The main objective of Alexander was improving *a quality without name* and planning cities, towns and buildings. Alexander wanted to create structures that are good for people and have a positive influence on them by improving their comfort and their quality of life. When we are developing a web site that's our goal too. We are interested in designing web sites with quality. But what is quality?. Quality is perhaps usability, understandability, learnability, operability, adaptability, accessibility, etc. We think that quality is everything mixed together.

The proposed web pattern language will be introduced by using a particular example. The patterns in the entire collection are depicted graphically in Figure 1 and summarised at the end of this section by using three tables, one of them for each level of patterns.

A problem is presented by each pattern and, under a context and a set of forces, a solution is proposed [7]. Problems are associated with user requirements and solutions look for improving usability web site. Usability is [ISO/IEC 9126-1] learnability, understandability and operability and these elements are improving by providing navigation, functionality, control, language, feedback, consistency, error prevention and visual clarity facilities.

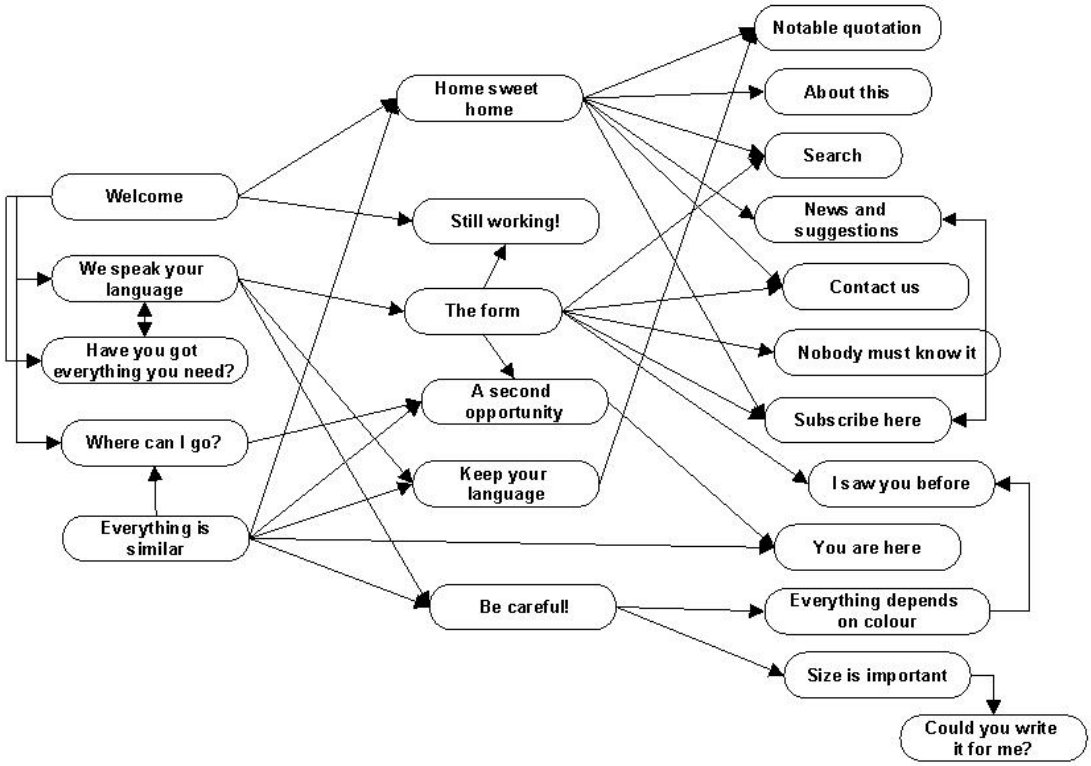


Figure 1. Proposed pattern language

Figure 1 shows the proposed pattern language. It has a structure of a network, in the left column there are patterns of web site level, in the centre column there are patterns of web page level and, finally, in the right column there are patterns of ornamentation level.

The following tables summarise the patterns in this pattern language for reference purposes. These patterns could be integrated on a methodology to develop user interfaces like IDEAS [5]. There are patterns at requirements level, like these, that can be used in the beginning of an usability-based iterative life cycle. So patterns can be used to improve a participatory design, evaluate web site under usability criteria and facilitate communication between stakeholders involved in web site development.

Web site patterns		
Problem	**Solution**	**Pattern name**
How does the user know where he is?	*Set up a reception point where user finds information about the web site*	Welcome
How does the user know where he can go and what will he find there?	*Web site musts provide the needs mechanisms that allow any user to move from one place to another places*	Where can I go?
How can the user do a	*Speak user's language is*	We speak your language

useful use of the web site and access information at your own pace?	*"design for all"*	
How does the user know where he is? How does the user know that he is visiting the same web site?	*Web site should be designed by using the same criteria: colours, fonts, navigation location and layout*.	Everything is similar
How does the user know if he needs anything else for visiting a web site?	*The user should be informed of what he need for visiting a web site*.	Have you got everything you need?

Web page patterns		
Problem	**Solution**	**Pattern name**
How does the user know where the user is?	*Provide a checkpoint where the user feels like at home*.	Home sweet home
How can the user access the content of the web page in a simple and proper way?	*Keep your language clean and inoffensive*.	Keep your language
How does the user know when his operations have finished or what is their current state?	*Show the user a status information of some kind, indicating how far along the process is in real time*.	Still working
How can the user visit the web site to his/her own pace?	*Provide return approaches*.	A second opportunity
How can the user provide preformatted information?	*Provide appropriate "blanks" to be filled in, which clearly and correctly indicate what information should be provided.*	The form
How can the user visit the web site without be interrupted or disoriented by unnecessary effects?	*Design for all people, technology and knowledge level in a possible manner*.	Be careful!

Ornamentation patterns		
Problem	**Solution**	**Pattern name**
How does the user know what the main feature of the web site owner is?	*Include a tag line that explicitly summarise what the site or company does*.	Notable quotation
How can the user get a suitable print of information?	*Provide a text version of web pages directly printable*	Could you write it for me?
How can the user have got access to additional and periodic information?	*Provide an approach to user cans book on-line*	Subscribe here
How can the user get additional information on products or documents?	*Include a "Contact Us" link on the homepage*	Contact us

How can the user find specified information?	*Offer a search engine*	Search
How does the user know where he/she has been?	*Use an approach to maintain state variables on the Web, like cookies*	I saw you before
How can the user access to the content of web site in a suitable way?	*Use suitably colours*	Everything depends on colour...
How can the user access the content of web site in a suitable way?	*Provide on-demand*	Size is important
How can provide private information in secure way?	*Implement security in web site*	Nobody must know it!
How does the user know what are the news in the web site?	*Include at homepage news and suggestions sections*	News and suggestions
How does the user know where he/she is?	*Include location references in web site*	You are here
How does the user know who the owner of web site is?	*Include a link to an "About Us" section*	About this

In the next section, we will describe some of these patterns by using them to solve problems related with the creation of a web site where usability degree should be high. This task helps us to introduce some of these patterns with more detail than this section.

When each pattern is introduced a last field, examples and implementation details, provides references (urls) and brief considerations on how this pattern can be recognised when we are navigating across the Web.

3. Example of use: Applying the Web Pattern Language

Suppose than you have to promote an institution, a business or a research group. An interesting possibility could be the creation of a web site where users can find out about services, products, latest news supplied by that institution.

You are a beginner designer and know that having a good web site is important, but you have not got enough experience in web design. There are guidelines, experiences and knowledge gathered in web site design field but it is difficult to use and in some occasions can be contradictory. However, a language is the most powerful tool of communication, but in a language there are words, sentences and relationships between words and sentences. On the other hand, patterns are a way of documenting design expertise. A pattern language can be useful in two senses: firstly as a tool to document the experience and secondly, as a tool to communicate between stakeholders of the project (end-user included).

How could I use this collection?

1. Read the resumed list of patterns.
2. Scan down the list, and find the pattern, which best describes the overall scope of the project or the problem that you want to solve.
3. Read the starting pattern. Tick all of the low order patterns and ignore all the high order patterns.

4. Turn to each pattern and now tick only relevant low order patterns.
5. Keep going like this, until you have ticked all the patterns you want for your project.
6. Adjust the sequence by adding your own material where you haven't found a corresponding pattern.
7. Change any patterns where you have a personal version, which is more relevant.

As an example, and after of reading the list of patterns, we could select some of them, for instance, Welcome, Home sweet home, Notable quotation, About this, Search, News and suggestions, Contact us, or Subscribe here. These patterns are related like shown figure 1 and they are of different levels. Then we can read them in more detail.

Welcome

Motivation:

When a user arrives at a web site, like he/she arrives at a city, town or any important building needs to know where he/she is, what can he do there, and what he need for visiting that web site.

Problem:

How does the user know where he/she is? How does the user know where he can go? Who does the site manage? What is the purpose of the site?

Forces:

- Users need know where they are
- User wants to know where they can go next
- A complex web site can be very disorienting for users
- Users who are familiar with the structure and content of a web site should be able to jump straight to the space where they want to go

Solution:

Set up a reception point where the user finds information about the web site. From this welcome point, user will be able to enter to home page (Home sweet home). User's information, such as language or monitor size should be gathered to the provision of web site's services to user (We speak your language). In its defect, the user should be informed about the best conditions for the visiting web site (Have you got everything you need?). User finds information about content (About this) and owner (Contact us) of the web site in this page. Welcome and homepage is the same in many occasions.

Consequences:

Provide improvements on the navigation, functionality and feedback

Examples & implementation details: http://www.aosa.es, http://www.alanismorissette.com These web sites, and many others on the Web, have got a initial page where users are received. These web pages have as main features their low-load time, offer the possibility to customise the language or browser properties, and provide information on who, what, when and where the user can find on the web site.

Home Sweet Home

Motivation:

A web site can be achieved by random way, but always must have a point of reference. When an user arrives at a web site, like he/she arrives at a city, town or any important building needs to know where he/she is, what he can do there, and what he need for visiting that web site. The homepage is an essential component of a web site. Questions such as: who?, what?, when? and where? should have answer on it.

Problem:

How does the user know where the user is? How does the user know where he will go? Who does the site manage? What is the purpose of the site?

Forces:

- Users need know where they are
- User wants to know where they can go next
- A complex web site can be very disorienting for users
- Users who are familiar with the structure and content of a web site should can jump straight to the space where they want to go

Solution:

Provide a checkpoint where the user feels like at home. Homepage is a place where the user can go back if he is disoriented. Its layout puts important information at top (News and suggestions), includes logos (Notable quotation), search approaches (Search) and information contact (Contact us, About this, Subscribe here) .

Consequences:

Provide improvements on the functionality, control and navigation

Examples & Implementation details: http://www.apple.com, http://www.ieee.org Any web site has a homepage. It is a specific page that introduces distinctive features. It has links to different sections of the web site, such as news, contact, about us, search etc. References to homepage should be included in every pages of the web site (A second opportunity).

Notable quotation

Motivation:
When you are designing a web site you should provide information about its purpose.
Problem:
How does the user knows what is the main feature of the web site owner?
Forces:
- Users are in a hurry
- Users don't read web pages, they have a look at pages

Solution:
Include a tag line that explicitly summarise what the site or company does. Its should be brief, simple and to the point. Include a short description of the site in the window.
Consequences:
Provide improvements on visual clarity, functionality and feedback
Examples & Implementation details: http://www.coolhomepages.com, http://www.bbva.es These web pages has images or taglines that implements this pattern. A tagline is a short phrase that communicates the "who" and "why' of your Web site.
The following elements create effective taglines: subject + audience + organization.

About this

Motivation:
All business web sites need to provide a clear way to find information about the company no matter how big or small the company is.
Problem:
How does the user know who the owner of web site is?
Forces:
- People like to know with whom they are doing business
- Getting company information might be the sole reason that users come to the site
- Many users want to know who is behind the service

Solution:
Include a link to an "About Us" section that gives users an overview about the web site owner and links to any relevant details about your products, services, company values, business proposition, management team, and so forth.
Consequences:
Provide improvements on functionality and feedback.
Examples & Implementation details: http://www.sunspot.net, http://www.ireland.com This pattern is implemented by adding a page or a section where information about owner of the page can be found. Normally a link to this section is situated in the homepage.

Search

Motivation:

Search is one of the most important elements of a homepage (Home sweet home), and it is essential that users be able to find it easily and use it effortlessly.

Problem:

How can the user find specified information?

Forces:

- User wants to know if the searched information is on the web site
- User doesn't read web site. He/she has a look at it.

Solution:

Offer a search engine. Give users an input box on the homepage to enter search queries, instead of just giving them a link to a search page [9]. Search on the homepage should search the entire site default [12].

Consequences:

Provide improvements on functionality and control.

Examples & Implementation details: http://www.paginasamarillas.es, http://www.microsoft.com This pattern is implemented by providing a search form. Search forms are the user interface of the search engine. It can consist on a very simple form with just a text field and a button, it can be a page and add a lint to it in your navigation. Advanced search capabilities can be worth adding. An advanced search page with options for phrases, multiple fields, special collections or zones, and date ranges allows them to perform more precise searches.

News and suggestions

Motivation:

Users want to know if there are new features in the web site. Users admit suggestions and want to know offers and promotions.

Problem:

How does the user know what the news in the web site are?

Forces:

- User doesn't read web site. He/she has a look at it
- Users are in a hurry

Solution:

Include news and suggestions sections at the homepage where users will have rapid access to new services offered by Web site

Consequences:

Provide improvements on functionality and navigation

Examples & Implementation details: http://www.microsoft.com, http://www.terra.es This pattern is implemented by placing latest news or suggestions in an outstanding place in the homepage (Subscribe here).

Contact us

Motivation:

All business web sites need to provide a clear way to contact with web site owner.

Problem:

How can the user get additional information on products or documents?

Forces:

- People like to know with whom they are doing business
- Getting company information might be the sole reason that users come to the site
- Many users want to know how is behind the service

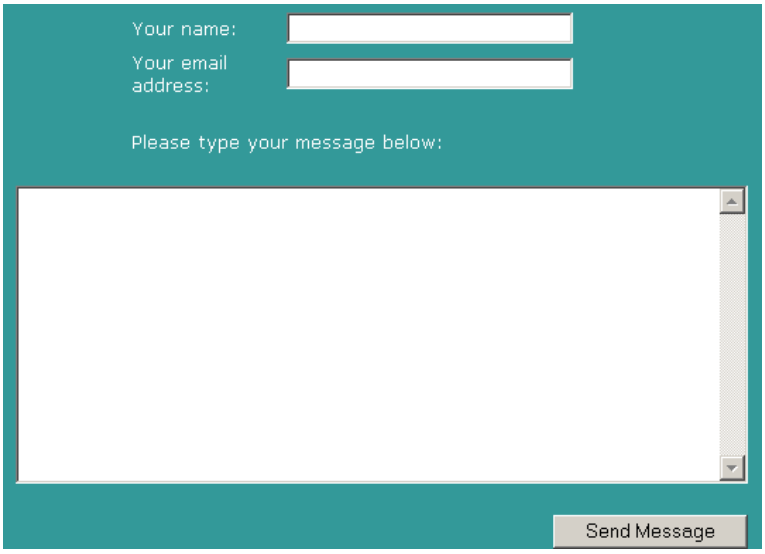
Solution:

Include a "Contact Us" link on the homepage that goes to a page with all contact information for your company (About this).

Consequences:

Provide improvements on feedback.

Examples & Implementation details: http://www.intel.com, http://www.lucent.com This pattern is implemented by including a page or a section where user can find contact information, in many occasions this information is included at the bottom of all the pages of the web site. In others cases, a form is provided to the user. This form contains features like a text area or text fields for the user can provide his email and other comments.



Subscribe here

Motivation:

Users want not to visit a web site everyday, they prefer to be informed when new products or news arrive.

Problem:

How can the user have got access to additional and periodic information?

Forces:

- User is in a hurry
- User wants to be informed

Solution:

Provide an approach where users can book on-line by providing an e-mail. So, the web site owner can send information to registered users about news and suggestions (News and suggestions).

Consequences:
Provide improvements on feedback.

Examples & Implementation details: http://www.prenhall.com, http://www.sun.es This pattern is implemented by using a simple form where user usually only have to provide an email. In other occasions is necessary provide more information related with the user's profile and preferences, so it is possible provide personalised information. Unsubscribe option should be provided too.

In this moment, we have a departure point to start the development of the web site. With a language, the redaction of pattern includes references to others patterns of low order. If in this moment, our beginner designer identifies the need of getting information from the user, could read The form pattern and related ones – Still working, Search, Contact us, Subscribe here, Nobody must know it and I saw you before. Next, some of them are explained

The form

Motivation:
The user has to provide information, usually short answers to questions

Problem:
How can the user provide preformatted information?

Forces:
- The user needs to know what kind of information to provide
- Users generally do not enjoy supplying information this way
- It should be clear what is required, and what is optional
- The user is in a hurry

Solution:
Provide appropriate "blanks" to be filled in, which clearly and correctly indicate what information should be provided [11]. Search, Subscribe here, Contact us are examples of forms. In occasions, a form fills a complete page. The user needs know if his/her submit was correctly processed (Still working). In some situations, we need to get confidential information of the users then must to provide additional security (Nobody must know it!)

Consequences:
Provide improvements on the functionality

Examples & Implementation details: http://www.iomega.com, http://www.iberia.es These web pages and some others like them, where the user can provide information implement this pattern. An HTML form is a section of a document containing normal content, markup, special elements called *controls* (checkboxes, radio buttons, menus, etc.), and labels on those

controls. Users generally complete a form by modifying its controls (entering text, selecting menu items, etc.), before submitting the form to an agent for processing (e.g., to a web server, to a mail server, etc.)

Particular examples of this patterns are Contact us, Subscribe here.

Still Working

Motivation:

Web sites are places where users can download information, images, files or applications, but this downloading can take a lot of time, create significant delays or be accomplished in different ways.

Problem:

How does the user know when his/her operations have finished or what is their current state?

Forces:

- The user wants to know how long they have to wait for the process to end
- The user wants to know how fast the progress is being made, especially if the speed varies
- Sometimes its impossible to tell how long the process is going to take

Solution:

Show the user a status information of some kind, indicating how far along the process is in real time. Images, files and any element that the user can download should have got information about size, so users can know how long have to wait for the download process. Images and text should be downloaded on-demand (Size is important).

Consequences:

Provide improvements on the functionality, feedback and error prevention

Examples & Implementation details:

http://www.google.com, http://www.acrobat.com Many web pages needs load a plug-ing to get a correct visualisation, a progress bar is used to provide such information. Sometimes a task running within a web site might take a while to complete. A web page with usability provides some indication to the user about how long the task might take and how much work has already been done. If you don't know or don't want to indicate how complete the task is, you can use a cursor or an animated image to indicate that some work is occurring. If, on the other hand, you want to convey how complete the task is, then you can use a progress bar like this one (http://java.sun.com):

Sometimes, you can't immediately determine the length of a long-running task. You can show this uncertainty by putting the progress bar in *indeterminate mode*. In this mode, the progress bar displays animation to indicate that work is occurring. In the Java look and feel, indeterminate progress bars look like this:

Nobody must know it!

Motivation:

If user provides private information, he/she will need to have the right to expect confidentiality. Rapid advances in communication technology have accentuated the need for security in the Internet.

Problem:

How can provide private information in secure way?

Forces:

- Users want to security
- Users do not need to know technical aspects

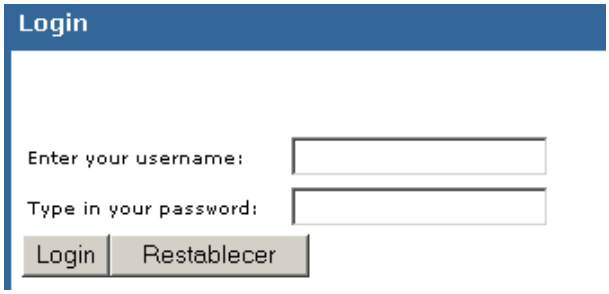
Solution:

Implement security in web site. Users should be registered in order to access to private sections on the web site, but sometimes only a login and password is not enough [13].

Consequences:

Provide improvements on feedback and control

Examples & Implementation details: http://www.bankofamerica.com, http://www.cdnow.com This pattern is implemented with login form where we'll ask the user for their username and password by using php or asp. But unless your form is located on a secure server, the information is transmitted in cleartext, and encryption won't occur until the php script runs.



I saw you before

Motivation:

When a user comes back to a web site he needs know what places he has visited, what documents he has downloaded and if there are modifications from last visit.

Problem:

How does the user know where he/she has been?

Forces:

- User does not want to loose his time
- User wants to receive personalised information

Solution:

Use an approach to maintain state variables on the Web like cookies. Since HTTP is a non-persistent protocol, it is impossible to differentiate between visits to a web site, unless the server can somehow mark a visitor.

Consequences:

Provide improvements on feedback and error prevention

Examples & Implementation details: http://www.kinkos.com, http://www.americanairlines.com This pattern is implemented by using cookies. Web cookies are simply bits of software placed on your computer when you browse web sites, so the web site will recognise the user's computer when he comes back to visit again. Cookies have some beneficial things. For example, when you log on or purchase online to certain sites, did you ever notice that when you return again you do not have to sign on the next time? That's because it stored your password and id on your machine in a cookie. So user's workload is reduced.

4. Conclusions

This paper presents a first approach of a web design pattern language. Its main goal is to gather the experience on web design and provides a communicative tool than can be used by every stakeholder in a project. The pattern language distinguishes between three design levels: the web site, a web page and the ornamentation. The recurring principle through the pattern language is supporting users to achieve usability improvement.

Web site patterns are associated with common features that can be found on many web sites and are extrapolated from another different context. The user requires know where he/she is (Welcome) and where he/she can go (Where can I go?). The user wants to visit the web site in a suitable way (We speak your language [6], Have you got everything you need?, Everything is similar).

Web page patterns introduce design patterns related with web page design. They are usual and considered features when we are designing web sites. In these hierarchical structure a homepage is necessary (Home sweet home). In some occasions, the user needs provide information then he/she must fill a form (The form) and always the user wants to have the control (Still working, A second opportunity) and to visit web sites to his/her own pace (Keep your language, Be careful!).

Ornamentation patterns introduce decoration features of a web site. These features provide improvements on the general usability of any web site. They are related with the use of colours (Everything depends on colour...), sizes (Size is important), security (Nobody must know it!) and providing location references (You are here, Contact us) and information (I saw you before, News and suggestions, Subscribe here).

Patterns can be used in web design to improve a human-centred design, as a communicative tool and as checklist to evaluate the usability of a web site, this last idea is in progress. This pattern language is been shepherd in this time in other paper that was sent to VikingPLoP.

5. Acknowledgements

This work is supported in part by the Spanish CICYT TIC 2000-1673-C06-06 and CICYT TIC 2000-1106-c02-02 grants.

6. References

[1]. Christopher Alexander. “A Pattern Language”, Oxford University Press, 1977.
[2]. Christopher Alexander. “The Timeless Way of Building”, Oxford University Press, 1979.
[3]. Constantine Stephanidis, Anthony Savidis. “Universal Access in the Information Society: Methods, Tools and Interaction Technologies.” Springer-Verlang. 2001.
[4]. Thomas Erickson, “Supporting Interdisciplinary Design: Towards Pattern Languages for Workplaces”. 1997. Http://www.pliant.org/personal/Tom_Erickson
[5]. María Lozano, Isidro Ramos, Pascual González. “User interface Specification and Modeling in an Object-Oriented Environment for Automatic Software Development”. IEEE 34th International Conference on TOOLS USA. 2000.
[6]. Fernando Lyardet, Gustavo Rossi. “Web Usability Patterns”. EuroPLoP, 2001. http://hillside.net/patterns/EuroPLoP2001/papers.html
[7]. Gerard Meszaros, Jim Doble, “A Pattern Language for Pattern Writing”, in Martin, Riehle, Buschmann, PloP Design 3. Reading, Mass: Addison-Wesley, 1994.
[8]. Jakob Nielsen, “Designing Web Usability: The Practice of Simplicity”. New Riders Publishing. 2000.
[9]. Jakob Nielsen, Marie Tahir. “Homepage Usability: 50 Websites deconstructed”. New Riders. 2002.
[10]. Ben Shneiderman. “Designing the User Interface: Strategies for Effective Human-Computer Interaction”. Addison Wesley. 1998.
[11]. Jenifer Tidwell. “Common Ground: A Pattern Language for Human-Computer Interaction”. http://www.mit.edu/~jtidwell/. 1998/99
[12]. Martijn van Welie. Interaction design patterns. http://www.welie.com/. 2001
[13]. Joseph Yoder, Jeffrey Barcalow. “Arquitectural Patterns for Enabling Application Security”. PloP’97 D-4 book. 1998.
[14]. Yale C/AIM WWW Style Manual http://info.med.yale.edu
Apple’s Web Design Guide http://applenet.apple.com
IBM Web Design Guidelines http://www.ibm.com/IBM/HCI/guidelines
Mary Evans. “Web Design: An Empiricist’s Guide”. University of Whasington, Seatle, Washington. 1998.

Software Decisions with Pattern Relations

Martin Auer Wolfgang Zuser Valter Vieira de Camargo

Vienna University of Technology
Institute of Software Technology
Research Industrial Software Engineering
1040 Vienna, Austria
{m.auer, zuser}@swt.tuwien.ac.at

University of São Paulo
Instituto de Ciências Matemáticas
e de Computação (ICMC/USP)
Cep 13560-970 São Carlos, SP, Brasil
valtercamargo@hotmail.com

Abstract

The concept of patterns is gaining widespread acceptance in the software community--in understanding domains, in reusing elegant solutions' designs and in communicating efficiently. Several types of patterns (for example, analysis patterns, design patterns or idioms) cover different aspects of software solutions. Other patterns describe important issues of the software development process.

Despite recent efforts, many patterns or pattern languages remain isolated and don't relate well to each other. While analysis patterns are widely used to support domain understanding, other powerful ways to use them remain unexploited.

In this paper, applications of patterns and idioms are described to support software decisions. First, existing analysis patterns are transferred to a new domain, and modified wherever necessary due to the target domain's requirements. Then, analysis patterns are related to concrete ways of designing and implementing working solutions, expressed by design patterns and idioms. Finally, these relations are used to support software decisions.

The proposed approach of translating existing analysis patterns to new domains and relating them to solution-oriented patterns substantially eases domain understanding and software decision making.

1. Introduction

The concept of patterns is becoming ubiquitous in software engineering and development. Originally introduced in [AIS77] in the context of architecture, it gained widespread acceptance with [GHJ95], which cover design patterns. Other pattern families include analysis, process and architectural patterns [Fow97, Amb98, BMR96].

Patterns are short, simple solutions to common problems arising time and again in comparable situations, or, as [CNM97] put it, "templates worthy of emulation". They help to

- understand domain-related problems;

- find existing and working solutions;
- communicate consistently and thus more efficiently by defining an instrumental vocabulary, a pattern language.

Yet many possible applications of patterns remain unexploited. This paper proposes the usage of patterns to

- efficiently reach domain understanding by applying existing patterns from similar domains and modifying them according to the target domain's requirements;
- connect analysis issues (expressed in analysis patterns) directly to design and implementation issues (design patterns, language idioms, or commercial off-the-shelf solutions) in order to collect and document possible design and implementation decisions along with their implications;
- support software decisions which have to take into account high-level and non-functional requirements as maintainability, understandability, stability etc.

This application of software patterns is being investigated in an on-going research effort at the Institute of Software Technology at the Vienna University of Technology, which tries to use analysis patterns and pattern relations to assess the domain of software measurement and to improve software measurement infrastructure with protocols and data formats.

Especially the design of protocols and data formats (as in this case) or the design of database structures which should be stable with regards to future changes can benefit from the proposed approach.

Section 2 points out related work. Section 3 describes the process of collecting a possibly extensive set of analysis patterns. Section 4 describes relations between patterns at different software development life-cycle stages. Section 5 shows the application of pattern relations with regard to software decision making. Section 6 summarizes the benefits of the proposed approach.

2. Related Work

[AIS77] defined many patterns in the context of architecture. [GHJ95] used similar templates in describing software design patterns. Other patterns families include analysis patterns [CNM97, Fow97], process patterns [Amb98] and architectural patterns [BMR96, HBH99].

This paper has several different points of contact to patterns-related issues.

1. *Domain understanding and coverage*. Analysis patterns have been used successfully to devise domain-specific applications and to share domain knowledge by acting as a common vocabulary [Fer98]. Examples include:
    - [Kel98] presents several patterns from the domain of insurance systems.
    - [WH98] describe analysis patterns for e-commerce transactions and proposes a component library based on these patterns.

- [Fer98] points out the similarity of a health-care solution to a commercial database application.

2. *Relations between patterns*. Right from the first pattern-related publications, patterns were described as to be closely related to other patterns. This fact was expressed in [GHJ95] by a dedicated section "related patterns" for each pattern's descriptions and by an overview chart describing common relations between different design patterns.

   [Zim95] goes one step further and organizes pattern relationships in different categories like "X uses Y in its solution", "variant of X uses Y in its solution", "X is similar to Y". Some limitations of existing approaches are:

   - Usually, relations are described between patterns of the same type (e.g. analysis patterns, design patterns,..) only.
   - Relations are described between patterns of the same publication only. Although recently authors tend to establish relations to existing patterns, these are mostly limited to well known patterns like those from [GHJ95].

3. *Making software decisions*. When building software systems, many decisions have to be made, for example, whether a solution should be "strong" (i.e., very specialized and efficient) or "weak" (i.e., more general and flexible, yet less efficient [VG98]). Other issues involve reusability, maintainability, simplicity, understandability etc. Especially architectural patterns [BMR96] consider typical high-level trade-off decisions.

   It remains difficult to reuse software decisions because of the strong dependency between the domain/initial requirements and these decisions. Even in the case of similar domains and requirements, software decisions usually can't be reused because of the lack of documentation and missing mappings to domain issues or requirements.

These three points of contact build the elements of a process which uses analysis patterns and pattern relations to support software decisions.

3. Create Analysis Pattern Set

[Fow97], who applies patterns from the health care domain to the corporate finance one and modifies them according to the specific needs, states: "By allowing patterns to migrate like this, I hope that more and more useful patterns will emerge, […]".

Indeed, many patterns proposed by [Fow97] can be transferred with little or no change to other domains.

This transfer of analysis patterns to new domains substantially eases the creation of an initial set of patterns which describe the new domain. The resulting patterns can then be modified according to the target domain's needs.

Thus, the creation of the analysis pattern set for a new domain, the target domain, can be performed with the following steps:

1. Identify similar domains. Many domains have a lot of similarities which allows adopting many of their analysis patterns to the new domain.
2. Identify those patterns which can be transferred to the new domain without any changes.
3. Identify those patterns which can not be transferred to the new domain due to the new domain's context.
4. Identify those patterns which have to be modified in order to be applicable to the new domain, modify them and document the rationale of the modifications.
5. Create new patterns which are not covered by the existing and modified patterns or which should replace the omitted ones.

This approach has several benefits:

- Working solutions are applied to similar problems (this benefit is the one usually obtained by applying patterns).
- Analysis issues are less likely to be forgotten or underestimated. For example, when going through a list of analysis problems from a similar, well-explored domain some analysis patterns which may not have been considered important or considered at all may come to the attention of the new domain's analyst.
- Using this approach several times, and updating the set of patterns over time makes it far more likely to reach a very extensive and maybe even complete coverage of the domain with analysis patterns.

By documenting the rationale of the transfer decisions, the new domain's analysis steps can be discussed and considered when changes or updates have to be made, maybe by different persons. A lot of background and domain information, as well as trade-off decisions are encoded in an easily accessible format for future usage.

Example

The step of creating a set of analysis patterns for the (target) domain of software metric collection is currently being investigated at the Institute of Software Technology at the Vienna University of Technology. In order to devise a set of "software metric analysis patterns", well-known analysis patterns from the domains of observation and measurement in health care and corporate finance (described extensively in [Fow97]) are transferred to the target domain. Some patterns are changed, if necessary, according to specific software metric domain requirements.

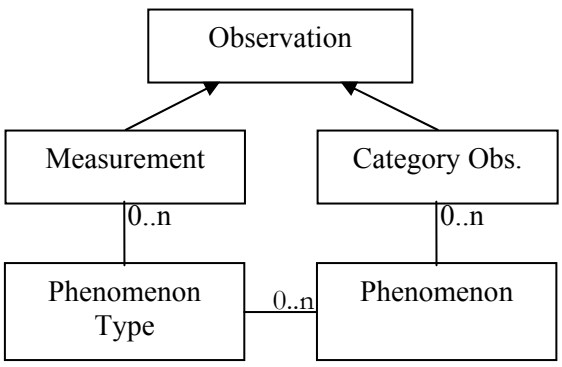
The main differences between the source and target domains (health care/corporate finance and software metrics, respectively) are:

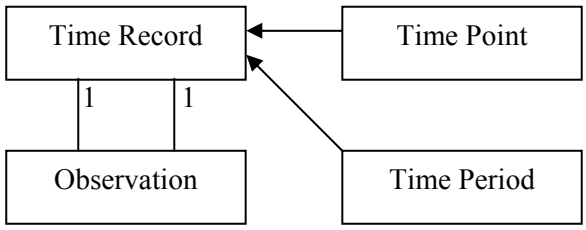
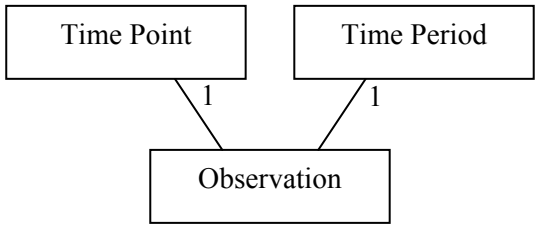
- The source domain environment is quite stable;
- The source domain has rigid restriction on data security and history;
- The target domain of software metric collection operates in the ever-changing and highly heterogeneous software development environment;
- The target domain has less rigid restrictions on data security and archives.

Despite the differences, the similarity of the two domains allows to reuse many existing patterns and considerations.

In our example, from the approximately 16 measurement patterns proposed in [Fow97],
- 4 can be transferred without changes (the basic patterns Quantity, Measurement, Protocol, Phenomenon With Range);
- 8-10 can be applied to the new domain with minor changes (for example, Conversion Ratio, Active Observation/Hypothesis/Projection);
- 2-4 can be omitted in the new domain (for example, the Subtyping Observation pattern).
- Furthermore, 4-6 new patterns have to be introduced to specifically address software metric issues not covered in the existing patterns (for example, to handle distance metrics like coupling between classes).

Examples for some transferred and/or modified patterns:

Pattern	*Explanation*
Observation, Category Observation	Fowler proposes several patterns to handle qualitative observations in addition to quantitative observations. We propose to reuse and transfer the pattern using Phenomenon objects [Fow97, page 45] to the target domain and omit the one using Category objects [Fow97, page 43]. A concrete occurrence of this pattern in the domain of software metrics would be the measurement of complexity. Complexity can be measured using McCabe's metrics (which would be an object of the class Measurement in the diagram below); alternatively it can be assessed "manually" by visually interpreting the graphical complexity of a program's flow graph and mapping this impression on an ordinal scale, i.e., to different perceived complexity levels like "simple", "complex", "very complex". The latter mapping can be expressed with objects of the class Phenomenon. Original pattern: 

	If rules of thumb are used to map McCabe values to complexity classes (for example, McCabe>15 => "very complex") then in order to specifically address this dependency between quantitative and qualitative observations, the Associated Observation pattern [Fow97, page 50] can be applied.
Dual Time Record	Fowler's original Dual Time Record pattern includes two Time Record objects: one for the applicability on an observation and one for the recording time. Each object can be either a Time Point or a Time Period object. We propose to explicitly make the applicability a Time Period and the recording time a Time Point object. This could avoid problems in the semantics of Time Point-type observation applicability. Original Dual Time Record pattern:  Modified Dual Time Record pattern:  In the original pattern's description, support for approximate time points is mentioned as potentially valuable (when for example medical information is recorded by the doctor, the patient might not remember the exact time point of a certain event). This should not be necessary in the (usually time-stamped) context of software artifacts.

A detailed list of the transferred analysis patterns and the changes made to them is shown in [Aue02b].

4. Patterns Relations

After constructing a possibly extensive set of analysis patterns for the domain, each pattern should be related to elements of the solution, namely to design patterns or language idioms. While there are several approaches to support the transition form analysis to design (even a pattern language is proposed to support it, see [Ker95]) we think that the simple concept of relation is sufficient to document possible analysis-design transitions.

We don't interpret the mathematical notation of "relation" strictly. We propose this metaphor to connect two or more elements of different sets, where the relation should express "Pattern A can be expressed or implemented by pattern or idiom or solution B in some way".

Examples
The analysis pattern Associated Observation [Fow97, page 50] records the chain of events in a diagnosis procedure. The idea of the pattern is to allow links between different observations (as an example, the link "thirst and weight loss indicate diabetes" is given).

Such associated observations can lead to different designs and implementations. First, let us take a look at different ways of *designing* the solution of this pattern:

- The analysis pattern Associated Observation could be expressed by the design pattern Observer [GHJ95] at the design level, yielding the analysis/design relation (Associated Observation, Observer).
  The Observer design pattern describes objects that are notified whenever the object they are depending on changes its state. One could think of a "Diabetes Observer" class which changes state when certain conditions are met in the primary diagnosis data.
- Another, less obvious design approach would be to use the design pattern Interpreter [GHJ95]. This pattern interprets languages defined by a representation grammar.
  One could express diagnoses as sentences of a grammar-based language, which serve as input to an interpreter. A "Diabetes Interpreter" would then try to assess whether certain conditions are met by a specific diagnosis sentence.

At the implementation level, there are many idioms that might be used to implement the analysis pattern.

- Triggers could be used at the database level to implement the pattern.
- Alternatively, it could be implemented using some Java idiom for dynamically interpreting rules or formulas encoded in strings. This way, a concrete associated observation would be defined in the software system by a rule or formula (a string like "if some_attribute='x' then diabetes_observation='true'") which is easy to customize, as it is not hard-coded in Java.

- Another way of implementing the concept would be to use some OLAP (online analytical processing) tool's dynamic formula capability. Hand in hand with this idiom, a whole solution context is proposed -- the use of a COTS tool instead of custom coding.

All relations express that the analysis patterns' solution part can be designed or implemented using a specific approach. Such relations describing possible future designs or implementations can affect software decisions in very early life-cycle stages (see next section).

An example of an application of pattern relations is the development of a metric data exchange format at the Institute of Software Technology at the Vienna University of Technology. Analysis patterns are related to XML idioms which can express the analysis patterns' structure. For example, there are dedicated data types in the XML schema language which support range information encoding and verification including whether a range or interval is open or closed[1]. This makes it the ideal idiom to be related to the Range analysis pattern (described in [Fow97, page 76]).

The transition between analysis and design/implementation has always been of interest to software engineers. Basing transition decisions on known sets of design patterns and idioms may ease this process.

Several benefits can be derived from mapping analysis patterns to possible design patterns and idioms early in the software life-cycle:

- People tend to implement solutions in their preferred languages and methods. If forced to document these transition decisions early, sub-optimal decisions can be easily detected in a very early life-cycle stage.
- The overall process (create a set of analysis patterns, then relate its elements to possible designs and implementations) lets developers proficient in different design paradigms and programming languages easily compare and weigh their different proposals.
- By documenting not only one proposed design or implementation, but also other possible variants, the factors affecting the implementation decision become an integral part of the documentation.
- The tracing of the domain model to its implementation details is improved. Changes to the domain model can be easily propagated into the design model and implementation details.

[1] The following XML schema snippet defines a range type based on the built-in primitive data type "integer" by setting the so-called *facets* "minInclusive" and "maxInclusive". Facets allow to customize schema elements in a wide variety of ways.

```
<xsd:simpleType name="myInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="1000"/>
    <xsd:maxInclusive value="9999"/>
  </xsd:restriction>
</xsd:simpleType>
```

Certainly there are some problems in this approach as well:

- Sometimes there will be no adequate design pattern for a certain analysis issue, especially if the analysis pattern is large and its solution design counterpart therefore unlikely to be of general interest--thus being no design pattern by itself. An example for such a large analysis pattern is the Portfolio analysis pattern [Fow97, page 183] consisting of 10 classes and 10 relations. Yet in such cases the solution design can usually be split up into several smaller parts of general applicability, into real design patterns. One part of the Portfolio pattern, for example, expresses a range of dates, which can be mapped to an XML range idiom (see the example above).
- The design patterns given in [GHJ95] can't cover all analysis patterns. Many other pattern languages must be considered as well, but collecting, learning, refining and classifying the large number of patterns is an immense task--in fact, we very much doubt if there will be anything close to a "unified design pattern language" any time soon. Yet notwithstanding such gaps, existing pattern sets offer a good starting point for possible transitions, whereas a gap might indicate the need for a new design pattern.

Another interesting issue in this context is how non-functional requirements can be expressed with pattern relations. Although some analysis patterns are concerned about later non-functional implications of the model they provide (for example, in the description of the Measurement analysis pattern [Fow97, page 41] it is indicated that this pattern's approach can avoid a class to expose too large a number of operations in its interface), they mostly focus on structural or dynamic domain aspects from a functional point of view.

Design patterns and idioms, on the other side, usually are aware of such non-functional issues like performance, understandability and maintainability.

By using relations between analysis patterns striving for a simple model on one hand and design patterns or idioms solving problems under constraints on the other, non-functional implications can be attached to the analysis model.

5. Software Decision Support

Once a set of pattern relations is obtained, it can be used effectively to support early life-cycle software decisions.

A first step might be to depict the relations between analysis patterns to design patterns and idioms expressing similar problems or to idioms expressing the same problem in different programming languages in a tabular form. This can point out sets of design patterns and/or idioms according to several criteria:

- Are there any analysis patterns that can only be implemented with one single implementation idiom or design solution? Are there any idioms, whose related analysis patterns can all be expressed with other idioms as well?

- What is the minimal set of design patterns and idioms that cover (i.e., express, implement) all analysis patterns?
- How many design patterns and idioms are necessary in different contexts (for example, using two different programming languages or object frameworks) to express all analysis patterns?
- Will newly introduced analysis patterns (maybe some already known patterns for the next release of the software) still be expressible with the set of idioms or design solutions? Will a database design be stable when requirements slightly change? Can a data format or protocol encode all identified analysis issues?

Example
Pattern relations between analysis patterns and XML idioms are currently being used at the Institute of Software Technology at the Technical University of Vienna to specify and formally verify the expressiveness of the metric data exchange format SIMDEF (whose initial draft is described in [Aue02])[2].

The analysis patterns obtained by transferring and modifying existing patterns, as well as by introducing new ones are related to XML idioms able to encode software metric data structures. The final version of the metric data exchange format SIMDEF will be a subset of all possible idioms and strives to encode metric data in a light-weight and human-readable way.

Several high-level requirements influence the final software implementation decisions:
- Although a very small set of XML idioms could be used to express all software measurement value types, or patterns, some additional idioms are likely to be used to enhance the formats understandability, for example, by specifically supporting the common case of multiple selection measurement values, instead of modeling it using a set of single selection values.
- Those XML idioms are preferred that can easily and transparently be mapped to flat database structures for later analysis operations.
- Those XML idioms are preferred that support automatic verification of data formats, therefore XML schemas are used to define the data structure.

---

[2] SIMDEF (SImple Metric Data Exchange Format) is an XML data format defined entirely by XML schema expressions to exchange typical metric information (like work report data, questionnaire data, etc.) between *metric data providers* (like IDEs, static source code analyzers,..) and a central *metric hub* which is implemented as a Web service to a central repository, a relational database. The data is communicated as SOAP (Simple Object Access Protocol) messages to the hub. The format and the corresponding protocol's technological choices are aimed to support heterogeneous and ever-changing software environments, thus reducing data collection costs.

6. Conclusions

The proposed approach of creating a domain-covering set of analysis patterns, possibly by starting off with patterns from similar domains, of relating analysis patterns directly to design or even implementation patterns/idioms which can express the analysis patterns, and of selecting those relations for implementation which satisfy some high-level requirements like reusability, maintainability, stability etc. has several benefits:

- Working solutions are used as a starting point (this is pattern intrinsic).
- Using this approach repeatedly in the same domain makes it more likely to reach a very extensive and maybe even complete coverage of a domain with analysis patterns.
- By documenting the rationale of the analysis to design/implementation transitions, the domain's analysis steps can be traced back, and domain information and trade-off decisions are encoded in an easily accessible format.
- Sub-optimal design decisions can be detected in early life-cycle stages.
- Developers with different backgrounds can easily compare and weight their different approaches.

Especially database structure or protocol design can benefit from this approach, which substantially enhances the construction of stable software solutions. The approach is currently being adapted to the domain of software metric collection at the Institute of Software Technology at the Vienna University of Technology.

7. References

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel, A Pattern Language, Oxford University Press, 1977

[Amb98] Scott W. Ambler, Process Patterns: Building Large-Scale Systems Using Object Technology, Cambridge University Press, 1998

[Aue02] M. Auer, Measuring the Whole Software Process: A Simple Metric Data Exchange Format and Protocol, Proc. of 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002), Málaga, June 11th, 2002

[Aue02b] M. Auer, Translating Measurement Patterns to Software Metrics, Tech. Report 02-08, Institute of Software Technology, Vienna University of Technology

[BMR96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Pattern Oriented Software Architecture - A System of Patterns, Wiley, 1996

[CNM97] Peter Coad, David North, Mark Maryfield, Object Models: Strategies, Patterns, and Applications, Yourdon Press, New Jersey, 2nd ed., 1997

[GHJ95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[HBH99] J. Hall, L. Barroca, P. Hall, editors, Software Architectures - Advances and Applications, Springer-Verlag, 1999

[Fer98] Eduardo B. Fernandez, Building systems using analysis patterns, in Proceedings of the third international workshop on Software architecture, ACM, pages 37-40, Orlando, Florida, 1998
[Fow97] Martin Fowler, Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997
[Kel98] Wolfgang Keller, Some Patterns for Insurance Systems, PLoP'98, *http://www.objectarchitects.de/ObjectArchitects/papers/index.htm*
[Ker95] Normal L. Kerth, Caterpillar's Fate: A Patterns Language for the Transformation from Analysis to Design, in: James O. Coplien, Douglas C. Schmidt, ed., Patterns Languages of Program Design, pp 293--324, Addison-Wesley, 1995
[VG98] Iris Vessey, Robert L. Glass, Strong Vs. Weak Approaches to Systems Development, CACM 41(4), pp 99--102, 1998
[WH98] Hans Weigand, Willem-Jan van den Heuvel, Meta-Patterns for Electronic Commerce Transactions based on FLBC, Hawaii Int Conf on System Sciences (HICSS'98), IEEE Press, 1998
[Zim95] Walter Zimmer, Relationships Between Design Patterns, in: James O. Coplien, Douglas C. Schmidt, ed., Patterns Languages of Program Design, pp 345--364, Addison-Wesley, 1995

Aplicabilidade da Família de Padrões de Reengenharia FaPRE/OO na Engenharia Reversa Orientada a Objetos de Sistemas Legados COBOL

Valter Vieira de Camargo
Fundação Educacional de Fernandópolis - FEF
valtercamargo@hotmail.com

Edson Luiz Recchia
PPGCC-DC-UFSCar / Universidade Anhembi Morumbi
erecchia@terra.com.br

Rosângela Penteado
Departamento de Computação
Universidade Federal de São Carlos
rosangel@dc.ufscar.br

Resumo

FaPRE/OO é uma família de padrões desenvolvida para conduzir processos de reengenharia orientada a objetos de sistemas legados procedimentais. Este artigo trata a aplicabilidade de forma evolutiva e complementar dos padrões relativos à fase de engenharia reversa a sistemas legados desenvolvidos em Cobol. Por evolutiva, entende-se que o processo de aplicação é em ciclos, isto é, padrões que são aplicados no início do processo, podem ser reutilizados posteriormente, a fim de aumentar o entendimento e refinar os produtos obtidos. Por complementar, entende-se que os padrões podem se complementar uns aos outros para a resolução de um problema. O processo de engenharia reversa é exemplificado pela aplicação em um sistema de controle de estoque, implementado em Microfocus COBOL 85, com 74 Kloc.

Abstract

FaPRE/OO is a pattern family developed to conduct object oriented reengineering processes of procedural legacy systems. This paper deals with the applicability, in an evolutionary and complementary form, of the patterns related to the reverse engineering phase, to legacy systems developed in COBOL. As evolutionary we mean that the application process is in cycles, that is, patterns that are applied at the beginning of the process, can be reused in subsequent phases, so as to improve understanding and refine the products obtained. As complementary we mean that the patterns can complement each other for the solution of a problem. The reverse engineering process is exemplified through its application to an inventory control system, implemented in Microsoft COBOL 85, with 74 Kloc.

1. Introdução

A engenharia reversa consiste em analisar um sistema existente, identificando seus componentes e representando-os em um nível mais alto de abstração [6]. Existem duas alternativas para realizar a engenharia reversa orientada a objetos de um sistema legado procedimental, como pode ser visto na Figura 1. A primeira, mais tradicional, é realizar a engenharia reversa procedimental do sistema legado e, a partir dos resultados dessa atividade, efetuar a engenharia avante orientada a objetos. Em outras palavras, na primeira fase obtém-se a documentação procedimental e, na segunda, com base nela, constrói-se a documentação de análise orientada a objetos. A documentação obtida na primeira fase reflete exatamente como o sistema legado está implementado e, mesmo que inconsistências sejam encontradas, não devem ser solucionadas, pois o serão na segunda fase. A segunda alternativa, que foi objeto de vários trabalhos já publicados [1, 2, 3, 4, 9, 10, 11, 12], é realizada identificando-se diretamente possíveis objetos no código legado procedimental. Dessa forma, busca-se, logo no início, identificar e solucionar possíveis inconsistências do código legado. Não se obtém documentação

que represente exatamente como o legado está implementado, mas sim uma documentação que está mais próxima da documentação orientada a objetos final.

A Família de Padrões para reengenharia orientada a objetos de sistemas legados procedimentais (FaPRE/OO), proposta em [14], possui padrões para conduzir a engenharia reversa, que resulta em modelos de análise orientados a objetos. O exemplo mostrado pelos autores da FaPRE/OO utiliza a primeira alternativa da Figura 1, enquanto que este trabalho utiliza a segunda alternativa, por meio de uma variação na ordem em que os padrões da FaPRE/OO são aplicados.

Assim, o objetivo deste trabalho é de mostrar que a FaPRE/OO pode ser utilizada de forma evolutiva e complementar, para obter o modelo de análise orientado a objetos diretamente a partir do sistema legado procedimental. Além disso, é feita a aplicação dos padrões da FaPRE/OO, instanciando-os para o caso específico da linguagem COBOL.

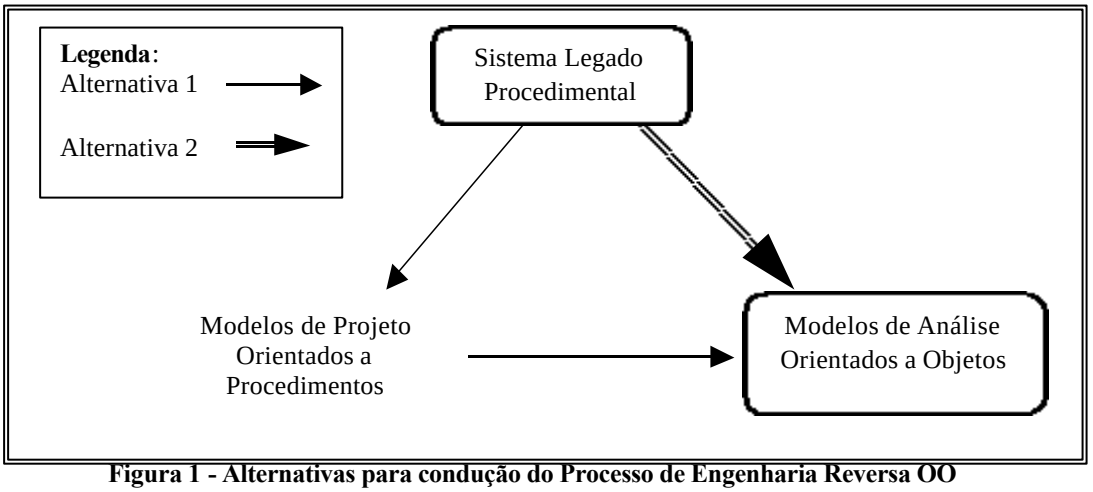


Figura 1 - Alternativas para condução do Processo de Engenharia Reversa OO

Este trabalho está organizado da seguinte forma: na Seção 2 apresenta-se, resumidamente, os padrões para a realização da engenharia reversa da Família de Padrões para a Reengenharia Orientada a Objetos de Sistemas Legados Procedimentais, FaPRE/OO. Na seção 3, o uso dos padrões da FaPRE/OO é exemplificado para um sistema de controle de estoque implementado em COBOL, e na Seção 4 são apresentadas as considerações finais.

2. Família de Padrões para Conduzir Processos de Reengenharia Orientada a Objetos de Sistemas Legados Procedimentais

Os padrões da FaPRE/OO foram divididos em *clusters,* cada um agrupando os padrões relacionados a situações similares. A Figura 2 ilustra graficamente os *clusters* e os padrões existentes em cada um deles. No primeiro *cluster*, "Modelar os Dados do Legado" extraem-se informações a partir dos dados e do código fonte do sistema legado gerando o MER - Modelo Entidade Relacionamento (visão procedimental dos dados) e o MASA - Modelo de Análise do Sistema Atual - Diagrama de Pseudo-Classes (visão orientada a objetos dos dados). Esses padrões guiam o engenheiro de software quando se tem o primeiro contato com um sistema legado. Fazem parte desse *cluster* os seguintes padrões:

- Iniciar Análise dos Dados
- Definir Chaves
- Identificar Relacionamentos

- Criar Visão OO dos Dados

No segundo *cluster* "Modelar a Funcionalidade do Sistema", os padrões são agrupados para obter a funcionalidade do sistema, criando modelos que recuperem as regras de negócio da empresa contidas no sistema legado. Esses padrões habilitam o engenheiro de software a obter um entendimento detalhado dos componentes do sistema, aprofundando sua compreensão sobre o mesmo. Fazem parte desse *cluster* os seguintes padrões:

- Obter Cenários
- Construir Diagramas de Use Cases
- Elaborar a Descrição de Use Cases
- Tratar Anomalias

No terceiro *cluster*, "Modelar o Sistema Orientado a Objetos", padrões são agrupados para se obter o diagrama de classes e os diagramas de seqüência do sistema, através da interação dos produtos obtidos pelos padrões dos *clusters* anteriores. Esses padrões habilitam o engenheiro de software a obter o MAS-Modelo de Analise do Sistema, sendo o modelo orientado a objetos a servir de suporte ao processo de reengenharia. Fazem parte desse *cluster* os seguintes padrões:

- Definir as Classes
- Definir Atributos
- Analisar Hierarquias
- Definir Métodos
- Construir Diagramas de Seqüência

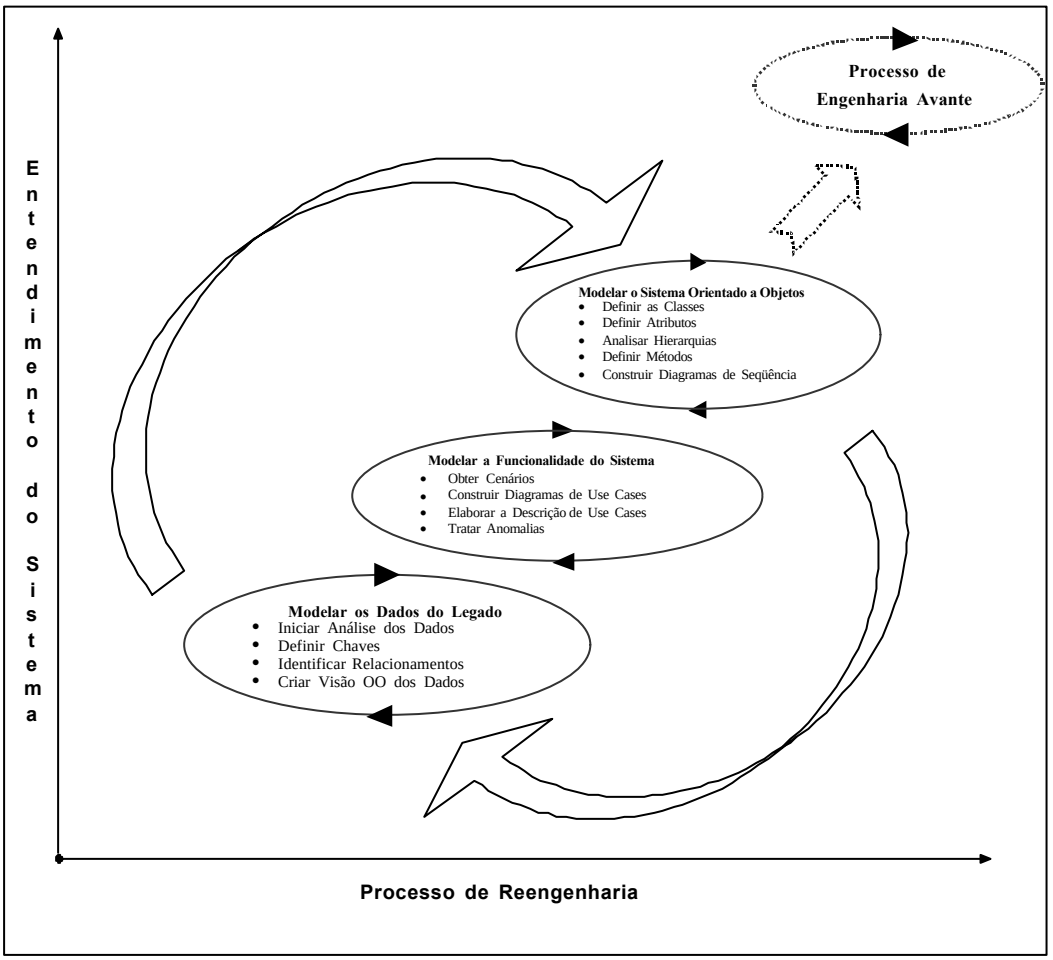


Figura 2 - Padrões do Processo de Engenharia Reversa da FaPRE/OO

Como a FaPRE/OO é para a reengenharia, a Figura 2 apresenta quatro *clusters*, sendo que os três primeiros são para a engenharia reversa e o último para a engenharia avante, completando o processo de reengenharia. Como o enfoque deste trabalho é apenas a engenharia reversa, somente os três *clusters* iniciais é que são detalhados e que serão apresentados nas próximas seções.

3. Estudo de Caso

Utilizou-se como estudo de caso um sistema de controle de estoque, implementado em COBOL *Microfocus* 85, composto por 498 módulos, sendo que 69 são programas e o restante *CopyFiles*[1]. O sistema possui setenta e quatro (74) Kloc e sua função principal é cadastrar materiais, fornecedores, contas, previsão de compras e alguns documentos, como: requisição de material, solicitação de material, comunicado de recebimento, devolução de materiais ao fornecedor e correção de estoque físico. Esse sistema faz parte de um maior, que integra contabilidade (quatrocentos e doze (412) Kloc), e folha de pagamento (cinqüenta e dois (52) Kloc). O engenheiro de software responsável pela aplicação dos padrões não conhecia e nem teve contato com os mantenedores e/ou desenvolvedores do sistema.

O objetivo desse estudo de caso é comprovar a viabilidade de aplicação da FaPRE/OO de forma evolutiva e complementar. Como evolutiva entende-se que o processo de aplicação é em ciclos, isto é, padrões que são aplicados no início do processo, podem ser utilizados novamente, em fases posteriores a fim de elevar o entendimento e refinar os produtos obtidos. Como complementar entende-se que os padrões podem interagir entre si para a resolução de um problema.

Os resultados obtidos com a aplicação da FaPRE/OO durante o processo de engenharia reversa são comentados a seguir. Cada sub-seção representa a aplicação de um dos seus padrões.

3.1. Padrão: Iniciar Análise dos Dados

Deve-se analisar os arquivos de dados do sistema e verificar os papéis que esses arquivos representam no mundo real. Geralmente um arquivo de dados possui um papel principal e pode ou não possuir papéis secundários. Cada um desses papéis deve ser considerado como uma entidade no DER (Diagrama Entidade Relacionamentos).

A identificação de papéis nos arquivos de dados do sistema faz com que se obtenha um DER sem inconsistências e redundâncias, eliminando, assim, essa preocupação em fases posteriores. Esse tipo de identificação foi utilizado, pois se objetiva avaliar a viabilidade da FaPRE/OO para a segunda alternativa apresentada pela Figura 1.

Se a primeira alternativa tivesse sido escolhida, o produto seria um DER representando o sistema legado procedimental como ele é. A identificação das possíveis classes (pseudo-classes), seria realizada pelo padrão Tratar Anomalias e Analisar Hierarquias. Quando da adoção da segunda alternativa, antecipam-se as consolidações que devem ocorrer posteriormente. Dessa forma, pode-se dizer que os padrões Iniciar Análise dos Dados, Tratar Anomalias e Analisar Hierarquias podem ser utilizados de forma complementar durante a análise dos dados.

A utilização desse padrão para o domínio da linguagem COBOL considerou as três formas de utilização do comando `REDEFINES` identificadas por Camargo [5]. A primeira forma sugere a criação de um relacionamento de agregação cuja cardinalidade do relacionamento é 1 para 1, pois, ocorre redefinição de um item elementar para um item de grupo simples. A segunda ocorre quando determinado item de grupo é redefinido em outro, fornecendo similaridade ao

[1] Trechos de código que são copiados para dentro de um programa durante a compilação.

comportamento de hierarquias de generalização/especialização da orientação a objetos. A terceira forma ocorre quando um item de grupo de nível geral é redefinido em outro, sendo assim, o registro possui dois comportamentos bem distintos e representa dois papéis bem distintos no mundo real. A Figura 3 apresenta um exemplo da segunda forma. O trecho de código mostra que o arquivo de dados COBOL possui como papel principal empregado, e como papéis secundários: secretaria, operador e engenheiro. Além disso, um relacionamento de herança pode ser identificado analisando-se que os itens elementares `NRO-EMP`, `SALARIO` e `ENDERECO` são gerais e comuns a todos papéis que o arquivo venha a assumir.

As tabelas obtidas como produto da aplicação desse padrão são:

a) Tabela 1, com três colunas, mostra na primeira coluna os programas que foram considerados durante a engenharia reversa; na segunda os arquivos de dados utilizados por cada programa e, na terceira, o papel principal de cada arquivo.
b) Tabela 2, mostra os papéis secundários que alguns arquivos possuem. Nota-se um número elevado de papéis secundários para o arquivo `FDES030`. Isso ocorre porque esse arquivo armazena dados de vários documentos e todos eles compartilham dados comuns. Para cada papel identificado uma entidade deve ser criada no DER.

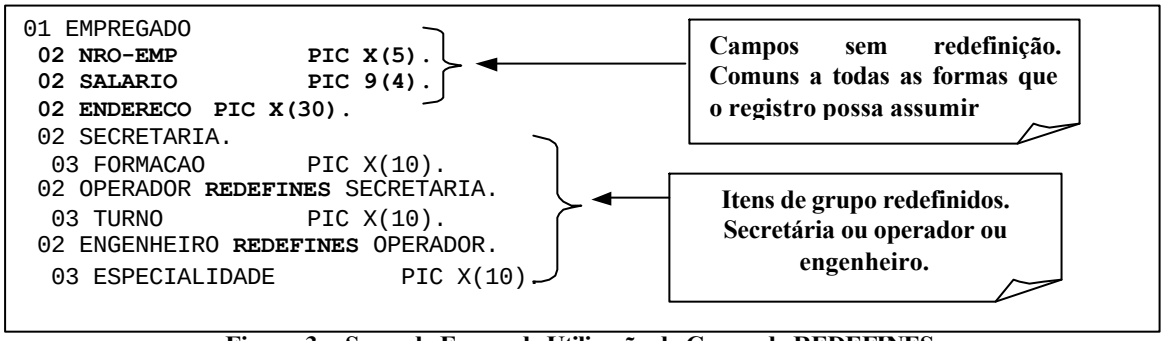


Figura 3 – Segunda Forma de Utilização do Comando REDEFINES

Durante a análise, pode-se identificar arquivos de dados cujo papel é relacionado à implementação e que não devem ser considerados como entidades do DER. Exemplos desse tipo de arquivos são aqueles que lidam com impressão de relatórios e a exibição de telas.

Tabela 1 – Papéis Principais

Programas	Arquivos	Papel Principal
ESFO2010	**FDES010**	Material
	FDES020	Conta
	FDSP020	Fornecedor
	FPES2010	Implementação
ESFO3000	FDES010	Material
	FDES030	Movimento
	FDES033	Movimento
	FDSA001	Conta de aplicação
	FDSA002	Conta de aplicação
	FDSP010	Compra
	FDSP020	Fornecedor
	FPES3000	Implementação
...	...	...

Tabela 2 – Papéis Secundários

Arquivos	Papéis Secundários
FDES010	Fornecimento
	Estoque
	Almoxarifado
FDES030	Estoque
	Almoxarifado
	Requisição de material
	Devolução de material
	Devolução de material ao fornecedor
	Transferência entre estoques
	Solicitação de material
	Correção de estoque físico
	Comunicado de recebimento tipo 1
...	...

Por meio de inspeções ao código notou-se que, uma das decisões de projeto do engenheiro de software responsável pelo desenvolvimento do sistema, foi a de não criar arquivos separados para estoque e almoxarifado. A solução foi adicionar campos representando essa funcionalidade em todos os arquivos do sistema como parte da chave primária. Porém, durante o processo de engenharia reversa com a aplicação da FaPRE/OO no sistema COBOL optou-se por

criar as entidades Estoque e Almoxarifado, por representarem diferentes papéis no domínio de aplicação do sistema. A identificação dos programas e arquivos de dados foi realizada com o auxílio computacional da ferramenta *Legacy Aid* [8].

3.2. Padrão: Definir Chaves

O padrão sugere a criação de apenas uma tabela, porém, optou-se por criar duas, uma para as chaves primárias e outras para as estrangeiras. Uma coluna adicional, com mnemônicos mais significativos para as chaves, guiou essa decisão.

Deve-se, primeiramente, analisar as entidades que foram geradas a partir da Tabela 1, que mostra os papéis principais. Para cada uma delas deve-se inspecionar o FD (*File Description*) que a gerou e, em seguida, a `INPUT-OUTPUT SECTION` do programa correspondente. Isso deve ser feito porque programas COBOL possuem, nessa seção, a declaração `RECORD KEY` que indica a chave primária do arquivo de dados. Dessa forma, identifica-se a chave primária para o papel principal do arquivo.

Após a análise de todas as entidades geradas, através de papéis principais, deve-se iniciar a análise das entidades geradas por papéis secundários. A identificação das chaves primárias para essas entidades consiste em analisar o trecho de código da FD responsável pelo papel secundário e inferir uma chave primária com base na sua funcionalidade em relação ao papel principal que o arquivo possui.

A Figura 4 apresenta um trecho de código do `FDES030`, um FD que possui como papel principal o movimento de materiais e, como papéis secundários, outros documentos apresentados pela Tabela 2, entre eles, Requisição de Material, delimitado com linha tracejada na Figura 4. A chave primária para o papel principal é o item de grupo `ES030CHAVE-PRINCIPAL`, que pode ser identificado analisando-se a `INPUT-OUTPUT SECTION` para esse FD. Porém, identificar uma chave primária para o item de grupo `ES030-RM`, que representa um papel secundário desse arquivo, necessita de análise adicional. Esse papel corresponde a uma requisição de material que só existe se o papel principal existir. Isto é, pode-se inferir que uma requisição de material é uma entidade fraca que depende da existência da entidade Movimento, representada pelo papel principal do arquivo. Sendo assim, a chave primária da entidade gerada pelo papel secundário é a chave primária da entidade gerada pelo papel principal do arquivo.

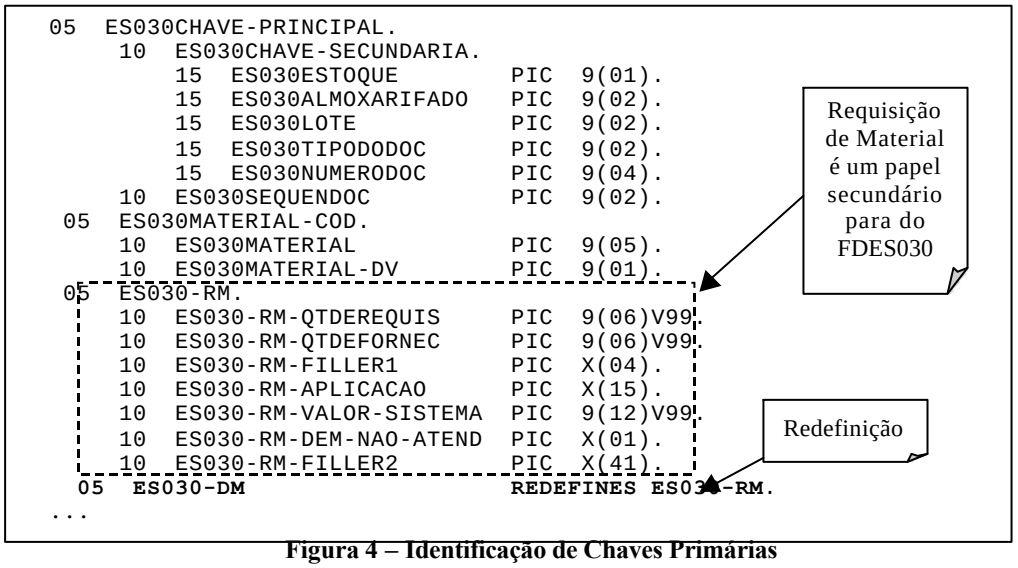


Figura 4 – Identificação de Chaves Primárias

Porém, outros fatores também podem influenciar na identificação da chave primária para uma entidade gerada por meio de um papel secundário. Optou-se por definir como chave primária da entidade `RequisicaoMaterial` os itens elementares `ES030LOTE`, `ES030TIPODODOC` e `ES030NUMERODOC`. Nem todos os itens elementares da chave primária do papel principal foram utilizados para definir a chave do papel secundário. Isso ocorre devido ao conhecimento do engenheiro de software sobre o sistema que está sendo analisado. Inspeções no código e a análise da funcionalidade mostraram que `ES030ESTOQUE` e `ES030ALMOXARIFADO` são itens elementares existentes em todos os outros arquivos. Essa decisão de projeto foi utilizada para que não fossem criados arquivos que representassem essas funcionalidades. Sendo assim, como o DER criado conterá uma entidade `Estoque` e uma `Almoxarifado`, esses itens de dados podem ser desconsiderados. A última linha do trecho de código fonte apresentado pela Figura 4 mostra a redefinição do item de grupo que representa a requisição do material. Desse ponto em diante, um novo papel secundário é identificado.

O primeiro produto obtido da aplicação desse padrão é a Tabela 3. A primeira coluna exibe o nome das entidades identificadas, a segunda exibe as chaves primárias com a mesma nomenclatura do código fonte e, a terceira, apresenta as chaves com a nomenclatura alterada para mnemônicos mais significativos.

Tabela 3 - Tabela de Chaves Primárias

Entidades	Chaves Primárias	Mnemômicos Significativos
Material	ES010CHAVE	codigo
Produto Acabado	ES018CHAVE	codigo
PrevisaoCompra	ES080CHAVE	codigo
Fornecimento	ES010CHAVE + SP020CHAVE	cod_material + cod_fornecedor
Movimento	COD_ESTOQUE + COD_ALMOXARIFADO + NUMERO + TIPO_DOC + NUMERO_DOC + LOTE	cod_estoque + cod_almoxarifado + numero + lote
RequisicaoMaterial	TIPO_DOC + NUMERO_DOC + LOTE	tipo_doc + numero_doc + lote
...	...	...

O processo de identificação de chaves estrangeiras inicia-se com a análise de programas que contenham dois arquivos, depois três e assim sucessivamente. O fato de um programa lidar com mais de um arquivo é um forte indício de relacionamento entre eles.

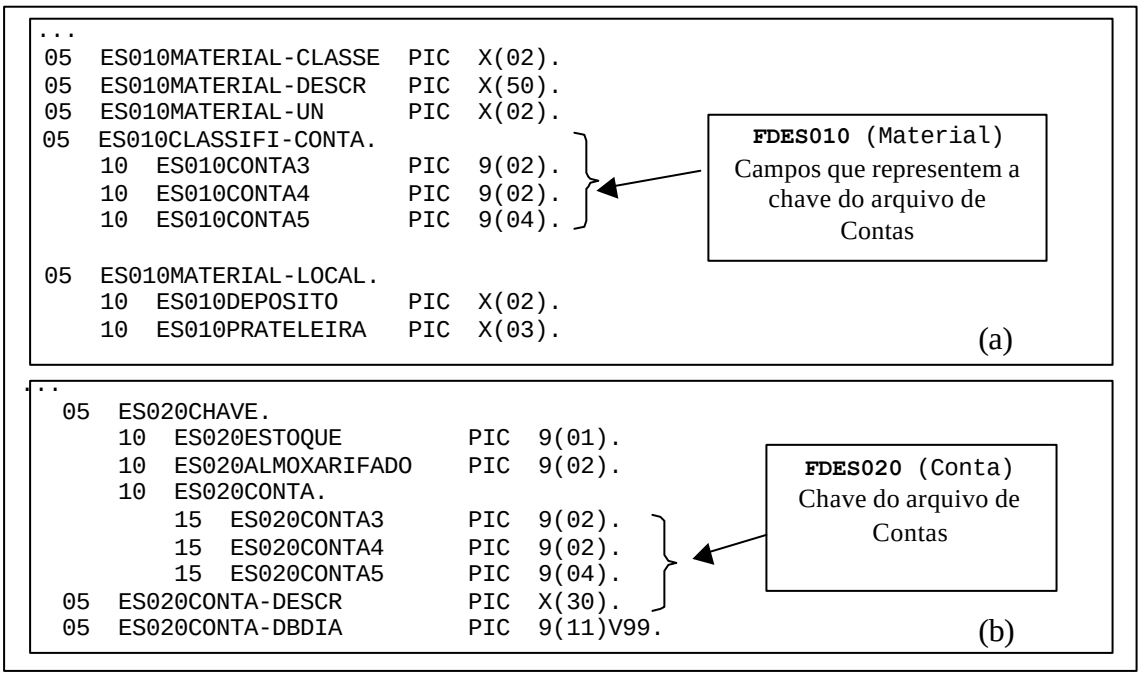


Figura 5 – Identificação de Chaves Estrangeiras

Deve-se iniciar o processo de análise para os papéis principais. Por meio da Tabela 1 pode-se identificar alguns dos programas que lidam com mais de um arquivo. O programa ESFO2010, responsável pelo cadastro de materiais, têm três arquivos: associados FDES010, FDES020 e FDSP020, que possuem como papel principal "material", "conta" e "fornecedor", respectivamente. Analisando-se o código fonte do FDES010 nota-se a presença de campos que representam, semanticamente, a chave primária do arquivo de contas, FDES020. A Figura 5 apresenta, na parte superior rotulada com a letra (a), o trecho de código do FDES010 e, na parte inferior rotulada com (b), o trecho de código do FDES020. Nota-se que no FDES010, cujo papel principal é material, há campos que provavelmente representam semanticamente a chave do arquivo FDES020, cujo papel principal é contas. Para certificar-se disso deve-se verificar se o tipo e o tamanho dos campos são os mesmos. Além disso, também se deve analisar o código fonte a fim de verificar se, em determinado ponto do fluxo de execução, há uma atribuição dos dados que representam a chave primária do arquivo de contas para os campos do arquivo de material que representam a chave estrangeira.

A análise do fluxo de execução foi realizada com o auxílio computacional de um recurso da ferramenta *Legacy Aid* [8] denominado *code Walkthrough*. Esse recurso permite simular a execução do código e, paralelamente, visualizar o grafo de fluxo de controle à medida que o controle passa de nó para nó. Esse recurso permite, também, que a interação do usuário respondendo sim/não em condições de seleção.

Tendo realizado o processo de identificação de chaves estrangeiras para os papéis principais, deve-se, em seguida, analisar as entidades que foram geradas por meio de papéis secundários. Como citado anteriormente, alguns papéis secundários darão origem a entidades fracas. Sendo assim, para esses tipos de papéis, o processo de identificação de chaves estrangeiras já está pronto. Porém, vale a pena revisar os trechos de código responsáveis pela geração de papéis secundários, para verificar possíveis chaves estrangeiras escondidas sob muitas decisões de projeto.

A Tabela 4 representa parte do segundo produto obtido da aplicação do padrão Definir Chaves para o sistema de controle de estoque COBOL. A primeira coluna tem o nome das entidades, a segunda, as chaves estrangeiras com a mesma nomenclatura do código fonte e, a terceira, as chaves estrangeiras alteradas para mnemônicos mais significativos.

Tabela 4 – Tabela de Chaves Estrangeiras

Entidades	Chaves Estrangeiras	Mnemônicos Significativos
Material	ES010CLASSIF-CONTA ES010NUMEROFORN	cod_conta cod_fornecedor
Produto Acabado	ES018CDMATERIAL	cod_material
Compra	SP010PREVISAO-COMPRAS SP010MATERIAL	cod_previsaoCompras cod_material
Fornecedor	---	---
PrevisaoCompra	ES080MATERIAL	cod_material
RequisicaoRessuprimento	SP050-MATERIAL SP050-PREVISAOCOMPRAS	cod_material cod_previsaoCompras
Movimento	ES030MATERIAL	cod_material
RequisicaoMaterial	ES030NUMERODOC ES030MATERIAL	numero_documento cod_material
...	...	...

3.3. Padrão: Definir Relacionamentos

Com base na tabela de chaves estrangeiras pode-se desenvolver o Diagrama Entidade Relacionamento (DER - Figura 6) com suas entidades e respectivos relacionamentos. A cardinalidade dos relacionamentos é obtida verificando o número de repetições de uma chave

estrangeira dentro de um arquivo de dados. O DER não representa a estrutura de arquivos existentes no sistema legado. Durante a aplicação do primeiro padrão já se preocupou em resolver inconsistências de projeto representadas no código fonte. Dessa forma, elimina-se carga de responsabilidade dos padrões posteriores. Outra vantagem disso é utilizar a segunda alternativa de realização da engenharia reversa, apresentada pela Figura 1.

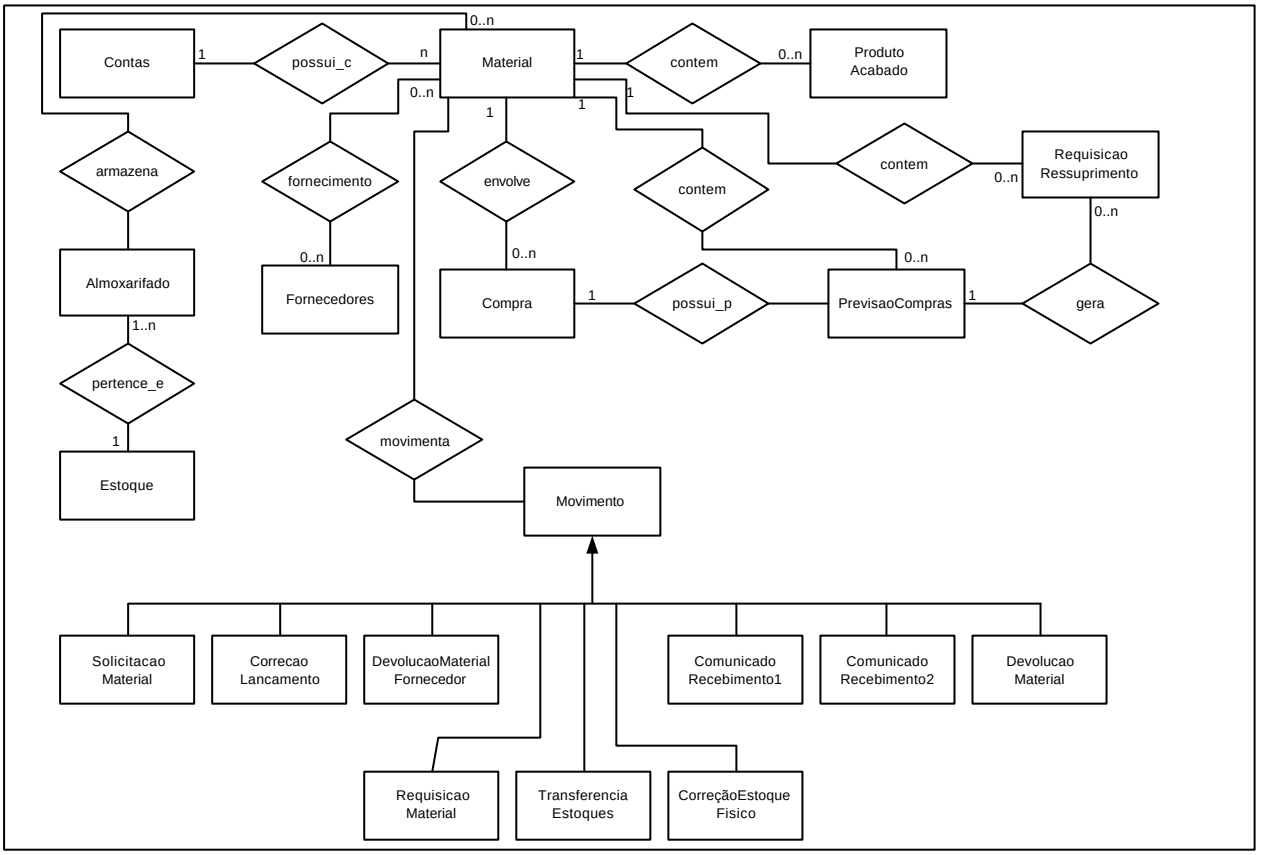


Figura 6 – Diagrama Entidade Relacionamento

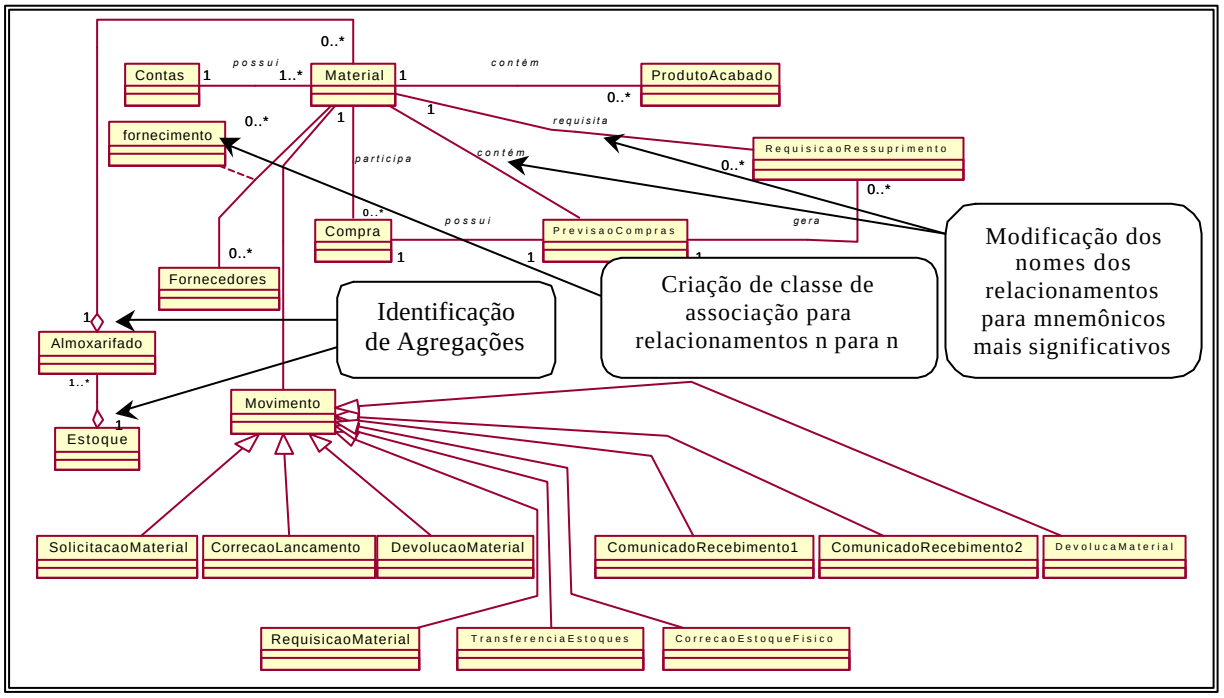


Figura 7 – Modelo de Análise da Solução Atual

3.4. Padrão: Criar Visão Orientada a Objetos dos Dados

Deve-se mapear cada entidade identificada anteriormente para uma pseudo-classe do Modelo de Análise da Solução Atual (MASA), como proposto pelo método de engenharia reversa Fusion/RE [9]. Durante o mapeamento, deve-se identificar possíveis relacionamentos de agregação e, também, criar classes de associação para relacionamentos com cardinalidade n para n. Além disso, também é aconselhável, sempre que possível, alterar o nome dos relacionamentos para mnemônicos mais significativos.

A Figura 7 apresenta o produto obtido da aplicação desse padrão. Utilizando a segunda alternativa de engenharia reversa mostrada pela Figura 1, esse pseudo-modelo de classes se aproxima do modelo final. Isso ocorre porque, no processo de resolução, as inconsistências e redundâncias existentes no código fonte procedimental são distribuídas ao longo de todo o processo de engenharia reversa e não são concentradas em poucas fases. A concentração de esforços em algumas fases de um processo aumenta a probabilidade dos engenheiros de software se desmotivarem e, conseqüentemente, realizarem tarefas mal feitas.

3.5. Padrão: Obter Cenários

Esse padrão foi aplicado como sugerido pela Família, não havendo necessidade de instanciá-lo para a linguagem COBOL, pois a maioria dos sistemas legados apresenta interfaces com menus e submenus. A Tabela 5 é parte do produto obtido da análise de interfaces do sistema legado COBOL.

Tabela 5 - Tabela de Cenários do Sistema

Cenários (Opções do Menu Principal)	SubOpções do Menu Principal
Atualização de Cadastros	Materiais Compras Previsão de Compras Fornecedores
Consultas	Cadastro de Materiais Arquivo do Movimento Cadastro de Fornecedores Cadastro de Contas Cadastro de Compras Cadastro de Especificações
...	...

3.6. Padrão: Construir Diagramas de Use Case

Esse padrão, como o anterior, foi aplicado como sugerido pela Família, não havendo necessidade de especializá-lo para a linguagem COBOL. Ele utiliza, de forma complementar, o padrão Iniciar Análise dos Dados, pois, os diagramas devem ser elaborados levando em consideração os papéis que são manipulados pelo sistema.

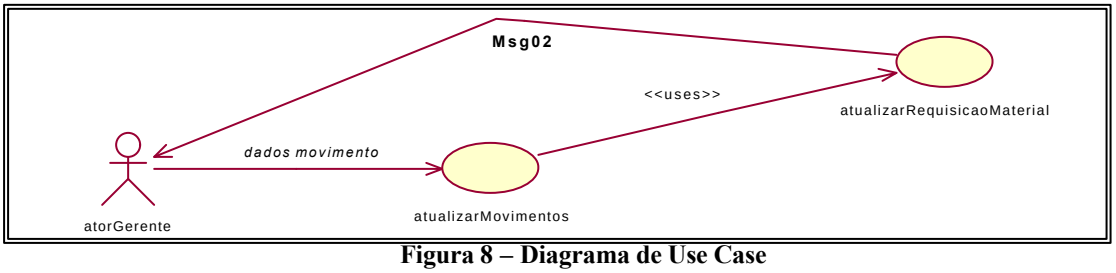


Figura 8 – Diagrama de Use Case

A Figura 8 apresenta um diagrama de *Use Cases*, especificado em UML, que demonstra a funcionalidade do sistema referente à movimentação dos materiais. Apesar do comportamento do *Use Case* atualizarMovimentos ser composto por vários outros *Use Cases*, esse diagrama apresenta apenas um, o atualizarRequisicaoMaterial. O diagrama mostra que o ator Gerente interage com o *Use Case* atualizarMovimento enviando a ele dados da movimentação do material. Esse, por sua vez, invoca o *Use Case* atualizarRequisicaoMaterial a fim de manipular dados da requisição.

3.7. Padrão: Elaborar a Descrição dos *Use Cases*

A descrição de cada *Use Case* identificado no padrão Construir Diagramas de Use Case deve ser elaborada analisando-se o código fonte correspondente a ele.

Esse padrão também utiliza o padrão Iniciar Análise de Dados complementarmente, pois, durante a descrição de um *Use Case* deve-se considerar os papéis que foram identificados anteriormente.

3.8. Padrão: Tratar Anomalias

Esse padrão considera a construção de uma tabela que solucione as anomalias encontradas no código fonte procedimental. A elaboração dessa tabela, denominada Detalhes de Implementação, leva em consideração a descrição dos *Use Cases* obtida no padrão Elaborar a Descrição dos *Use Cases*. Deve-se analisar se a descrição do *Use Case* faz acesso a mais de uma entidade (papel) no DER, pois se fizer, o princípio de encapsulamento da orientação a objetos está sendo violado e essa anomalia deve ser solucionada. A classificação de anomalias utilizada é a mesma definida pelo método Fusion/RE.

Como o sistema do estudo de caso foi desenvolvido utilizando o paradigma procedimental, é comum a existência de sentenças que fazem acesso a dados de diferentes papéis. A Figura 9 apresenta um trecho de código de um parágrafo que quebra o princípio de encapsulamento ao escrever (gravar) dados em dois arquivos distintos. O parágrafo de nome `10-00-INCLUSAO` utiliza o comando `MOVE` para mover o conteúdo da variável `REG10-MATCOD` para o campo `ES010MATERIAL`, do arquivo de dados `FDES010-FD`, e o conteúdo da variável `REG10-FORNECEDOR` para o campo `SP020FORNECEDOR`, do arquivo de dados `FDSP020-FD`. O *Use Case* correspondente a esse parágrafo será categorizado como C+ (`FDES010-FD/FDSP020-FD`).

É provável que um *Use Case* dê origem a tantos métodos quantos forem seus acessos a entidades diferentes, isto é, um *Use Case* que faz acesso a três arquivos será transformado no mínimo em três métodos um para cada entidade a que tem acesso. Os nomes dos novos métodos devem utilizar mnemônicos significativos e seguirem a padronização da UML [15].

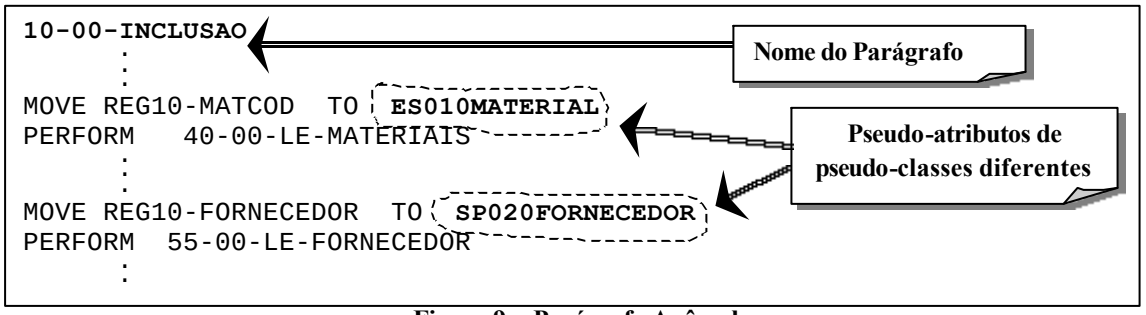


Figura 9 – Parágrafo Anômalo

A Tabela 6 apresenta parte do produto obtido da aplicação desse padrão. A primeira linha da tabela, destacada em negrito, mostra que o *Use Case* AtualizarMateriais, correspondente ao programa `ESFO2010`, possui acesso de leitura (O) a duas entidades e de escrita (C) a uma única entidade. Sendo assim, o código fonte correspondente a esse *Use Case* deve ser analisado a fim de solucionar a anomalia encontrada. A solução consiste em criar três métodos, com funcionalidades distintas de leitura e gravação e, colocá-los em suas classes correspondentes.

Vale ressaltar que esse padrão foi utilizado de forma complementar pelo padrão Iniciar Análise de Dados, pois a identificação das classes parte daqui.

Tabela 6 – Tabela Detalhes de Implementação

Use Case	Programas	Pseudo-Classes	Tipo de Anomalia	Possíveis Métodos	Pseudo-Classes Revisadas
atualizarMateriais	**ESFO2010**	**Material**	**c**	**incluir**	**Material**
		Contas	**o**	**selecionarConta**	**Conta**
		Fornecedor	**o**	**selecionarFornecedor**	**Fornecedor**
AtualizarRequisicaoMaterial	ESFO3000	Material	o	selecionarMaterial	Material
		Movimento	o	incluir	Movimento
		RequisicaoMaterial	c	incluir	Requisicao Material
atualizarConta	ESFO2020	Contas	c	incluir	Conta
...	...	...	...	...	...

3.9. Padrão: Definir as Classes

A elaboração do diagrama de classes do sistema possui como base o MASA obtido no padrão Criar Visão Orientada a Objetos dos Dados. A regra geral é que cada pseudo-classe no MASA seja mapeada para uma classe no Modelo de Análise do Sistema (Figura 10).

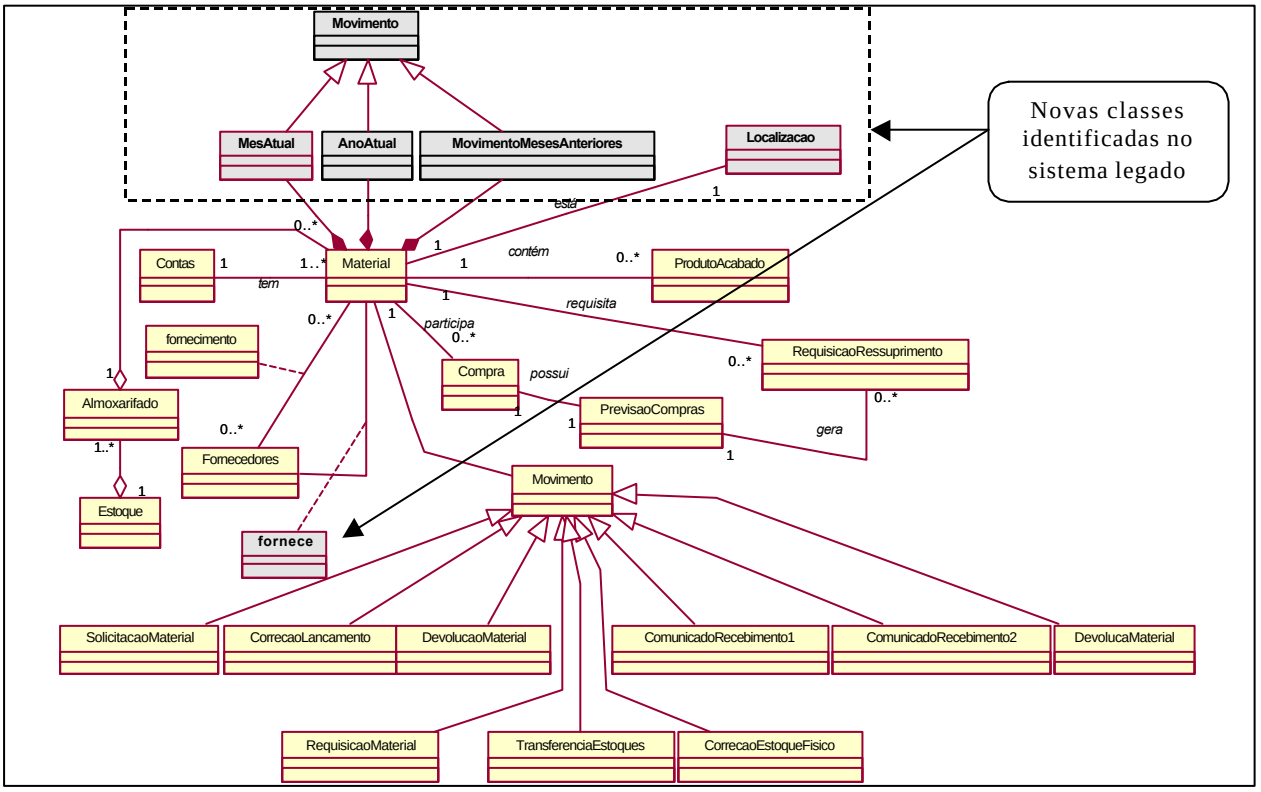


Figura 10 - MAS (Modelo de Análise do Sistema)

Relacionamentos que envolvem entidades fracas determinam que a existência dessa entidade depende da existência de uma entidade forte relacionada, isto é, o tempo de vida da entidade fraca depende do tempo de vida da entidade forte. Sendo assim, há uma similaridade semântica com relacionamentos de agregação da orientação a objetos, também conhecidos como Todo-Parte. As entidades que foram geradas por meio de papéis secundários geralmente são fracas e, sendo assim, devem fazer parte de um relacionamento de agregação no diagrama de classes do sistema.

O MAS apresentado na Figura 10, foi obtido em duas etapas. Na primeira, fez-se um mapeamento direto das classes do MASA. Posteriormente, observou-se que, como o sistema foi desenvolvido sem a utilização de técnicas de projeto adequadas, algumas classes e relacionamentos poderiam ser melhorados. Sendo assim, iniciou-se um processo de refinamento que a preocupação foi a de tornar o modelo de classes o mais orientado a objetos possível, sem redundâncias ou inconsistências. Esse processo de refinamentos considera a utilização evolutiva dos padrões: Definir Atributos, Definir Métodos e Analisar Hierarquias, descritos a seguir.

As classes representadas em cinza na Figura 10 foram identificadas por meio da aplicação deste padrão. Nota-se que houve pouca alteração nas classes já existentes no modelo.

3.10. Padrão: Definir Atributos

Deve-se analisar os trechos de código dos papéis principais dos arquivos de dados a fim de obter os atributos das classes existentes no MAS. A identificação dos atributos não apresenta muitas dificuldades, pois, como os papéis já foram identificados, o esforço concentra-se na identificação dos trechos de códigos geradores dos papéis. Tendo feito essa identificação, obter os itens elementares, que representam os atributos, é algo relativamente fácil.

3.11. Padrão: Definir Métodos

Representar os métodos relacionados na coluna Possíveis Métodos da Tabela Detalhe de Implementação (Tabela 6) nas respectivas classes especificadas na coluna PseudoClasses da mesma tabela. Assim, obtém-se os métodos das classes no Diagrama de Classes por meio da Descrição do Use Case correspondente.

3.12. Padrão: Analisar Hierarquias de Herança

Sistemas legados implementados em COBOL, freqüentemente, apresentam a utilização do comando `REDEFINES`. Esse comando, como já citado anteriormente, confere a um registro COBOL comportamentos distintos. Camargo e Penteado [5] identificaram três alternativas de utilização do comando `REDEFINES` e como podem se tratadas durante o mapeamento para um modelo de classes orientado a objetos. Na segunda alternativa, determinados campos do registro sempre são utilizados para qualquer uma das formas assumidas por ele, enquanto que os outros, que são redefinições, são utilizados apenas em algumas. Esse comportamento que o comando `REDEFINES` confere a um registro COBOL aproxima-se do conceito de generalização/especialização de orientação a objetos, em que vários tipos de objetos compartilham informações comuns. A Figura 3 apresenta um trecho de código que mostra essa segunda alternativa de `REDEFINES`. Esse tipo de utilização do comando `REDEFINES` deve ser tratado como sugere a Figura 11. O nível 01 do `FD` dá origem à classe Todo, tendo como atributos os campos comuns. Já os itens de grupo tornar-se-ão classes Parte. A multiplicidade deve ser 0..1 em relação às classes Parte, representando que elas podem ou não serem utilizadas.

Como o processo de engenharia reversa descrito aqui utiliza a segunda alternativa apresentada pela Figura 1, pouco trabalho foi gasto nesse cluster para solucionar problemas de redundâncias e inconsistências, pois esses já foram distribuídos ao longo do processo. Os problemas encontrados nesta fase decorrem de decisões de projeto mal formuladas que foram feitas pelo engenheiro de software responsável pela implementação do sistema legado.

O MAS obtido após a fase de refinamentos possui poucas diferenças em relação ao MASA obtido no padrão Criar Visão Orientada a Objetos dos Dados. Isso ocorre porque durante a aplicação do padrão Iniciar Análise dos Dados cuidou-se da identificação de papéis, que são representações do mundo real que o sistema manipula. Os refinamentos realizados restringem-se à eliminação de relacionamentos redundantes, identificação de outros papéis obscuros no código fonte, devido a muitas decisões de projeto, e alteração de nomes para mnemônicos mais significativos.

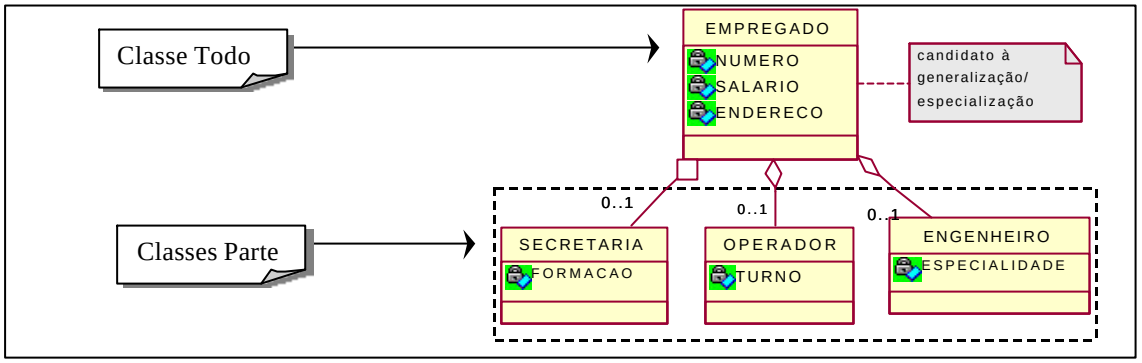


Figura 11 – Criação de Relacionamentos de Herança a partir de `REDEFINES`

Deve-se ressaltar que esse padrão foi utilizado de forma complementar durante a aplicação do padrão Iniciar Análise de Dados. Isso ocorre, pois, quando se utiliza a segunda alternativa de engenharia reversa, a preocupação com relacionamentos de herança já é tratada desde a aplicação dos primeiros padrões.

3.13. Padrão: Construir Diagramas de Seqüência

Deve-se elaborar os diagramas de seqüência a partir da descrição dos *Use Cases*. A Figura 12 apresenta o diagrama de seqüência para o Use Case AtualizarRequisicaoMaterial mostrado na Figura 8. A construção desse diagrama não apresenta complicações, já que se baseia na descrição dos *Use Cases* já identificados.

4. Considerações Finais

Este trabalho mostrou a aplicação da FaPRE/OO [14] para processos de engenharia reversa orientada a objetos de sistemas legados procedimentais de forma evolutiva e complementar. A aplicabilidade da FaPRE/OO usando a segunda alternativa para o processo de engenharia reversa orientada a objetos (ver Figura 1) pôde ser plenamente constatada. Nesse processo, papéis são identificados logo no seu início. Sendo assim, não se obtém um DER que represente as estruturas de dados do sistema legado, mas um que já se aproxima bastante do modelo de análise orientado a objetos final. Nessa alternativa, alguns padrões são utilizados de forma a interagir para a resolução de um problema e de modo evolutivo, destacando o poder da FaPRE/OO e distribuindo as consolidações ao longo de todo o processo.

A Tabela 7 mostra os padrões que foram utilizados de forma complementar para a realização do processo de engenharia reversa de sistemas COBOL apresentados neste trabalho.

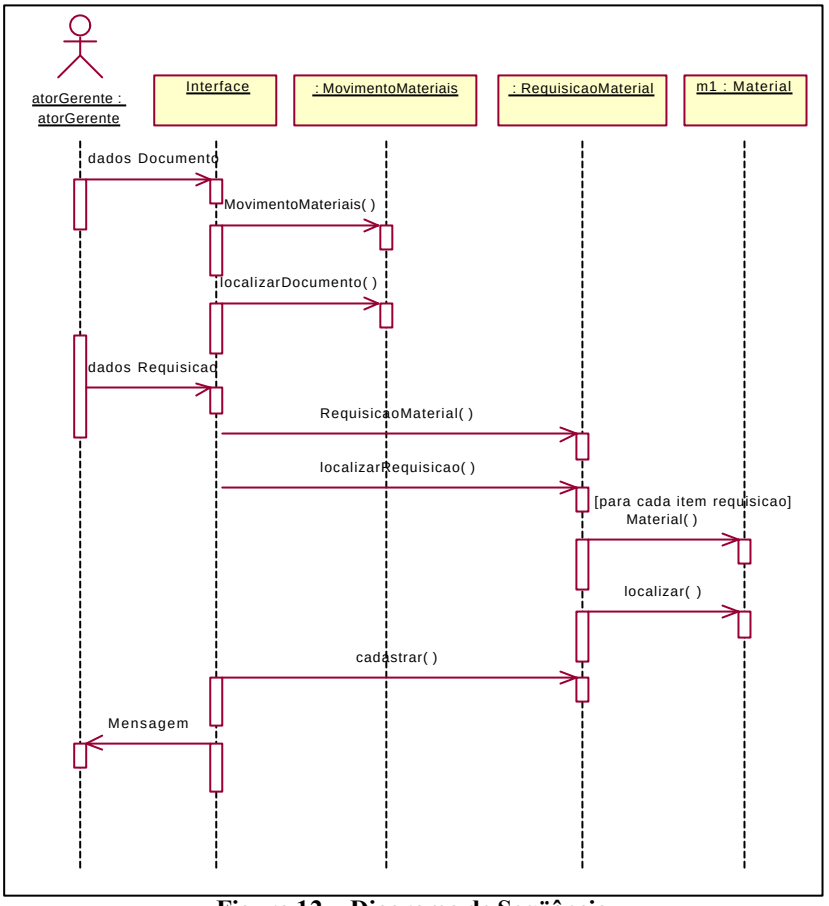


Figura 12 – Diagrama de Seqüência

Tabela 7– Padrões Complementares

Cluster	Padrões	Padrões complementares
Modelar os Dados do Legado	Iniciar Análise dos Dados	Tratar Anomalias Analisar Hierarquias
Modelar a Funcionalidade do Sistema	Construir Diagramas de *Use Cases* Elaborar a Descrição dos *Use Cases*	Iniciar Análise dos Dados
Modelar o Sistema Orientado a Objetos	Definir as Classes	Obter Visão Orientada a Objetos dos Dados

O padrão Iniciar Análise dos Dados utiliza, complementarmente, os padrões Tratar Anomalias e Analisar Hierarquias. Isso ocorre porque durante a identificação dos papéis é preciso analisar as hierarquias (itens de grupo) existentes em arquivos de dados COBOL. O padrão Tratar Anomalias é utilizado de forma complementar porque é durante a sua aplicação, que as pseudo-classes (papéis) são identificadas. Os padrões Construir Diagramas de *Use Cases* e Elaborar a Descrição dos *Use Cases* utilizam, complementarmente, o padrão Iniciar Análise dos Dados porque tanto a identificação, quanto a descrição dos *Use Cases*, são baseadas nos papéis que foram identificados nesse padrão. O padrão Definir Classes utiliza complementarmente o padrão Obter Visão Orientada a Objetos dos Dados, pois a definição das classes do modelo final deve considerar o modelo que mais se aproxima dele, que é o MASA.

As duas alternativas apresentadas na Figura 1 fornecem diretrizes bem definidas para que engenheiros de software possam realizar com segurança o processo de engenharia reversa. O engenheiro de software familiarizado com sistemas procedimentais tem, na FaPRE/OO [14], diretriz segura para que o processo de engenharia reversa seja passo a passo realizado e validado, primeira alternativa da Figura 1. Caso o engenheiro de software esteja familiarizado com

linguagens orientadas a objetos a alternativa 2 permite que, no início do processo de engenharia reversa, pela aplicação de forma complementar e evolutiva da FaPRE/OO, um modelo orientado a objetos do sistema legado procedimental seja obtido com facilidade.

Referências Bibliográficas

[1] Braga, R. T. V., **"Padrões de Software a partir da Engenharia Reversa de Sistemas Legados"**, São Carlos-SP, 1998. Dissertação de Mestrado. Universidade de São Paulo.

[2] Cagnin, M.I.; "Avaliação das vantagens quanto à facilidade de manutenção e expansão de sistemas legados sujeitos a engenharia reversa e a reengenharia", SãoCarlos – SP, 1999. Dissertação de Mestrado. Universidade Federal de São Carlos.

[3] Cagnin, M. I.; Penteado, R.; Masiero, P.C; "**Reengenharia com o uso de Padrões de Projeto"**. Florianópolis - Santa Catarina, 1999a. In: Simpósio Brasileiro de Engenharia de Software, SBES'99, 13, Anais.

[4] Camargo, V.V., "Reengenharia Orientada a Objetos de Sistemas COBOL com a utilização de Padrões de Projeto e Servlets", São Carlos-SP, 2001. Dissertação de Mestrado. Universidade Federal de São Carlos.

[5] Camargo, V.V.; Penteado, R.D. Diretrizes para a Realização da Engenharia Reversa de Sistemas COBOL utilizando Fusion/RE. In Proceedings do CLEI 2001 - México, (CD-ROM).

[6] Chikofsky, E. Reverse Engineering and Design Recovery - A Taxonomy. IEEE Software. v. 7, n. 1, p. 13-17. 1990.

[7] Demeyer, S.; Ducasse, S.; Nierstrasz, O., **"A Pattern Language for Reverse Engineering".** Proceedings of the 5th European Conference on Pattern Languages of Programming and Computing, (EuroPLOP'2000), Andreas Ruping(Ed.), 2000.

[8] Legacy Aid. www.casemaker.com. Consultado em 5/2002.

[9] Penteado, R. A. D., **"Um Método para Engenharia Reversa Orientada a Objetos",** São Carlos, 1996. 237 p. Tese (Doutorado em Física Computacional) - Instituto de Física de São Carlos, Universidade de São Paulo.

[10] Penteado, R., Germano, F., Masiero, P. C., **"An Overall Process Based on Fusion to Reverse Engineering Legacy code",** In: Working Conference Reverse Engineering, 3, 1996a, Monterey-California. Anais. IEEE, p. 179-188.

[11] Penteado, R., Braga, R.T.V., Masiero, P.C., **"Improving the Quality Legacy code by Reverse Engineering".** 4th International Conference on Information Systems Analysis and Synthesis, ISAS/98, págs 364-370, Julho/1998, Orlando-Flórida.

[12] Penteado, R.D.; Masiero, P.C.;Cagnin, M.I. – **"An Experiment of Legacy Code Segmentation to Improve Maintenability"**. In Proceedings of 3 rd European Conference on Software Maintenance and Reengineering. CSMR 99, Amsterdan, The Netherlands. IEEE, P91-1000, 1999.

[13] Recchia, E. L., **"Engenharia Reversa e Reengenharia Baseadas em Padrões"**, São Carlos-SP, 2002. Dissertação de Mestrado apresentada ao PPGCC - Universidade Federal de São Carlos.

[14] Recchia, E. L.; Penteado, R. – **FaPRE/OO: Uma Família de Padrões para Reengenharia Orientada a Objetos de Sistemas Legados Procedimentais** . Artigo a ser apresentado no The Second Latin American Conference on Pattern Languages of Programming. (Sugarloaf Plop, a ser realizado em Itaipava-RJ, agosto 2002)

[15] Unified Modeling Language. http//www.rational.com/uml/index.jtmpl. Consultado em 03/2002.

Special Session on Writing Patterns

WP wants to provide newcomers an opportunity to write their first pattern. There is no better way to learn what patterns are all about than by writing one yourself. After having done the tutorial about "Software patterns", in the first day of the Conference, the WP sessions aim at guiding newcomers to get started with their patterns.

The WP Session Dynamics

WP sessions were about 1 hour each. All participants were invited to read the papers in advance, so that they could contribute, during the session, with constructive criticism about the pattern. The patterns discussed in these sessions are beginning to emerge, so authors have heard about the format, the adequacy of notations, what to include in the several pattern elements, etc.

The session dynamics was similar to the Writers' workshop, but the author could interact with questions and clarifications during the session, rather than only in the last 10 minutes.

Papers of this session are candidates to be submitted to future PloP's, where they will be shepherded and workshoped.

Padrão Arquitetural para Sistemas Computacionais de Controle Supervisório

Centro Federal de Educação Tecnológica do Paraná
Programa de Pós-Graduação em Eng. Elétrica e Informática Industrial
Av. Sete de Setembro, 3165 - CEP 80.230-901 - Curitiba-PR – Brasil

Jean Marcelo Simão, Marcos Antonio Quináia, Paulo Cézar Stadzisz
{simao, quinaia, stadzisz}@cpgei.cefetpr.br

Resumo

Padrões de *software* representam uma área de pesquisa promissora em razão dos benefícios advindos da sua aplicação, principalmente em termos de produtividade alcançada com a reutilização. Em automática, padrões arquiteturais podem ser aplicados a problemas recorrentes envolvendo diversos tipos de sistemas computacionais. Uma aplicação complexa, para a qual padrões arquiteturais podem trazer grande contribuição, é o Controle Supervisório de Sistemas Automatizados de Manufatura (CS-SAM).

Este artigo propõe um padrão arquitetural para CS-SAM que atende a requisitos funcionais obtidos através da análise por diagramas de casos de uso. Esta análise considera diagramas específicos de casos de usos cujas recorrências são apresentadas em diagramas genéricos de casos de uso, empregando como notação uma extensão da UML. A concepção dos componentes do padrão arquitetural é realizada com um certo grau de generalização, uma vez que os requisitos funcionais genéricos estão estabelecidos. Estes componentes do modelo são, primeiramente, especializados de forma a atender requisitos específicos e, subseqüentemente, generalizados para comportar elementos recorrentes em maior grau de abstração.

Como características particulares, o padrão arquitetural apresenta um modelo de Monitoração/Comando, concebido em uma hierarquia de classes, e um modelo de Decisão/Coordenação que estabelece, em termos genéricos, uma lógica causal, na qual a avaliação e correlação de estados (observados na monitoração) implicam em uma seqüência de ordens (ativando comandos). A lógica causal no padrão arquitetural é expressa na forma de um Sistema Baseado em Regras genérico para CS-SAM. Cada sistema de Controle Supervisório instanciado a partir do padrão arquitetural proposto apresenta grupos de objetos entendidos como agentes, constituindo-se, portanto, em um Sistema Especialista realizado por agentes reativos e cooperativos.

Palavras-Chaves: Padrões Arquiteturais, Reuso de *Software*, Controle Supervisório, Sistemas Automatizados de Manufatura, Sistemas Baseados em Regra, Agentes.

Abstract

Software patterns represent a promising research area in reason of the subsequent benefits of its application, mainly in terms of productivity reached with the reuse. In automatic, architectural patterns can be applied in recurrent problems involving diverse types of computational systems. A complex application, for which architectural patterns can bring great contribution, is the Supervisory Control of Automated Manufacturing Systems (SC-AMS).

This article proposes an architectural pattern for SC-AMS that takes care of the functional requirements gotten through the analysis by use cases diagrams. This analysis considers specific use cases diagrams whose recurrences are presented in generic use cases diagrams, employing an UML extension as notation. The conception of the architectural pattern components is carried out with a certain generalization degree, since that the generic functional requirements are established. These components of the model are firstly specialized aiming to solve specific requirements and, subsequently, they are generalized to comprise recurrent elements in a wide abstraction degree.

As particular feature, the architectural pattern presents a model of Monitoring/Command, conceived in a class hierarchy, and a model of Decision/Coordination that establishes, in generic terms, a causal logic in which the evaluation and correlation of states (observed in the monitoring) implies in a sequence of orders (activating commands). The causal logic in the architectural pattern is expressed in the form of a generic Rule Based System for SC-AMS. Each system of Supervisory Control instantiated from the considered architectural pattern presents groups of objects understood as agents, therefore consisting in an Expert System carried out by reactive and cooperative agents.

Keywords: Architectural patterns, Software Reuse, Supervisory Control, Automated Manufacturing Systems, Rule Based Systems, Agents.

1 Problema

Conceber, Instanciar e Realizar Controle Supervisório de Sistemas Automatizados de Manufatura (CS-SAM) via padrão arquitetural de *software*.

2 Contexto

O enfoque deste trabalho é a apresentação de um padrão arquitetural para CS-SAM no qual os elementos constituintes são genéricos e aplicáveis a uma certa classe de fábricas.

Nesta seção é apresentada uma célula de manufatura que serve como objeto de compreensão do macro-contexto, bem como dos requisitos para a concepção do padrão em questão. A obtenção dos requisitos se dá pela compreensão e generalização do funcionamento desta célula e da estruturação e relacionamento entre os componentes que implementam o Controle Supervisório.

2.1 SAM / CS-SAM

Um SAM é um complexo sistêmico envolvendo sistemas eletromecânicos (i.e. equipamentos) e elementos computacionais (e.g. sistemas computacionais) integrados e cooperativos no sentido de produzirem manufaturas. O SAM faz parte de um contexto de automação maior intitulado automática. A automática consiste de uma sinergia de elementos de computação e automação voltada para a automatização de sistemas.

A Figura 1 apresenta um exemplo de SAM simulado na ferramenta ANALYTICE II (Koscianski et al., 1999) (Simão, 2001) que permite expressar as características fundamentais de sistemas industriais reais. Esta célula de manufatura é composta por diversos equipamentos e sua função é produzir peças fictícias dos tipos A e B.

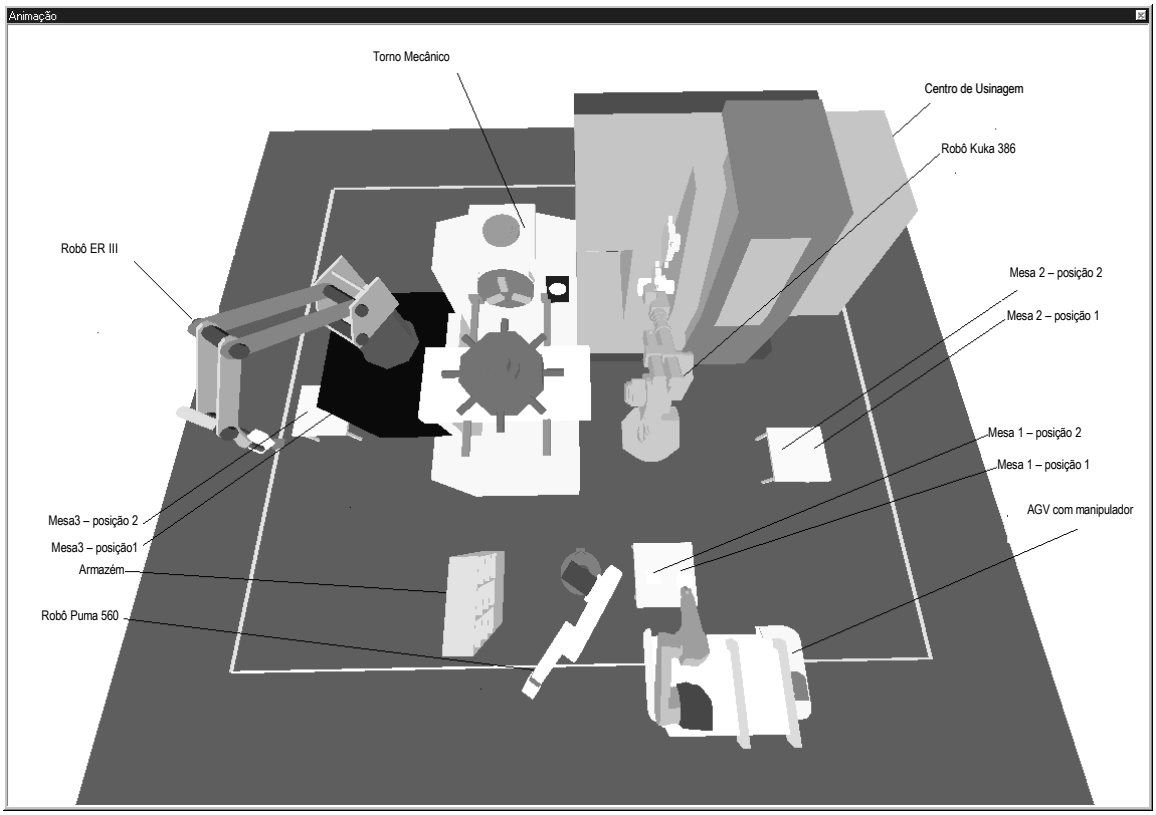


Figura 1 – Célula de Manufatura simulada em ANALYTICE II

Cada peça processada neste SAM possui um plano de processo gerado em um outro sistema de decisão chamado de Planejamento. O plano dita quais máquinas a peça deve visitar e quais operações devem ser realizadas sobre ela (Künzle, 1990) (Bongaerts, 1998). Omitindo as operações, oO plano de processo para peças A é {<armazém> <mesa 1> <centro-de-usinagem> <mesa 2>} e para peças B é {<mesa3> <torno> <mesa3>}. Poderiam existir ainda alternativas de fabricação no plano de processo, caso existisse um escalonador dinâmico para realizar as seleções em tempo de execução.

O papel do *software* de Controle Supervisório é fazer com que os elementos constituintes do SAM (e.g. tornos e robôs) trabalhem de forma harmônica para realizarem a fabricação das peças segundo os planos de processo (Mendes, 1995) (Miyagi, 1996). De uma forma geral, os elementos de um SAM podem ser classificados em equipamentos, elementos de hierarquia e elementos de processo.

Uma divisão comum para equipamentos é classificá-los como de atuação (realizam operações sobre as peças), de transporte (realizam o translado de peças) e de armazenagem (realizam o armazenamento de peças). No exemplo proposto, o Torno e o Centro de Usinagem são classificados como equipamentos de atuação, o Puma, o Kuka e o ERIII como de transporte e o Armazém e as Mesas como de Armazenagem.

Os elementos de hierarquia são subsistemas da planta industrial, como a estação de trabalho (i.e. grupo de equipamentos), a célula de manufatura (i.e. grupo de equipamentos e estações de trabalho) e a planta (i.e. grupo de equipamentos, estações e células). Como exemplo, o SAM ilustrado na figura 1, poderia conter três estações {<torno> <mesa3> <ERIII>}, {<centro-de-usinagem> <mesa 2> < Kuka>} e {<armazém> <mesa 1> <Puma>}.

O SAM como um todo poderia ser considerado como uma célula composta pelas três estações e pelo equipamento de transporte <AGV>.

Esta divisão hierárquica propicia o desenvolvimento do CS-SAM em diversos níveis, conhecido como Controle Supervisório Hierárquico (Künzle, 1990). Por exemplo, um CS Hierárquico pode determinar que algumas peças vão para uma célula e não para outra. Uma vez na célula, outro nível de coordenação deste CS-SAM determinará quais elementos daquela célula processarão as peças.

Quanto aos elementos de processo, eles englobam as peças, os lotes de peças e os paletes. Um lote de peças consiste em um grupo de peças de um mesmo tipo que avançam em conjunto no sistema de manufatura. Um lote tem uma prioridade de processamento e um plano de produção, permitindo escopos mais amplos de controle supervisório, como saber qual lote deve visitar qual célula. Por fim, um palete é um elemento sobre o qual uma ou mais peças (dependendo do modelo) são colocadas para fins de proteção e padronização no transporte. Os paletes são recursos limitados no SAM. Dependendo da morfologia das peças, certos SAMs podem não empregar paletes, como ocorre no exemplo estudado.

3 Forças

Em automática, padrões podem ser utilizados no desenvolvimento de *software* para Controle Supervisório de Sistemas Automatizados de Manufatura (CS-SAM).

Apesar dos numerosos estudos envolvendo CS-SAM (Chaar et al., 1993) (Mendes, 1995) (Miyagi, 1996) (Cury et al., 2001), nota-se uma carência de pesquisas específicas ao desenvolvimento de padrões arquiteturais para estes sistemas computacionais (Schmid, 1995). Considerando-se a complexidade e dimensão típica de sistemas CS-SAM, o desenvolvimento e emprego de padrões arquiteturais poderiam trazer uma contribuição importante para os desenvolvedores.

Um padrão de software é a materialização de uma solução genérica e reutilizável em diversos problemas recorrentes, baseado na observação de experiências de desenvolvimentos passados. Um padrão pode ser aplicado a uma classe de problemas análogos, diminuindo o esforço de concepção (Gamma et al., 1994).

Os padrões são alvo de várias discussões e estudos, caracterizando-se como uma nova área de pesquisa. Existem padrões em vários níveis de abstração e extensão, como por exemplo: padrões arquiteturais (Buschmann et al., 1996), de análise (Fowler, 1996), de projeto (Gamma et al., 1994), de programação (Coplien et Schmidt., 1995), de persistência e padrões para hipertexto/hipermídia (Vlissides et al., 1996) (Martin et al., 1997). Os padrões podem ser utilizados em diversos domínios de aplicação como em telecomunicações, sistemas de informação e automática.

Mais especificamente, um padrão arquitetural expressa uma organização ou esquema estrutural fundamental para sistemas de *software*. Este tipo de padrão prevê um conjunto predefinido de subsistemas, especificando suas responsabilidades e inclui regras e linhas gerais para a organização e relacionamento entre eles (Buschmann et al., 1996).

O processo de concepção de uma arquitetura computacional genérica robusta (e.g. padrão arquitetural) para CS-SAM não é, entretanto, uma tarefa simples pois além de se conceber uma estratégia de controle da fábrica, é necessário generalizá-la a um conjunto de

situações de controle de fábricas semelhantes. Algumas abordagens têm sido propostas na literatura (Schmid, 1995) (Bongaerts, 1998) (Langer et al., 2000) (Simão, 2001).

4 Solução

Este artigo tem como objetivo apresentar um padrão arquitetural aplicável na construção de *software* para CS-SAM. A solução começa com a análise funcional (específica e genérica) de uma fábrica fictícia, modelada na ferramenta de simulação ANALYTICE II (Koscianski et al., 1999) (Simão, 2001). A partir desta análise, sintetiza-se, na forma de um padrão, uma arquitetura genérica aplicável a outros sistemas de controle semelhantes.

Em síntese, o Padrão Arquitetural proposto para CS-SAM tem como essência uma arquitetura de *software* genérica no formato de um Sistema Genérico Baseado em Regras, modelado sob o paradigma da Orientação a Objetos, onde as instâncias constituem-se em Sistemas Especialistas realizados por Agentes Computacionais reativos e cooperativos que implementam uma técnica eficiente de inferência.

4.1 Notação Utilizada

Para a proposição de um padrão arquitetural para CS-SAM utiliza-se uma análise de requisitos funcionais seguida de uma análise estrutural. A análise de requisitos funcionais primeiramente define as responsabilidades de um Controle Supervisório para o sistema de manufatura proposto na Figura 1 e, então, as generaliza para uma classe maior de CS-SAM. Na análise estrutural definem-se as classes para este sistema realizando-se, a seguir, a generalização do modelo.

Neste trabalho, diagramas específicos permitem definir responsabilidades e localizar recorrências que são então expressas em diagramas genéricos, segundo um formalismo estabelecido. Os diagramas genéricos definem o padrão arquitetural e a notação utilizada para compô-los pode ser considerada uma extensão da UML.

A análise de requisitos funcionais específicos baseia-se em diagramas de casos de usos comuns e a análise de requisitos funcionais genéricos baseia-se em uma extensão de diagramas de casos de uso. Esta extensão é denominada Diagrama de Caso de Uso Genérico que contém casos e subcasos de usos genéricos e atores genéricos, além das primitivas da UML.

Atores genéricos são representados acrescentando-se um círculo cinza ao fundo do ícone típico da UML, o que equivale à definição de um estereótipo <<genérico>> (Figura 2). Um ator genérico representa um ator que pode ter múltiplas ocorrências em um diagrama de casos de uso derivado. Considerando-se, como exemplo, o ator genérico Equipamento de um sistema CS-SAM, quando a arquitetura genérica fosse derivada para um sistema em particular, este ator poderia ser mapeado em mais de um ator representando os diferentes equipamentos da fábrica. Pode-se entender que um ator genérico possui uma cardinalidade indicando o número de ocorrências que podem existir deste ator em um mesmo diagrama de casos de uso. Na notação proposta, a cardinalidade é indicada no interior da elipse do caso de uso usando a notação padrão da UML.

Os casos de uso genéricos têm como representação gráfica uma elipse com uma faixa cinza à sua esquerda (Figura 2), o que equivale à definição de um estereótipo <<genérico>>. Um caso de uso genérico representa um caso de uso que pode ter múltiplas ocorrências em um diagrama de casos de uso derivado do modelo genérico. Assim como para atores, um caso de uso genérico possui uma cardinalidade indicando o número de ocorrências. Como exemplo, pode-se considerar o caso de uso genérico Monitorar Equipamento que poderia ter várias ocorrências como: Monitorar Robô Puma, Monitorar Centro de Usinagem, etc.

As análises estruturais específica e genérica empregam diagramas com classes instanciáveis e abstratas conforme a notação original da UML.

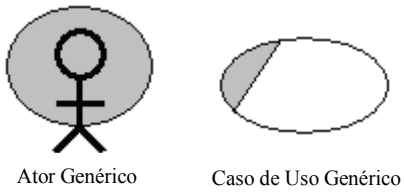


Figura 2 – Notação utilizada

4.2 Requisitos Funcionais do Padrão Arquitetural

Para que o Controle Supervisório consiga controlar a fabricação de peças é necessário monitorar os estados de elementos da fábrica (equipamentos, peças, paletes, hierarquias), decidir como coordenar estes elementos segundo os estados monitorados e, então, comandá-los segundo uma coordenação apropriada.

A análise de requisitos funcionais para o CS-SAM pode ser dividida nos aspectos relacionados ao Comando, à Monitoração e à Decisão/Coordenação dos elementos de um Sistema Automatizado de Manufatura.

A atividade de Comando consiste em enviar comandos a equipamentos ou a elementos de hierarquia (e.g. “robô transporte peça” ou “célula processe o lote de peças”). A atividade de Monitoração consiste em acompanhar os estados discretos dos elementos do SAM (e.g. “robô parado”, “estação de trabalho ocupada”, “peça na etapa três do plano de processo”). A Decisão/Coordenação é responsável pela análise dos estados dos elementos do SAM, de forma a decidir quando coordená-los (i.e. instigar comandos em uma determinada seqüência) para que realizem o processo produtivo segundo planejamento prévio e protocolos específicos (Simão, 2001).

Durante a análise de requisitos funcionais, são criados os diagramas de casos de uso específicos que são, então, generalizados obtendo-se os diagramas de casos de uso genéricos. Na análise de requisitos são considerados apenas os aspectos de *software*, uma vez que o CS-SAM é essencialmente um sistema computacional.

4.2.1 Requisitos Funcionais do Comando

Um serviço importante em CS-SAM é o envio de comandos aos equipamentos. Por exemplo, o Controle Supervisório deve ser capaz de comandar o robô Puma para realizar a

tarefa de pegar uma determinada peça do Armazém e pô-la na Mesa-1. Este serviço é, então, mapeado no subcaso de uso *Comandar Puma* no diagrama da Figura 3.

A Figura 3 ainda traz os casos de uso Comandar AGV, Comandar Centro de Usinagem e Comandar Armazém para melhor exemplificar a atividade de comando dos equipamentos.

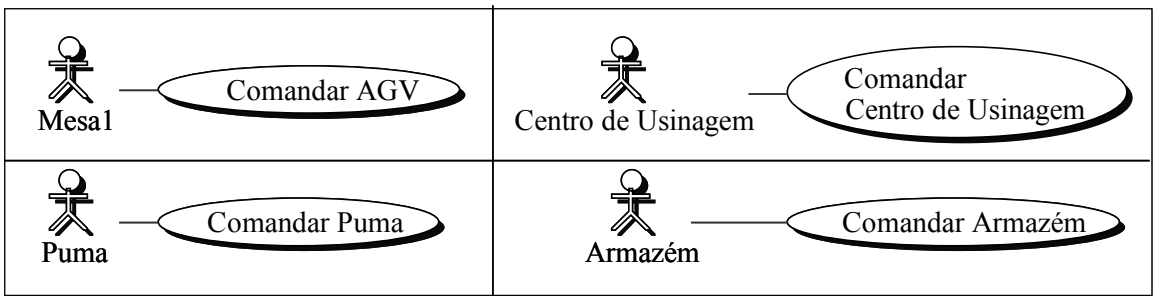


Figura 3 - Comando de Equipamentos

A partir da observação dos diagramas específicos (Figura 3), obtém-se um diagrama de caso de uso genérico para qualquer equipamento (Figura 4). Neste diagrama observa-se o requisito funcional genérico necessário à concepção de um padrão arquitetural no tocante ao Comando de equipamentos no CS-SAM.

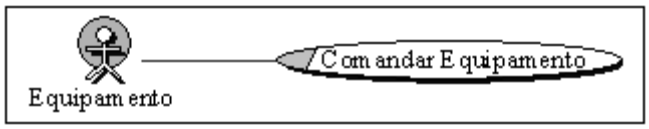


Figura 4 – Comando de Equipamento

Uma técnica possível de generalização é analisar se uma solução aplicada a um grupo de elementos do sistema aplica-se também a um conjunto maior de elementos. Por exemplo, quando se comanda um equipamento, está se executando esta operação sobre um subsistema fabril, portanto indaga-se a "genericidade" dos requisitos funcionais de Comando para outros subsistemas ou elementos da fábrica.

Analisando os requisitos funcionais para comandar os elementos de hierarquia, nota-se a similaridade para com os equipamentos. O diagrama da Figura 4 pode ser generalizado para suportar estes elementos (Figura 5).

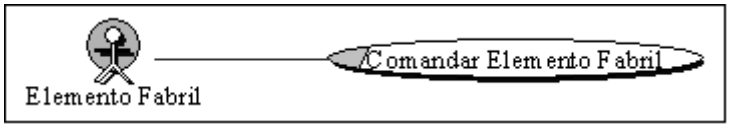


Figura 5 - Comando de Elemento Fabril

4.2.2 Requisitos Funcionais da Monitoração

É função do CS-SAM estabelecer os momentos apropriados para o envio de comandos. Por exemplo, um comando não pode ser dado ao robô Puma enquanto ele se

encontrar em um estado de quebra ou em um estado de execução de outra atividade. Portanto, um serviço que precede o Comando é a observação dos estados discretos dos equipamentos. Os casos de uso *Monitorar Puma, Monitorar Centro de Usinagem*, *Monitorar AGV* e *Monitorar Armazém* exemplificam esta funcionalidade (Figura 6).

Processos de *Monitoração* incluem tanto o registro quanto a disponibilização dos estados de cada atributo do objeto observado (e.g. monitorar cada posição do Armazém que pode estar *ocupada* ou *desocupada*), sendo que o estado geral é dado pela composição dos estados específicos. Desta forma, *Monitorar* inclui, genericamente, dois subcasos de usos chamados *Registrar Estados* e *Disponibilizar Estados* (Simão, 2001).

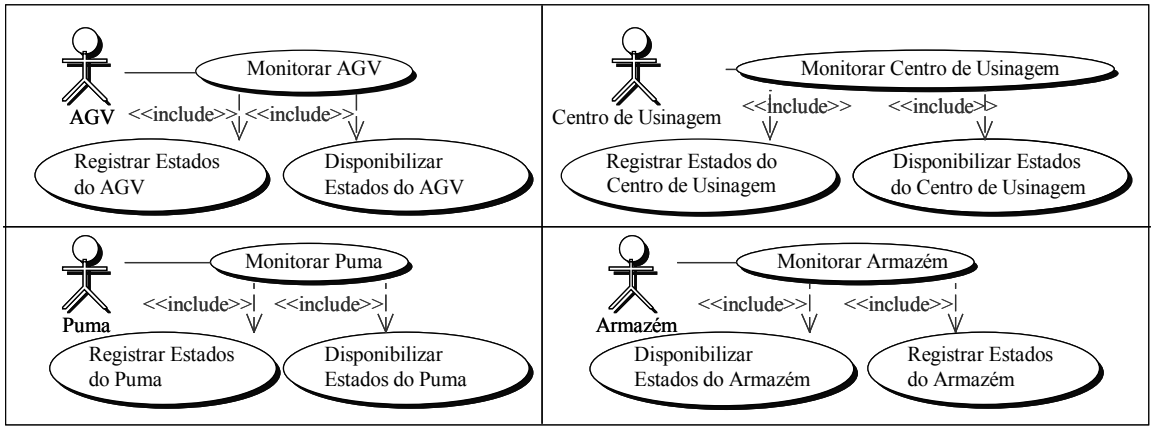


Figura 6 - Monitoração/Comando de Equipamentos

A partir da observação dos diagramas específicos de Monitoração (Figura 6), obtém-se um diagrama de casos de uso genérico para qualquer equipamento (Figura 7). Neste diagrama observa-se os requisitos funcionais genéricos necessários à concepção de um padrão arquitetural no tocante a Monitoração de equipamentos no CS-SAM.

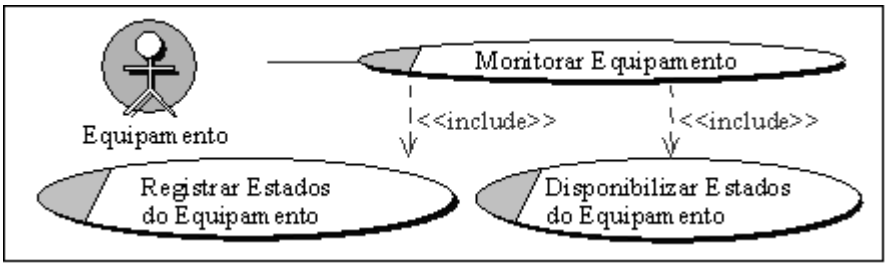


Figura 7 - Monitoração de Equipamento Genérico

Analisando os requisitos funcionais para monitorar os elementos de hierarquia, nota-se a similaridade com os equipamentos. O diagrama da Figura 7 pode ser generalizado para suportar estes elementos.

Em um SAM existem ainda outros elementos fabris que devem ser considerados na Monitoração. Estes elementos são as peças, o lote de peças e os paletes, chamados de elementos de processo. No que diz respeito às peças e lotes de peças, é necessário acompanhar seus estados em relação a seus planos de processo e, também, a suas prioridades

de produção. Com relação aos paletes, seus estados são determinados pela existência de peças nas suas posições e, também, pela sua posição na planta.

Propõe-se, então, uma generalização ainda maior do caso de uso para Monitoração de forma a comportar tanto elementos de processo quanto elementos de hierarquia (Figura 8).

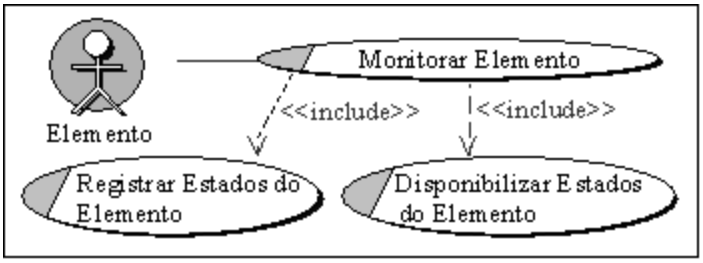


Figura 8 - Monitoração de Elemento Genérico

4.2.3 Requisitos Funcionais da Decisão/Coordenação

No que diz respeito à atividade de Decisão/Coordenação, os requisitos funcionais estão relacionados com a tomada de decisão baseada nos estados dos elementos monitorados e com a coordenação destes elementos segundo uma lógica de comandos.

Estas funcionalidades da Decisão/Coordenação podem ser observadas nas interações entre os equipamentos para a realização do beneficiamento das peças. Um exemplo é o transporte realizado pelo robô Puma, entre o Armazém e a Mesa-1. Para que se possa realizar esta atividade, é necessário haver uma peça no Armazém, que a Mesa-1 esteja com uma posição livre para receber a peça e que o Puma esteja disponível. Além disso, é necessário saber se o plano de processo da peça considerada prevê, neste momento, que ela seja transportada para a Mesa-1. Em suma, antes de decidir se o Puma realizará o transporte, é necessário avaliar os estados dos elementos envolvidos conforme demonstra o caso de uso da Figura 9.

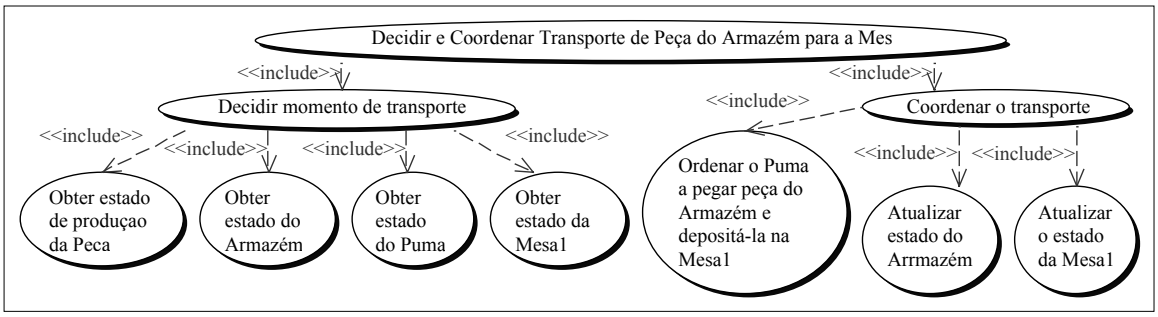


Figura 9 – Decisão/Coordenação do transporte de Peça do Armazém para a Mesa1

De mesma forma, se for avaliado o caso do transporte de peças da Mesa-1 para a Mesa-3, via AGV, encontram-se similaridades com o caso anterior. Para que este transporte ocorra, também são necessárias algumas condições sobre os elementos envolvidos (i.e. "ter uma peça na Mesa-1", "o AGV estar livre", "a Mesa-3 dispor de uma posição vaga para receber a peça" e "o plano de processo da peça prever a Mesa-3, neste momento, como a próxima visitação"). Ainda analisando o transporte das peças de Mesa-1 para o Centro de

Usinagem ou o transporte da Mesa-3 para o Torno, verificam-se similaridades funcionais. Também, em todos os casos de uso, nota-se que, além de realizar estas avaliações, é necessário tomar decisões a respeito dos estados discretos observados e, se pertinente, enviar ordens para que os equipamentos realizem suas tarefas.

Do estudo do diagrama de casos de uso específico, obtém-se o caso de uso genérico, retratado na Figura 10. Este diagrama comporta a interação entre os diversos elementos do SAM (i.e. de processos, de hierarquia e equipamentos).

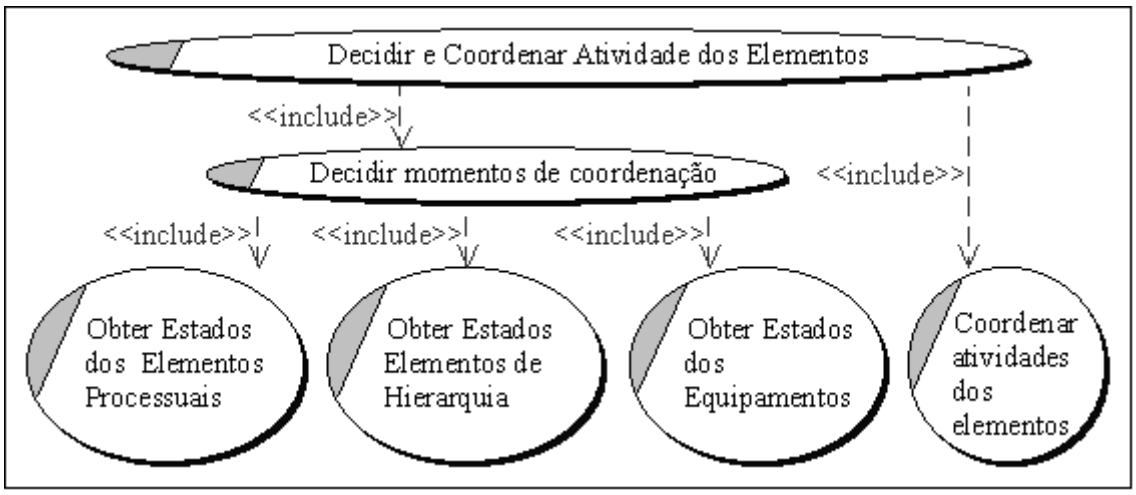


Figura 10 - Decisão e Coordenação das atividades dos elementos fabris

4.3 Padrão Arquitetural para CS-SAM

Tendo como base a análise específica e genérica de requisitos funcionais, esta seção apresenta diagramas de classes específicos e genéricos, utilizados na descrição do padrão arquitetural para CS-SAM. Os diagramas são apresentados em duas etapas. A primeira etapa envolve a atividade de Monitoração e a atividade de Comando (quando esta for pertinente), uma vez que estas atividades são realizadas sobre os elementos fabris. A segunda etapa visa as atividades de Decisão/Coordenação.

Ressalta-se que, em termos de engenharia de *software*, cada uma destas atividades pode ser classificada como um padrão de projeto. Um padrão de projeto consiste de um ou vários elementos de projeto de *software*, tais como módulos, interfaces, classes, objetos, métodos, relações entre elementos, e uma descrição do seu comportamento. (Buschmann et al., 1996) (Gamma et al., 1994).

Os diagramas de classes para Monitoração/Comando são, inicialmente, concebidos com um nível de genericidade intermediário (e.g. como diagramas genéricos para qualquer equipamento, para qualquer elemento de hierarquia ou para qualquer elemento de processo). A seguir, são incluídas especializações refinando as classes até o nível de classes instanciáveis, projetadas para tratar os elementos do SAM. Esta etapa constitui-se na parte descendente (*top-down*) do processo de concepção do modelo de Monitoração/Comando. A próxima etapa da concepção da Monitoração/Comando se dá pela definição de uma classe base para as classes mais abstratas existentes, reunindo-as em uma raiz de herança única. Esta

segunda etapa caracteriza-se como a parte ascendente (*bottom-up*) do processo de concepção da Monitoração/Comando.

Os diagramas de classes para Decisão/Coordenação são concebidos de forma que as instâncias possam avaliar e correlacionar quaisquer estados observados na Monitoração, bem como coordenar quaisquer comandos predeterminados no Comando. Para reduzir a complexidade no tratamento dos diferentes estados e no envio dos diferentes comandos aos elementos fabris, define-se um conjunto de classes padronizadas para a Decisão/Coordenação.

O padrão arquitetural apresenta colaborações de classes fortemente acopladas cujas especializações permitem, portanto, criar instâncias em grupos de objetos coesos. Cada um destes grupos é considerado um agente computacional. Esta abstração na forma de agente facilita a compreensão das interações entre os grupos de objetos.

Os conceitos relativos a agentes, encontrados na literatura, são variados. Neste trabalho considera-se um agente como um módulo de *software* com alto grau de coesão, com escopo bem definido, com autonomia e pertencendo a um certo contexto, no qual seu comportamento atual pode e, provavelmente, influenciará na sua existência futura (Franklin et Graesser, 1996) (Müller, 1998) (Rich et Knight, 1991) (Russell et Norvig, 1995).

O padrão arquitetural proposto é orientado a objetos e o conceito de agente é aplicado com reservas. De fato, pretende-se identificar entidades mais abstratas nas instâncias das especializações do padrão, através do conceito de agentes cooperativos e reativos. Estas instâncias não se classificam necessariamente como um sistema multi-agente, embora pudessem ser consideradas como tal (Franklin et Graesser, 1996) (Müller, 1998) (Rich et Knight, 1991) (Yufeng et Shuzhen, 1999).

Uma vez estabelecida a parte estrutural do padrão arquitetural e apresentadas classes que permitam instanciar agentes para o SAM, um diagrama de atividades demonstra, genericamente, a dinâmica de interações entre os elementos da Monitoração/Comando com a Decisão/Coordenação.

4.3.1 Diagramas de Monitoração/Comando

Para atender aos requisitos funcionais da Monitoração/Comando, propõe-se um agente computacional associado a cada elemento do SAM. Em termos de Monitoração, o agente é responsável por padronizar o registro e a disponibilização dos estados do elemento representado. Quanto ao Comando, cabe ao agente comunicar-se com o elemento fabril para estimular a realização de alguma atividade, gerando uma provável mudança de estado. Desta forma, cada agente de Monitoração/Comando inclui o protocolo específico de comunicação com o elemento fabril associado.

Monitoração/Comando de Equipamentos

Cada equipamento do SAM (e.g. robô, torno ou esteira) possuirá um agente para o representar, monitorar e comandar. Este agente é designado como um ***cae*** (comando ativo de equipamento) e a classe principal de seu modelo é a ***CAE***. Um ***cae*** ainda tem como objetos associados um conjunto de ***at***s e um conjunto de ***mt***s, que provêm respectivamente das classes ***Atributo*** e ***Metodo***.

Um ***at*** representa e mantém o estado discreto de uma característica de um ***cae***. Como exemplo, um ***at*** para acompanhar os estados de uma posição de um armazém (i.e. com peça

ou sem peça) ou um ***at*** para registrar o estado do atuador de um robô (i.e. fechado ou aberto). O acompanhamento de estado é feito pelo ***cae*** através do interfaceamento com os dispositivos (via rede local ou porta de comunicação) ou por inferência através de algum artifício dedutivo como temporizadores. Cada ***at*** ainda tem como função notificar a mudança de estados a elementos de monitoração e decisão apropriados.

Um ***mt*** é responsável por alterar o valor de um ***at***. Um ***cae*** ainda pode se associar a um ou mais ***cms*** (i.e. agentes cuja classe principal é denominada ***Comando***, derivada da classe ***Metodo***). Um ***cm*** tem como responsabilidade adicional comandar, via o ***cae*** associado, o equipamento representado. Cada ***cm*** ainda apresenta a capacidade de inferir informações (e.g. qual ferramenta usar numa operação) e, normalmente, recebe parâmetros para realizar suas atribuições.

Para melhor representar os equipamentos, pode-se ter hierarquias de classes derivadas de ***CAE***, de ***Atributo*** e de ***Metodo***. Um bom nível de especialização a partir de ***CAE*** é definir conjuntos de classes para equipamentos de armazenagem, transporte e atuação, nos quais se especifica elementos mais refinados mas ainda genéricos e aplicáveis a cada escopo.

A Figura 11 apresenta um conjunto de classes para representar individualmente equipamentos de armazenagem. A classe ***CAEArmazenagem*** possui uma relação de agregação com a classe ***ATPosicao*** concebida para tratar posições de armazenamento. Para cada objeto do tipo ***ATPosicao*** existe um objeto do tipo ***MTAlEstPos*** para realizar sua mudança de estado. A classe ***CAEArmazenagem*** é especializada em ***CAEArmazem*** e ***CAEMesa*** para instanciar, respectivamente, agentes responsáveis pela monitoração e comando do Armazém e das Mesas da célula de manufatura exemplo.

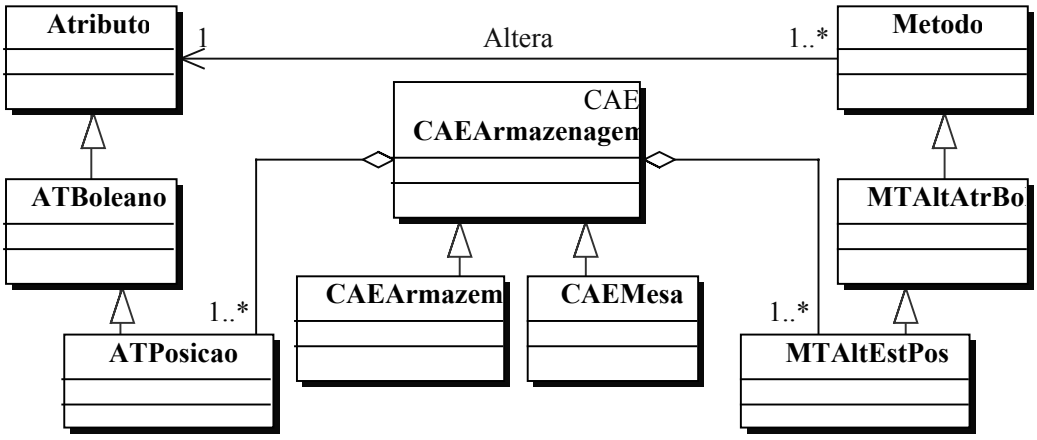


Figura 11 – CAE especializado para equipamentos de armazenagem

A Figura 12 apresenta uma hierarquia especializada para representar equipamentos de transporte. A classe ***CAETransporte*** é associada à classe ***ATLivre*** (criada para especificar a disponibilidade dos equipamentos representados pelos agentes instanciados). ***CAETransporte*** é especializada em ***CAETranpFixo*** e ***CAETranspMóvel***. Para ***CAETranspFixo*** há um ***CMMoverAtuador*** que trata a movimentação do atuador e um ***CMAbrirFecharAtuador*** para tratar sua abertura e fechamento. ***CAETranpFixo*** é ainda especializada em ***CAEPUMA560***, ***CAEKuka*** e ***CAEBraçoMec*** que servem para instanciar os objetos responsáveis por equipamentos da célula exemplo. Para ***CAETranspMóvel*** existe o ***CMMover*** para tratar movimentações na fábrica. Esta subclasse ***CAETranspMovel*** é especializada

CAETranspMvAtivo e ***CAETranspMvPassivo*** (i.e. com ou sem atuador). ***CAETranspMvAtivo*** permite derivar uma classe para representar o AGV.

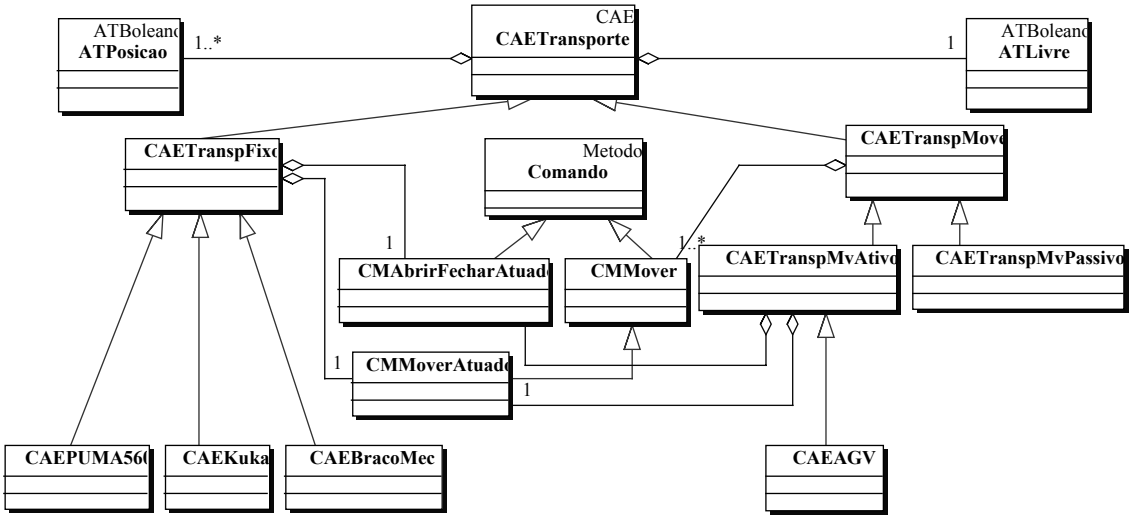


Figura 12 - CAE especializado para equipamentos de transporte

A Figura 13 traz uma classe especializada para equipamentos de processamento de peças, chamada ***CAEExecução***. Esta classe relaciona-se com a classe ***ATLivre***, bem como com a classe ***CMTrabalharPeça*** voltada ao comando do processamento da peça.

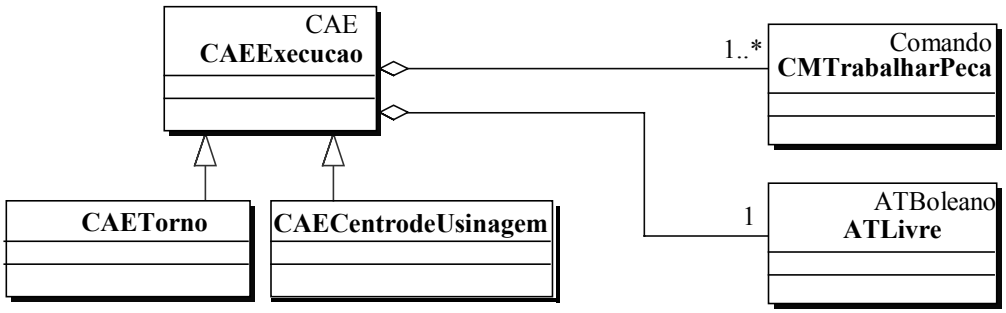


Figura 13 - CAE especializado para equipamentos de processamento de peças

Para que estes diagramas de classe representem equipamentos reais, é necessária uma maior especificação de detalhes de funcionamento interno das classes, bem como interfaceamento com os equipamentos fabris. De fato, para projetar um ***CAE*** para equipamento real, é necessário conhecimento multidisciplinar. Estes diagramas de classe têm como principal objetivo expressar as interfaces padronizadas para com a decisão e não detalhes de implementação.

Os elementos comuns às classes ***CAETransporte***, ***CAEArmazenagem*** e ***CAEExecução*** são agrupados ou associados à classe ***CAE***. Um exemplo é a associação com a classe ***ATListaPecaPlanoProcesso.*** A função de cada instância desta classe é indicar o próximo destino de cada peça segundo seu plano de processo.

A Figura 14, além de sintetizar os diagramas das figuras 11, 12 e 13, define a parte de Monitoração/Comando de equipamentos para o padrão arquitetural de CS-SAM. Em todos estes diagramas de classe, as únicas classes passíveis de instanciação são as que representam

equipamentos específicos, assim como as derivações de ***Atributo*** e ***Método*** relacionadas. Portanto, todas as demais classes são consideradas como classes abstratas.

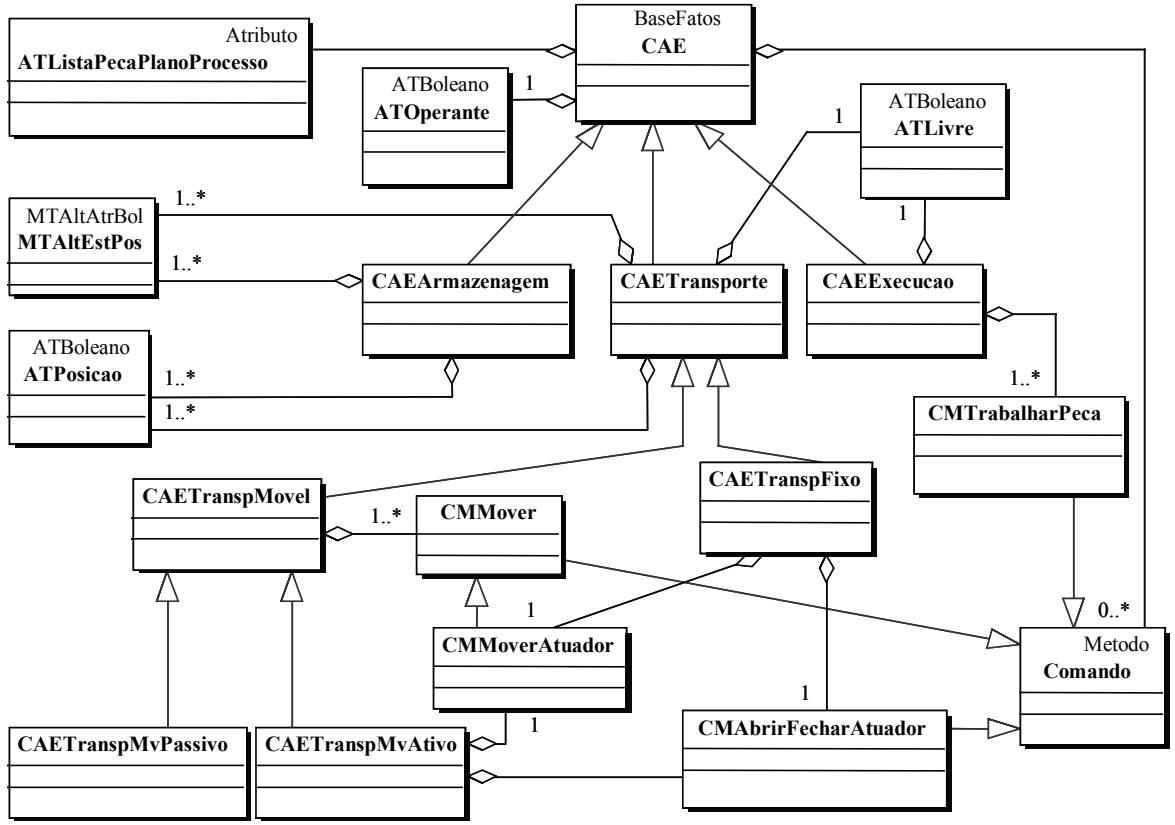


Figura 14 - Hierarquia de Equipamentos

Monitoração de Elementos de Processo (Peças, Lotes e Paletes)

Para realizar o CS-SAM é necessário monitorar os estados das peças, lotes e paletes que transitam no SAM. Com este intuito, definem-se classes de monitoração chamadas ***ElPrPalete***, ***ElPrPeça*** e ***ElPrLote***. Estas classes (Figura 15) são especializações da classe ***ElPr*** (Elemento de Processo) e cada qual também pode possui derivações de ***Atributo***, para registrar estados, e de ***Método***, para alterá-los.

Para a classe ***ElPrPeca*** existe uma classe ***ATPlanoProcesso*** para registrar o estado da peça representada com relação ao seu Plano de Processo. A classe ***ElPrPalete*** relaciona-se com a classe ***ATPosicao***. Ainda, para a classe ***ElPrPalete***, são definidas duas classes chamadas ***MTReceberLiberarPeça*** e ***MTFixarDesfixarPeça***, relacionadas a ***ATPosicao***. ***ElrPrLote*** representa um lote de peças e os estados de suas instâncias influenciam-se pelas instâncias de ***ElPrPeça*** relacionadas. Por fim, a classe base ***ElPr*** possui um atributo especificando sua prioridade de processamento.

As peças não fazem parte do SAM, elas são processadas por ele. Por isto, os elementos de processo não são focados diretamente pela Decisão/Coordenação, mas sim de forma indireta, uma vez que servem para compor o estado dos demais elementos do SAM passíveis de monitoração e comando. Por exemplo, um objeto do tipo ***ATListaPecaPlanoProcesso*** tem seu valor em função do estados de objetos do tipo ***ATPlanoProcesso***.

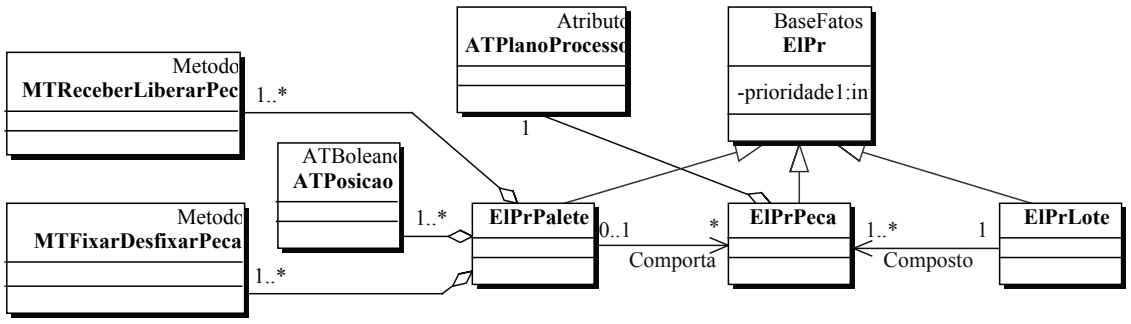


Figura 15 - Diagrama de Classes para Elementos Processuais

Monitoração/Comando de Níveis Hierárquicos

Para representar, decidir, monitorar e comandar níveis hierárquicos existe uma classe denominada ***ElHi*** (Elemento de Hierarquia). Esta classe especifica a capacidade de suas instâncias agregarem objetos dos tipos ***ElHi***, ***ElPr*** e ***CAE*** (Figura 16), assim como agregarem elementos de decisão.

Cada instância ***ElHi*** mantém ***at***s cujos estados advém de monitoração própria (e.g. realizada via interfaceamento com sensores) e em função dos valores dos ***at***s de seus agregados.

Uma instância ***ElHi***, similarmente aos ***cae***s, pode associar-se a ***cm***s especializados, chamados de ***cmcp*** (comandos compostos). Um ***cmcp*** tem como função induzir o início da atividade de controle supervisório no escopo do elemento de hierarquia representado. Por exemplo, o comando "produzir lote de peças na célula" ativa os agentes responsáveis pelo controle supervisório no escopo da célula comandada.

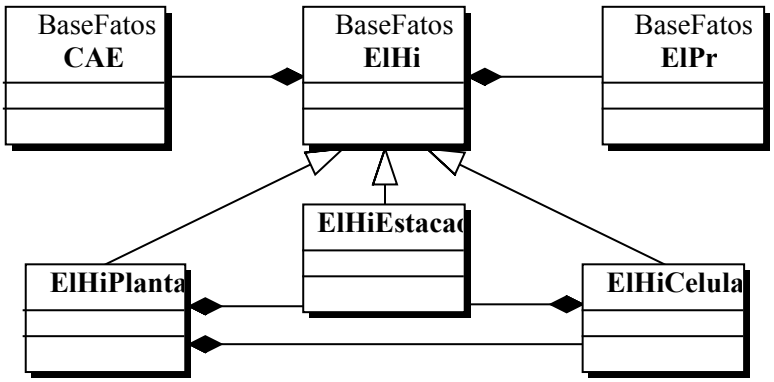


Figura 16 - Diagrama de Classes para ElHi

Monitoração/Comando no Padrão Arquitetural

Todos os agentes de Monitoração/Comando analisam estados de elementos do SAM e têm a capacidade de alterá-los. Cada estado que ocorre no sistema pode ser considerado como

um fato. Portanto, cada um dos agentes trata a mudança de fatos ocorridos sobre o elemento que ele representa.

Genericamente, ***CAE***, ***ElHi*** ou ***ElPr*** é considerado do tipo ***BaseFatos***. As instâncias (agentes) de qualquer derivado da classe abstrata BaseFatos são designadas como ***abf*** (agente da base de fatos) (Figura 17). Na ***BaseFatos*** são encapsuladas características comuns às suas especializações (e.g. identificador).

Conforme observado, toda derivação da ***BaseFatos*** compõe-se com derivados de **Atributo** e **Método**. Em última análise, a classe **Atributo** é uma interface padrão de saída e **Método** é uma interface padrão para entrada de informações nas derivações da ***BaseFatos***. Assim, a classe ***BaseFatos*** é uma forma de expressão comum (ou um padrão) para monitorar e comandar os elementos que possam influenciar a decisão.

A essência da Monitoração/Comando para este padrão arquitetural encontra-se nas figuras 14, 15, 16 e 17. Para conceber a Monitoração/Comando de CS-SAM especializam-se estes diagramas para tratar o caso específico e fazem-se as instanciações necessárias que, então, realizarão a atividade de Controle Supervisório.

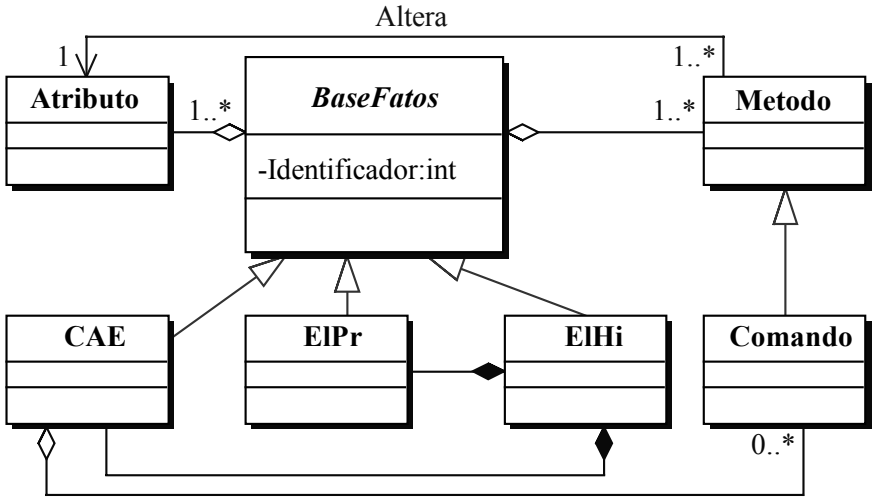


Figura 17 - Diagrama de Classes - Monitoração e Comando

4.3.2 Decisão/Coordenação

Para compor a parte do padrão arquitetural relativo à Decisão/Coordenação é necessário um conjunto de classes que modele os relacionamentos com a Monitoração/Comando, bem como a lógica causal existente entre a Decisão e a Coordenação. O modelo deve prever que a Decisão ocorre pela avaliação e correlação dos fatos observados nos ***at***s dos ***abf***s, enquanto a Coordenação envolve um conjunto de ordens sobre ***mt***s e ***cm***s dos ***abf***s, em resposta à decisão tomada, alterando estados de ***at***s.

Por conseqüência, observa-se que o CS-SAM trabalha sob a forma de um Sistema de Produção (SP) (Simão, 2001). Um modelo utilizado freqüentemente para realizar SPs é o de Sistema Baseado em Regras (SBR) (Pan et al., 1998) (Rich et Knight, 1991).

A Decisão/Coordenação, no padrão arquitetural proposto, toma a forma de um diagrama de classes que modela um SBR genérico. Portanto, cada instância criada, a partir da

especialização do padrão arquitetural, constitui-se de um Sistema Especialista, realizado por agentes, aplicado sobre um determinado sistema de produção industrial.

Instância de Decisão/Coordenação

Para que cada CS-SAM constituído, segundo o padrão arquitetural proposto, possa realizar uma lógica de Decisão/Coordenação, ela pode ser expressa em regras tais como as descritas nas Figuras 18 e 19.

A Figura 18 apresenta uma Regra expressando o conhecimento necessário para realizar o transporte de peças do Armazém para a Mesa-1. A regra dita o seguinte: se existe uma ou mais peça no Armazém cujos plano de processo indicam a Mesa-1 como próximo equipamento a ser visitado, se o Puma está livre e se a Mesa-1 tem alguma posição livre, então o Puma deve transportar uma das peças do Armazém para a Mesa-1, segundo parâmetros específicos. Quando mais de uma peça é passível de transporte, a escolha é feita pelo escalonador dinâmico.

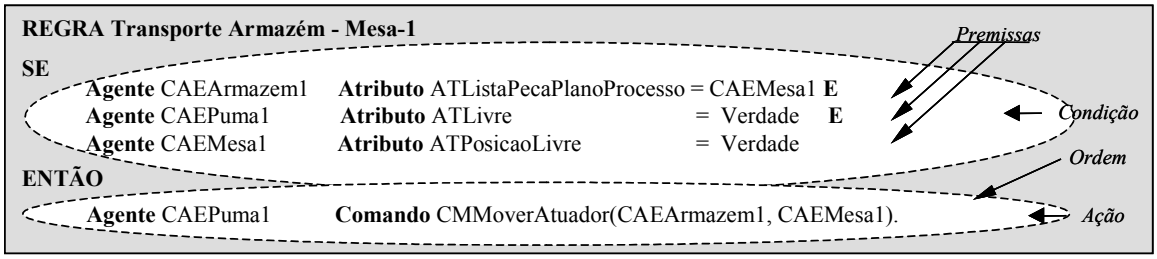


Figura 18 – Regra para decidir e coordenar transporte de peças entre Armazém e Mesa-1

A Figura 19 apresenta outra instância de Regra responsável pela Decisão/Coordenação de transporte de peça da Mesa-1 para o Centro de Usinagem, bem como o processamento da peça neste último equipamento. O conhecimento desta Regra é: se existe uma ou mais peças na Mesa-1 cujos planos de processo indicam o Centro de Usinagem como próximo equipamento a ser visitado, se o Centro de Usinagem está livre e se o KUKA386 está livre, então o KUKA386 deve transportar uma das peças da Mesa-1 para o Centro de Usinagem e esta deve processar a peça segundo parâmetros específicos.

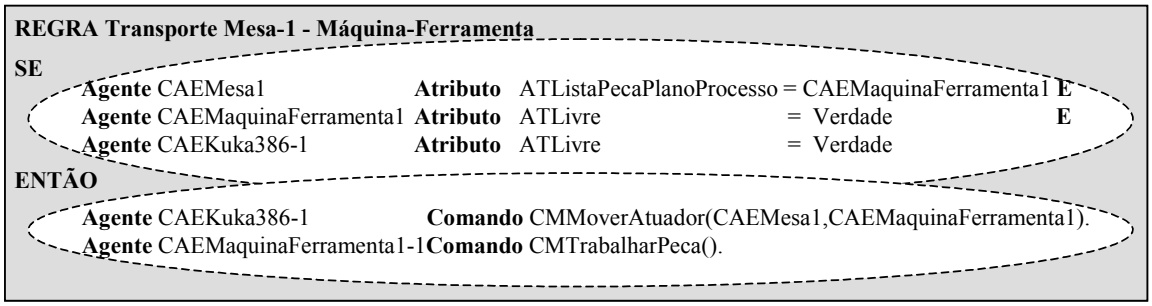


Figura 19 - Regra para decidir e transportar peca da Mesa-1 para Centro de Usinagem.

Cada Regra é composta por uma Condição e uma Ação. Na Condição há uma série de Premissas e na Ação uma série de Ordens. A Premissa pode ser: (i) Premissa Simples, que compara (ou avalia) o ***at*** de um ***abf*** com um determinado valor via algum operador; ou (ii) Premissa Composta, que correlaciona valores de ***at***s. A Ordem tem a função de instigar um ***mt*** ou um ***cm*** de um ***abf*** para realizar suas atribuições.

Em termos de Controle Supervisório, cada Condição representa a parte da Decisão, enquanto cada Ação representa parte da Coordenação. Portanto, a regra representa a relação causal entre estas partes. Todo o conhecimento pertinente a Decisão/Coordenação advém do conjunto de Regras do CS-SAM.

Para especificar um diagrama de classes genérico para comportar a Decisão/Coordenação, seguindo os princípios de SBR, pode-se observar regras específicas e delas obter elementos que comportem as funcionalidades genéricas desejadas.

Neste trabalho propõe-se uma estrutura de classes que permita instanciar objetos que comportem o conhecimento expresso em regras no formato apresentado. Os relacionamentos expressos no diagrama de classes devem permitir aos agentes realizarem uma inferência rápida e eficiente, uma vez que o CS-SAM pode necessitar de tempo curto de resposta.

Decisão/Coordenação no Padrão Arquitetural

O modelo da Decisão/Coordenação possui uma classe ***Regra*** para representar cada regra do SBR. As instâncias desta classe são chamadas de ***ar***s (agentes regra). A ***Regra*** agrega duas classes, a ***Condicao*** e a ***Acao***, cujas instâncias são designados por ***ac***s (agentes condição) e ***aa***s (agentes ação). Cada ***ac*** apresenta uma relação causal com um **aa** (Figura 20).

O ***ac*** faz o cálculo lógico do ***ar*** que o possui. Cada ***ac*** é conectado a um ou mais ***ap***s (agentes premissa) oriundos da classe ***Premissa***. Um ***ap*** possui: (i) um valor booleano sobre si mesmo, (ii) um apontamento a um único ***at*** (a *Referência*), (iii) um operador lógico (o *Operador*) e (iv) um valor ou um limite de valores (o *Valor*). O ***ap*** calcula seu valor booleano comparando o *Valor* e a *Referência* por meio do *Operador* (i.e. *Premissa Simples*). Outro recurso do ***ap*** é o *Valor* ser referência a um outro ***at***, permitindo correlações (i.e. *Premissa Composta*). Enfim, o ***ac*** faz o cálculo lógico através da conjunção dos valores booleanos dos ***ap***s conectados.

O ***aa*** realiza coordenações por meio de uma seqüência de ***aos*** (agentes ordem) associados que, além de serem instâncias da classe ***Ordem***, comandam assincronamente ***abf***s citados no respectivo ***ac***, via ***mt***s ou ***cm***s, para realizarem suas atribuições. Um ***aa*** só é passível de execução se o respectivo ***ac*** estiver em estado de verdade.

Outras características de um ***ar*** são: (i) autodestruição se um dos agentes referenciados no seu ***ac*** deixar de existir; (ii) modularidade de escopo (*i. e.* os ***ar***s de uma instância de um ***ElHi*** não enxergam os ***ar***s de outra instância); (vi) ***ap***s, conectados ao seu ***ac***, compartilhados com ***ac***s de outros ***ar***s, quando pertinente; e (vii) ***ao***s conectados ao seu ***aa***, compartilhados com ***aa***s de outros ***ar***s, também quando pertinente (estes compartilhamentos colaboram no processo de inferência).

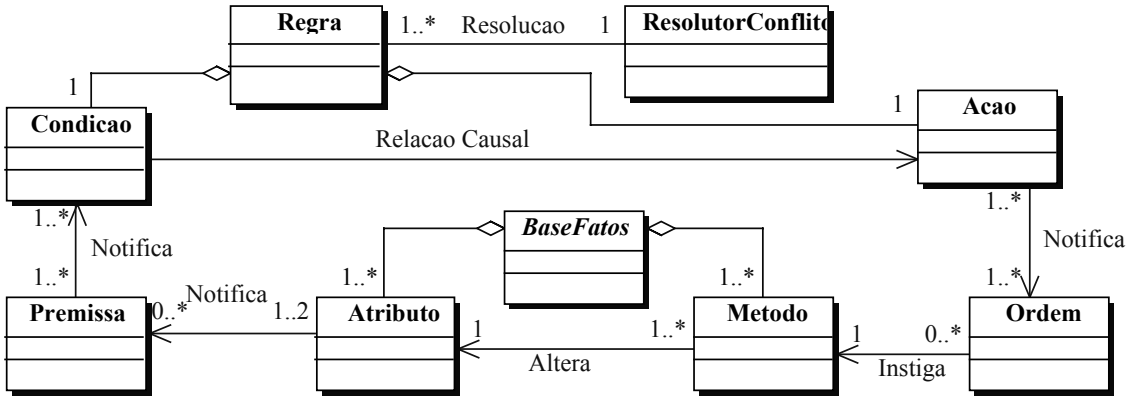


Figura 20 - Diagrama de classes para Decisão/Coordenação

A dinâmica de interações entre os agentes pode gerar conflitos. Um conflito ocorre quando um ***ap*** tem a *Referência* oriunda de um ***abf exclusivo*** (i.e um ***abf*** para recurso compartilhado, como um robô que atende a dois equipamentos) e está sendo utilizado por ***ar***s em estado de verdade (*elegíveis*). Por exemplo: a *Premissa* "**Agente** CAEKUKA386-1 **Atributo** ATLivre = Verdade" que hipoteticamente colabora para ***ar***s serem *elegíveis*, sendo o CAEKUKA386-1 exclusivo. Para resolver o impasse, escolhe-se um ***ar*** para ser ativado (*o eleito*), tomando como base alguns parâmetros de decisão (e.g. um escalonador dinâmico ou políticas de controle). Tendo *o eleito*, os demais até então *elegíveis* passam a ser considerados falsos ou *inelegíveis*. Toda a política de resolução de controle, neste padrão arquitetural, é encapsulada em uma agente do tipo ***ResolutorConflito*** (Figura 20).

A dinâmica de interações entre os agentes ainda permite uma inferência robusta em comparação com o modelo usual, em SBR, de pesquisar toda a base de fatos (Rich et Knight, 1991) ou mesmo com os modelos mais avançados, como por exemplo a eficiente abordagem proposta por Forgy (Forgy, 1982), intitulada *RETE network*.

Os agentes realizam a inferência utilizando notificações: o ***abf*** avisa aos ***at***s que devem mudar de valor quando percebem alguma mudança que os interessa no elemento monitorado (n1 na Figura 21). O ***at***, que sabe quais ***ap***s têm interesse na mudança de seu estado, notifica os ***ap***s interessados quando a mudança de estado ocorre (n2). Os ***ap***s notificados recalculam, então, seus valores lógicos. Cada ***ap*** sabe quais **ac**s têm interesse na sua mudança de valor booleano. Portanto, quando um ***ap*** muda de estado, ele notifica os ***ac***s interessados (n3). Cada ***ac*** notificado reavalia seu valor lógico e em caso de alteração avisa seu ***ar***. O ***ar*** em estado de "verdade" habilita seu ***aa*** a ser executado.

O relacionamento dos agentes cria um grafo de conexões, permitindo uma inferência ágil, sem buscas através de propagações de mensagens. Esta inferência consiste na dinâmica principal das instâncias criadas a partir deste padrão arquitetural.

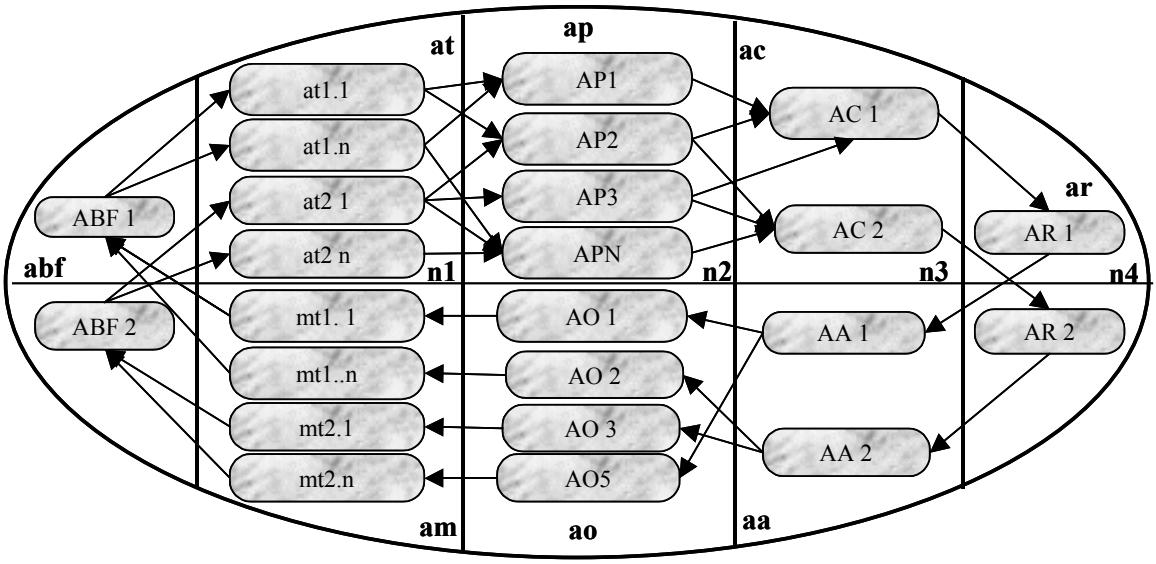


Figura 21 - Princípio de Notificações em Diagrama de Objetos

5 Conseqüências

Este artigo apresentou um padrão arquitetural para Controle Supervisório de Sistema Automatizados de Manufatura. Definiu-se uma extensão da UML para tratar a genericidade na análise de requisitos para a composição do padrão arquitetural. Esta extensão traz como benefício a capacidade de expressão genérica de requisitos funcionais através da qual vários diagramas de casos de uso específicos podem ser representados em um único modelo.

O padrão arquitetural proposto pode ser classificado como: (i) multicamada, com uma camada de Monitoração/Comando e uma camada de Decisão/Coordenação; (ii) baseado em regras, uma vez que o padrão é modelado segundo um SBR genérico; (iii) orientado a objetos; (iv) orientado a agentes, uma vez que grupos coesos de objetos são apresentados como agentes; e (v) dirigido a eventos, pois os agentes reagem e cooperam por estímulo de eventos.

O artigo apresentou um padrão arquitetural para uma área importante em computação relacionada com a automação industrial. Contribuições para concepção de sistema de controle supervisório são necessárias em razão da complexidade de desenvolvimento de sistemas computacionais para este fim.

O padrão arquitetural apresentado inclui conceitos de inteligência artificial, uma vez que o modelo de solução adotado é o de um sistema baseado em regras genérico para realizar CS-SAM. Este modelo emprega o conceito de agentes na instanciação das classes e adota um mecanismo avançado de inferência por meio de notificações.

A robustez do padrão arquitetural constituído, bem como a eficácia dos sistemas instanciados a partir do padrão, têm sido observados em sistemas de Controle Supervisório aplicados sobre simulações de plantas industriais em ANALYTICE II, incluindo a planta apresentada como exemplo neste trabalho. Também se pretende aplicar este padrão arquitetural a sistemas reais.

Como trabalhos paralelos, está sendo definida uma metodologia para a concepção de padrões arquiteturais segundo a abordagem apresentada e a definição de um padrão genérico

para concepção e realização de sistemas baseados em regras, consistindo basicamente, do uso dos níveis mais genéricos do padrão arquitetural proposto em outras áreas de aplicação.

Trabalhos futuros incluem a definição de um modelo de distribuição computacional do padrão arquitetural, desenvolvimento de um ambiente para instanciação de sistemas especialistas segundo o padrão, e enquadramento dos módulos constituintes do padrão arquitetural segundo os padrões de projeto existentes na literatura. Prevê-se, também, o desenvolvimento de padrões arquiteturais para a concepção e realização de outros sistemas de gestão de informação voltados a SAM, tais como o Planejamento, o Escalonamento e a Supervisão, de forma integrada ao padrão de Controle Supervisório proposto.

6 Padrões Relacionados

A literatura especializada em arquiteturas de *software* para CS-SAM não é vasta. Existem várias políticas para composição de CS-SAM, mas são muito específicas, não podendo ser consideradas como arquiteturas de *software*.

Dentre as arquiteturas propostas existem as holônicas. A arquitetura holônica é considerada uma arquitetura ágil. Uma arquitetura ágil para manufatura é aquela na qual encontram-se propriedades acentuadas de autonomia, organização, tolerância à falhas, adaptação, comunicação, dentre outras que permitem a concepção de sistemas de manufatura mais flexíveis e ativos. Estes sistemas têm sido inspirados em sistemas naturais (e.g. biológicos e sociais) que possuem as características citadas e envolvem os conceitos de hierarquia, heterarquia, agentes e multi-agentes (Bongaerts, 1998) (Sorensen et Langer, 1998) (Wyns, 1999).

No caso específico de Controle Supervisório, há a proposta de Sorensen (Sorensen et Langer, 1998) que consiste de uma hierarquia de classes tratando da divisão em subconjuntos similares aos apresentados (i.e. monitoração/comando e decisão/coordenação). Entretanto, não é explicitada uma forma padronizada de interação entre os módulos, a dinâmica entre eles é abordada em termos mais abstratos, necessitando de especializações que resolvam a questão (como por exemplo o padrão arquitetural proposto). Esta abordagem consiste-se, de fato, em uma arquitetura de *software* importante no tocante ao modelo holônico, porém não a apresentada na forma de um padrão arquitetural.

Uma solução encontrada na literatura é o artigo apresentado por Schmid (Schmid, 1995). Similarmente à proposta de Sorensen, naquele artigo são tratados os aspectos de divisão estrutural dos constituintes genéricos de um Controle Supervisório de Manufatura, porém com um grau de especialização maior. Apesar de ser um pouco mais detalhado, ainda não são dadas soluções efetivas para a modelagem, instanciação e realização da dinâmica de interação dos elementos. Por fim, a abordagem de Schmid ainda não é apresentada na forma de um padrão arquitetural, mas sim na forma de uma arquitetura de *software* composta por padrões de projeto.

Aarsten (Aasten et al., 1995) apresenta um framework, quase no formato de um padrão arquitetural, muito robusto e aplicado a CS-SAM. Nesta abordagem já é tratada a questão do relacionamento dinâmico entre os constituintes do CS-SAM, apresentando uma forma detalhada de concepção, de instanciação e de realização do controle. Um ponto positivo da proposta de Aarsten é a preocupação com detalhes observada desde a análise até a

implementação do controle supervisório, tanto no tocante a simulações quanto a aplicações reais. Ainda no que tange a implementação, há detalhes importantes de integração, como questões relativas à banco de dados e distribuição computacional. Todavia, não é tratada uma forma de expressão da decisão e nem mecanismos que permitam esta decisão apresentar inferências avançadas.

Aaesten (Aarsten et al., 1996) apresenta uma aplicação especializada de um padrão apresentado em (Aarsten, 1995). Em suma o trabalho consiste da uma especialização e respectiva aplicação do referido padrão para a situação de controle supervisório de robôs móveis.

Brugali (Brugali et al., 1997) apresenta algumas considerações e evoluções (no sentido de aplicações) dos trabalhos apresentados por eles e seus parceiros em artigos prévios (Aersten, 1995) (Aersten, 1996).

7 Referências Bibliográficas

Aarsten, A., Brugali, D. e Menga, G. *Designing Concurrent and Distributed Control Systems: an Approach Based on Design Patterns.* In Communications of the ACM - Special Issue on Design Patterns. 1996.

Aarsten, A., Elia, G. e Menga, G. *G++: A Pattern Language for the Object Oriented Design of Concurrent and Distributed Information Systems, with Applications to Computer Integrated Manufacturing* . In J. Coplien and D. Schmidt (eds.) Pattern Languages of Program Design. Addison-Wesley, 1995.

Bongaerts, L. *Integration of Scheduling and Control, In Holonic Manufacturing Systems*. Ph.D. Thesis - KatholiekeUniversiteit Leuven, 1998.

Brugali, D., Menga, G e Aarsten, A.. *The Framework Life Span: A Case Study for Flexible Manufacturing Systems*. In Communications of the ACM. Outubro 1997.

Buschmann F., Meunier R., Rohnert H., Sommerlad P. e Stal, M. *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons Ltd., 1996.

Chaar, J. K., Teichroew, D. e Volz, R. A. *Developing Manufacturing Control Software: A Survey and Critique*, The International Journal of Flexible Manufacturing Systems, Kluwer Academic Publishers, 1993. pp. 053-088.

Cheesman, J. e Daniels, J. *UML Components – A Simple Process for Specifying Component-Based Software*. Addison Wesley, 2001.

Coplien, J. e Schmidt, D. (eds.) *Pattern Languages of Program Design*, Reading-MA. Addison-Wesley, 1995.

Cury, J. E. R., de Queiroz, M. H. e Santos, E. A. P. *Síntese Modular do Controle Supervisório em Diagrama Escada para uma Célula de Manufatura*. V Simpósio Brasileiro de Automação Inteligente, Canela, RS, Brasil, 2001.

Forgy, C. L. *RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.* Artificial Intelligence, v.19,1982. p. 17-37.

Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1996.

Franklin, S. e Graesser, A. Is it an Agent, or Just a Program? A Taxonomy for Autonomous Agents, Proceedings of the 3th International Workshop on Agent Theories, Architectures and Languages, Springer-Verlag, 1996.

Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

Koscianski, A., Rosinha, L. F., Stadzisz, P. C., Künzle, L. A. FMS Design and Analysis: Developing a Simulation Environment In: Proceedings of the 15th International Conference on CAD/CAM, Robotics and Factories of the Future, Águas de Lindóia, v.2. 1999. p.RF25 - RF210.

Künzle, L. A. *Controle de Sistemas Flexíveis de Manufatura - Especificação dos níveis equipamento e estação de trabalho*, Dissertação de Mestrado, CEFET/PR, 1990.

Langer, G., Sorensen, C., Schnell, J. e Alting, L. *Design of a Holonic Shop Floor Control System for a Steel Plate Milling-Cell*, In : 2000 Int. CIRP Design Seminar on Design with Manufacturing: Intelligent Design Concepts Methods and Algorithms, Israel, 2000.

Martin, R.C.; Riehle, D. e Buschmann, F. (eds.) *Pattern Languages of Program Design 3*. Reading-MA. Addison-Wesley, 1997.

Mendes, R. S. *Modelagem e Controle de Sistemas a Eventos Discretos - Manufatura integrada por computador*, Belo Horizonte, Fundação CEFET-MG, 1995.

Miyagi, P. E. *Controle Programável – Fundamentos do Controle de Sistemas a Eventos Discretos*, Edgard Blücher, 1996.

Müller, J. P. *Architectures and Applications of Intelligent Agents: A Survey*. International House. Ealing London W5 5DB. Knowledge Engineering Review, 1998.

Pan, J., DeSouza G. N., Kak, A. C. *FuzzyShell: A Large-Scale Expert System Shell Using Fuzzy Logic for Uncertainty Resoning*. IEEE Transactions on Fuzzy Systems, Vol. 6. No 4, 1998.

Rich, E. e Knight, K. *Artificial Intelligence*, McGraw-Hill, 1991.

Rumbaugh, J., Jacobson, I. e Booch, G. *The Unified Modelling Language Reference Manual*. Addison Wesley Longman, 1999.

Russell, S. e Norvig, P., *Artificial intelligence: A Modern Approach*. Prentice Hall, 1995.

Schmid, H. A. *Creating the Architecture of a Manufacturing Framework by Design Patterns*. Fachbereich Informatik, Fachhochschule konstanz. OOPSLA'95, 1995.

Simão, J. M. *Proposta de uma Arquitetura para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes*. Dissertação de mestrado. CPGEI/CEFET-PR, 2001.

Sorensen C., Langer C. G. *Developing a System Architecture for Holonic Shop Floor Control*. International Federation of Automatic Control – Preprints of The 5th IFAC Workshop On Intelligent Manufacturing Systems, Gramado - RS, Brazil, November. 1998.

Vlissides, J.; Coplien, J.; Kerth, N (eds.). *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

Wyns J. *Reference Architecture for Holonic Manufacturing Systems - The Key to Support Evolution and Reconfiguration*. Ph. D. Thesis PMA/Katholieke Universit Leuven. 1999.

Yufeng, L. e Shuzhen, Y. *Research on the Multi-Agent Model of Autonomous Distributed Control System*, In 31 International Conference Technology of Object-Oriented Language and Systems. IEEE Press. China. 1999.

A Queue-based Algorithmic Pattern

Marcos C. d'Ornellas[1]
ornellas@inf.ufsm.br

Grupo de Pesquisas em Processamento de Imagens (PIGS)
Laboratório de Ciências Espaciais de Santa Maria (LACESM)
Universidade Federal de Santa Maria (UFSM)

Abstract

This paper briefly discuss the general class of algorithms that can be implemented using queue-based constructions. Common characteristics of these algorithms are also described in order to provide a generic representation for queue-based algorithms. In addition it describes the queue-based pattern in terms of wavefront propagation and its relationship within mathematical morphology. Such a pattern is essential for the development of morphological operators and operations. Examples of pattern usage are contour processing, morphological reconstruction, and watersheds for grey-scale and color images to name a few.

Keywords: *mathematical morphology, software development, implementation techniques, algorithmic patterns, and generic programming.*

1 Name

Queue-based

2 Intent

Serve as a foundation to provide a generic representation for queue-based algorithms. Queue-based provides flexibility in terms of requirements for queue-based algorithms and govern their effective implementation. In addition it enhances morphological algorithm reusability.

3 Also Known As

Wave-front propagation or just propagation.

4 Motivation

In parallel implementations, computational power is wasted because, at each iteration, only a small fraction of the processed pixels actually change values. This is even the case for the sequential ones. Although it is more efficient for operator iterations than the parallel pattern is [5]. Apart from that, morphological image operators implemented this way, are often iterated several times to achieve stability, which make them inefficient in practice. Therefore, reasonable

[1]Financial support from FAPERGS Process n. 02/0287.0

solutions not only in the algorithmic design but also in the implementational techniques are needed.

Algorithms that incorporate image scanning techniques based on queues have been proposed in the literature [8] [9] [28] [24] [27] to overcome the implementation constraints related to parallel and sequential implementations. Queue-based algorithms take advantage of the fact that the image data are finite and totally ordered. The algorithms guarantee that pixels that effectively contribute for the output results are processed only once. Queue-based algorithms can also be treated as a specialization of sequential implementations where the scanning order is derived from a predefined ordering relationship on the pixel intensities.

A priority-based queue is accomplished by first placing all pixel addresses related with a given intensity value into a large array of pixels. Then, a distributive procedure makes use of address calculations to get the frequency distribution of image intensities. This method allows for breadth-first propagation[2] at any pixel intensity with the use of a single queue. Another alternative for the data structure relies on hierarchical queues [1] [3] which make use of a double ordering relation for every pixel in the image. Consequently, queue-based data structures improve significantly the efficiency of morphological algorithms.

5 Applicability

The queue-based pattern may be used whenever a morphological operator can be implemented using a wavefront propagation interpretation from a set of seeds or markers. This operator is geared towards efficiency since relevant pixels that actually contribute for the output results are processed only once.

Algorithms that use a queue-based pattern share a certain number of characteristics, which are listed in the following:

- **Total Ordering Relation**: algorithms presume that the image data to be processed is totally ordered and finite. Every image pixel may contain additional information other than its pixel position and intensity, which might be used to order the image data;
- **Queue-based Data Structure**: algorithms utilize a queue-based data structure to speed up computations. Such queue-based structure contains only the relevant pixels, which are needed to be processed. In other words, image pixels are processed only once;
- **Two-step Algorithms**: an algorithm contains two steps namely initialization and data-driven propagation. The first step selects the pixels for the initial wavefront. The second step propagates the wavefront based on the image data and is bound to produce the result;
- **Wavefront Propagation**: wavefront propagation implies that the action of the morphological operator is closely associated with the notion of connectivity. The basic operation is the updating of all neighboring pixels based on the value of the propagated pixel;
- **Iteration Until Stability**: the data-driven propagation step after the initialization process assures that the algorithm will reach stability after a number of iterations (governed by a `while ...do` loop) or when a certain condition is met;

[2]The propagation is known since long as an iterative procedure until stability is reached. The computation is a growing process and is based on the propagation of sets of pixels called markers through a mask image. Algorithms using this approach have been described by a number of authors in the literature leading to efficient implementations [15] [21].

- **Origin Included in the Support**: algorithms assume that the origin is included in the structuring element support, which is important because only anti-extensive erosions and extensive dilations are considered.

6 Structure

The generic representation for queue-based algorithms given in figures 1 and 2 contain elementary information that needs to be described in order to make the queue-based pattern.

So far, the discussion is focused on the application of wavefront propagation in mathematical morphology. Indeed, there are two generic design structures for the queue-based algorithms [14]. These structures are characterized by whether the data-driven propagation is embedded or not in the initialization step.

The case of queue-based algorithms are shown in figure 1 and 2. Morphological applications related to the algorithm in figure 1 are: contour processing methods, morphological reconstruction, area opening and area closing, dynamics, levelings, fast marching methods, and watersheds to name a few. Connectivity analysis and extrema detection are examples of applications related to the algorithm in figure 2.

Figure 1: Queue-based algorithm showing a generic representation of the wavefront propagation. Note that the algorithm includes two steps: an initialization and a data-driven propagation.

```
Initialization:
...
for all (pixel p ∈ D_f) do
  if (condition1) then
    q.enq(f(p), p)
    ...
  end if
end for

Data-driven Propagation:
while (!(q.empty())) do
  p = q.deq()
  for all (p' ∈ N(p)) do
    if (condition2) then
      ...
      region settings
      q.enq(f(p'), p')
    end if
  end for
end while
```

Figure 2: Queue-based algorithm showing an alternative generic representation of the wavefront propagation. Note that the data-driven propagation is mixed in the initialization step.

```
for all (pixel p ∈ D_f) do
  if (condition1) then
    q.enq(f(p), p)
    ...
    while (!(q.empty()) do
      p = q.deq()
      for all (p' ∈ N(p)) do
        if (condition2) then
          ...
          region settings
          q.enq(f(p'), p')
        end if
      end for
    end while
  end if
end for
```

7 Participants

The queue-based pattern includes the following elements:

- **Iterator**: an iterator is a fundamental structure that abstracts the process of moving through a finite set of elements [7]. Iterators in the generic representation for queue-based algorithms are highly influenced by the queue-based data structures used in the wavefront propagation since these structures provide a mechanism to control the scanning order of pixels in the image. A priority queue is the data structure chosen for this pattern;

- **Pixel Lattice**: the pixel lattice $\mathcal{L}$ includes, among other elements like those that the value set V and the ordering $\leq$, the notions of the supremum operator $\bigvee$ and infimum operator $\bigwedge$ working on subsets of V. The largest of all elements in V is called lattice supremum, denoted as $\bigvee_{\mathcal{L}}$ while the smallest of all elements in V is called lattice infimum, denoted as $\bigwedge_{\mathcal{L}}$. All these elements must be explicitly defined by $\mathcal{L} = (V, \leq, \bigvee, \bigwedge, \bigvee_{\mathcal{L}}, \bigwedge_{\mathcal{L}})$;

- **Adjunction**: the adjunction is represented by $\mathcal{A} = (N)$ since algorithms using the wavefront propagation interpretation are extensions of flat morphology, using flat and symmetric structuring elements. Note that N specifies the connectivity, i.e. the number of pixels needed in neighborhood operations for every pixel in the priority queue.

These three elements form the building block representation of the generic queue-based pattern. The computational complexity of algorithms using a queue-based pattern is linear with the number of pixels put in the priority queue, which is responsible for the data-driven propagation.

8 Collaborations

- **Iterator** is coupled with queue-based data structures involved. All the data must be accessed by the iterator when an algorithm is implemented in both initialization and data-driven propagation. In addition, the **pixel lattice** serves as a framework for the development of the algorithms;

- **Pixel Lattice** defines the value set used for the data involved. In other words, it determines a set of common rules to be used in the wave-front propagation for a family of data types;

- **Adjunction** is closely related with the way a propagation step is conducted. Therefore, it works together with the **iterator**.

9 Consequences

- Some algorithms share a certain number of characteristics, which are listed as follows: total ordering relation, queue-based data structure, two-step algorithms, wavefront propagation, iteration until stability, and origin included in the support. An algorithm that has these characteristics is termed a queue-based algorithm. The set of characteristics serves as a basis to provide a generic representation for queue-based algorithms;

- A generic representation for queue-based algorithms utilizes wavefront propagation. It implies that the action of the morphological operator is closely associated with the notion of connectivity and is based on a kind of growing process, where the information is propagated through the image. Two possible generic representations for queue-based algorithms were introduced in order to describe the queue-based pattern. These representations are characterized whether the data-driven propagation is embedded or not in the initialization step;

- A queue-based pattern is constructed in order to provide flexibility in terms of requirements for queue-based algorithms and govern their effective implementation;

- A generic programming approach might be applied to morphological image operators based on wavefront propagation like contour processing methods, morphological reconstruction, and watersheds. Generic programming tools as `C++` with `STL` applies a more evolutionary and experimental approach to morphological algorithm development. The proposed queue-based pattern enhances morphological algorithm reuse.

- Care must be taken when a wavefront propagation interpretation is applied to an image with too many constraints (e.g. lines, areas, or polygons). In this case, the time complexity is proportional to the number of steps needed to end up the propagation process.

10 Implementation

Morphological operator design must comply with the complete lattice framework theory, i.e. algorithmic implementations must be tied to the generic pattern representation that includes the iterator, the pixel lattice, and the adjunction. Iterators in the generic representation for queue-based algorithms are highly influenced by the queue-based data structures used in the wavefront propagation since these structures provide a mechanism to control the scanning order of pixels in the image. This section gives an overview of common queue-based data structures like ordinary queues, priority queues and hierarchical queues. Later, the iterator is combined with the pixel lattice and the adjunction, producing the queue-based pattern.

The following implementation issues are relevant for the queue-based pattern:

Data Structures: Queues are often used in the implementation of efficient and reliable morphological image operators [22] [24]. Items stored in the queue are pixel addresses or pixel

values, but the queue must be able to handle other features according to the implementation. The size of the queue can be fixed [28] or be controlled dynamically [19]. There are various ways to implement priority queues. The easiest one is to use linked lists, consistently keeping the elements in the list in order of descending priority. A faster approach is to store the data in binary trees, with an ordering property imposed to make sure that the highest-priority element is always easily accessible. The appropriate ordering property is that the element stored at any node of the binary tree should have a priority greater than or equal to that of any element stored in either one of its subtrees. A binary tree that has this property is called a *heap*.

C++ and STL: The STL is a relatively small library which achieves a remarkable degree of reuse through its basis in the principles of generic programming and its use of C++ templates. Because of this, it has a particularly clear shape. The distinction between containers, iterators and algorithms is its most striking structural feature: dynamically, the way a container delivers iterators which are then used by algorithms is a consistent and fundamental pattern of use.

11 Algorithm Samples and Usage

This section deals with examples of morphological operators and operations constructed using a queue-based pattern. The examples covered are contour processing, morphological reconstruction, and watersheds.

Contour Processing Operations: The contour processing approach was introduced in mathematical morphology in [8] [9] [28]. Given a binary image X, it is processed using a raster scanning order and transitions from white (background) to black (foreground) are detected. Pixels belonging to the contour of X, denoted by $\partial(X)$, are then stored in a simple queue, which contains their coordinates and other additional information like pixel intensity.

Efficient implementations for dilations can also be obtained by following that $X \oplus A = X \bigcup(\partial(X) \oplus A)$. This is valid for any connected structuring element that contains the origin. Figure 3 shows the original algorithms proposed in [28] for the erosion algorithm based on contour processing.

The queue-based pattern for the contour processing is given as follows:

- **Iterator**: iterators are highly influenced by the priority queue used in the wavefront propagation since it provides a mechanism to control the scanning order of pixels in the image;
- **Pixel Lattice**: all the elements must be explicitly defined by $\mathcal{L} = (V, \leq, \bigvee, \bigwedge, \bigvee_{\mathcal{L}}, \bigwedge_{\mathcal{L}})$. For instance:

$$\mathcal{L} = ([0,1], \leq, \bigvee, \bigwedge, 1, 0)$$

- **Adjunction**: the adjunction is represented by $\mathcal{A} = (N)$. Erosions and dilations are given by:

$$(\varepsilon f)(x) = \bigwedge_{y \in N(x)} \{f(y)\} \quad (1)$$

$$(\delta f)(x) = \bigvee_{y \in N(x)} \{f(y)\} \quad (2)$$

Figure 3: A queue-based algorithm representation using the contour processing method for the erosion.

Note(s): *f - input image; $q1, q2$ - auxiliary queues, $N(p)$ - neighborhood of p.*

Queue Initialization:
$border = 0$
$Q \;\; q1, q2$
for all (pixel $p \in D_f$) **do**
  if ($f(p)! = 0$ and $(\exists(p') \in N(p) : f(p') == 0)$) **then**
    $q1.enq(p)$
  end if
end for
for all (pixel $p \in q1$) **do**
  $f(p) = 0$
end for

Data-driven Propagation:
while ($!(q1.empty())$) **do**
  $p = q1.deq()$
  for all (pixel $p \in q1$) **do**
    for all ($p' \in (N(p) \cap D_f)$) **do**
      if ($f(p') == 1$) **then**
        $f(p') = 0$
        $q2.enq(p)$
      end if
    end for
  end for
  $q1 = q2$
end while

Morphological Reconstruction: Many algorithms have been proposed in the literature for image segmentation, but just a few have shown widespread applicability. One method that often guides the segmentation process within mathematical morphology is morphological reconstruction. This algorithm takes a segmentation that has too many small regions and uses a heuristic evaluation function to combine regions with low local contrast. This contrast measure is computed as a function of the minimum-edge height values of regions and their boundaries. Thresholding the contrast measure at different levels produces segmentations at various scales of detail [19].

Definition 1 (Morphological Reconstruction) *The reconstruction by dilation of a mask image g from a marker image f ($D_f = D_g$ and $f \leq g$) is defined as the geodesic dilation of f with respect to g until stability is reached and is denoted by $\rho_g(f)$:*

$$\rho_g(f) \;\; = \;\; \bigvee_{n \geq 1} \delta_g^n(f) \tag{3}$$

Definition 2 (Dual Morphological Reconstruction) *The dual reconstruction or reconstruction by erosion of a mask image g from a marker image f ($D_f = D_g$ and $f \geq g$) is defined as the geodesic erosion of f with respect to g until stability is reached and is denoted by $\rho_g^*(f)$:*

$$\rho_g^*(f) \;\; = \;\; \bigwedge_{n \geq 1} \varepsilon_g^n(f) \tag{4}$$

When dealing with binary semantics, a morphological reconstruction is easily obtained by applying the wavefront propagation interpretation along with a queue-based data structure. In such case, it works like contour processing algorithms since pixels belonging to the contour(i.e. the marker image) are put in the queue in the initialization process. A second step is the data-driven propagation, where these pixels are propagated according to their connected components. The algorithm for binary reconstruction using a queue-based pattern is given in figure 4.

Figure 4: Binary reconstruction algorithm using a queue-based pattern.

Note(s): *g - binary mask image, f - binary marker image, $f \subseteq g$. The result is given in f.*

Initialization:
for all (pixel $p \in D_g$) **do**
  if $((f(p) == 1)$ and $(\exists q \in N(p) \mid f(q) == 0)$ and $(g(p) == 1))$ **then**
    $q.enq(p)$
  end if
end for

Data-driven Propagation:
while $(!(q.empty()))$ **do**
  $p = q.deq()$
  for all $(q \in N(p))$ **do**
    if $((f(q) == 0)$ and $(g(q) == 1))$ **then**
      $f(q) = 1$
      $q.enq(p)$
    end if
  end for
end while

It was shown in [3] [26] that the extension to grey-scale semantics can be achieved in a similar manner. Instead of propagating the contours of feature objects, propagation starts from the regional maxima of the image. The grey-scale reconstruction algorithm using a queue-based pattern is given in figure 5.

The queue-based pattern for the morphological reconstruction is given as follows:

- **Iterator**: iterators are highly influenced by the priority queue used in the wavefront propagation since it provides a mechanism to control the scanning order of pixels in the image;
- **Pixel Lattice**: all the elements must be explicitly defined by $\mathcal{L} = (V, \leq, \bigvee, \bigwedge, \bigvee_{\mathcal{L}}, \bigwedge_{\mathcal{L}})$. For binary and grey-scale reconstruction, the pixel lattice is given by:

$$\mathcal{L} = ([0,1], \leq, \bigvee, \bigwedge, 1, 0)$$
$$\mathcal{L} = ([0,255], \leq, \bigvee, \bigwedge, 255, 0)$$

- **Adjunction**: the adjunction is represented by $\mathcal{A} = (N)$. Erosions and dilations are given by:

$$(\varepsilon f)(x) = \bigwedge_{y \in N(x)} \{f(y)\} \quad (5)$$
$$(\delta f)(x) = \bigvee_{y \in N(x)} \{f(y)\} \quad (6)$$

Figure 5: Grey-scale reconstruction algorithm using a queue-based pattern.

Note(s): *g - grey-scale mask image, f - grey-scale marker image (regional maxima), $f \subseteq g$. The result is given in f.*

Initialization:
for all (pixel $p \in D_g$) **do**
  if $((f(p)! = 0)$ and $(\ \exists q \in N(p) \mid f(q) == 0))$ **then**
    $q.enq(p)$
  end if
end for

Data-driven Propagation:
while $(!(q.empty()))$ **do**
  $p = q.deq()$
  for all $(q \in N(p))$ **do**
    if $((f(q) \leq f(p))$ and $(g(q)! = f(q)))$ **then**
      $f(q) = min(f(p), g(q))$
      $q.enq(p)$
    end if
  end for
end while

Watershed Segmentation: Watershed analysis has proven to be a powerful tool for many image segmentation problems. In the flooding scheme [20], water slowly rises within the topographic surface represented by an image, so that all points below water level are immersed. Holes are punched in the regional minima and the topography is flooded from below. As the water rises, more surface minima are pierced, which in turn starts more catchment basins. The catchment basins expand as the water rises and floods more points. When two floods from different catchment basins meet, a dam is built at these points to prevent the catchment basins from merging. After the surface is completely flooded, only the tops of the dams are visible and are treated as dividing lines. These watershed lines separate the surface into catchment basins [2] [3] [26].

Two steps can compose the bare-bones watershed implementation: initialization step, and data driven propagation. Its algorithmic representation is given in figure 6. Similar versions of this algorithm can be found in [3] [6] [12] [13] [15] [16].

The queue-based pattern for watersheds when applied to grey-scale images, is given as follows:

- **Iterator**: iterators are highly influenced by the priority queue used in the wavefront propagation since it provides a mechanism to control the scanning order of pixels in the image;
- **Pixel Lattice**: all the elements must be explicitly defined by $\mathcal{L} = (V, \leq, \bigvee, \bigwedge, \bigvee_{\mathcal{L}}, \bigwedge_{\mathcal{L}})$. For instance:

$$\mathcal{L} = ([0, 255], \leq, \bigvee, \bigwedge, 255, 0)$$

- **Adjunction**: the adjunction is represented by $\mathcal{A} = (N)$. Erosions and dilations are

Figure 6: A Watershed algorithm for grey-scale images implementing a queue-based pattern.

```
Initialization:
PQ  q
for all (pixel p ∈ D_f) do
  if (M(p)! = 0 and( ∃(p') ∈ N(p) : M(p') == 0)) then
    q.enq(f(p), p)
  end if
end for

Data-driven Propagation:
while (!(q.empty()) do
  p = q.deq()
  for all (p' ∈ (N(p) ∩ D_f)) do
    if (M(p') == 0) then
      M(p') = M(p)
      q.enq(f(p'), p')
    end if
  end for
end while
```

given by:

$$(\varepsilon f)(x) = \bigwedge_{y \in N(x)} \{f(y)\} \tag{7}$$

$$(\delta f)(x) = \bigvee_{y \in N(x)} \{f(y)\} \tag{8}$$

12 Known Uses

Efficient algorithms in image processing are known to be designed and implemented using queue-based data structures. Several queue-based operators and operations have been proposed in the literature leading to a large collection of applications, which are characterized as follows:

contour processing methods: Contour processing methods were introduced in [9] [28] [21] with applications for fast binary neighborhood operations including erosion, dilation, distance transforms, and skeletons. Other possible applications of contour processing methods like contour filling and object labeling become available by slightly modifications of the basic operations.

morphological reconstruction: Reconstruction is a powerful morphological tool. The observation that the reconstruction operator could be implemented using design techniques based on queues gave birth to another important application of queue-based algorithms. With respect to binary images, reconstruction has been used to remove object features connected to the image border, hole filling, image labeling, skeleton pruning, and ultimate erosion to name a few [19]. Other applications involving grey-scale images have been proposed by [22] [25] like connected filtering, extrema detection, dome and basin extraction, minima imposition, and hole filling.

area opening and area closing: Area opening and area closing find its applicability in image filtering tasks and image segmentation [23]. Area opening and area closing are also used in the construction of more complex morphological operators. An example is

the morphological attribute openings and closings, which are generalizations of the area opening and area closing, and allow filtering of images based on a wide variety of shape or size based criteria [4].

levelings: In complex image-segmentation problems, it is possible to use a composition of morphological transformations such as geodesic opening and closing operations, in order to enhance marker selection. The composition of opening and closing operations by morphological reconstruction may be considered as a recently formalized technique, called levelings [10] [11]. Applications involving marker selection like watersheds take advantage that, after the levelings, an image has homogeneous regions (i.e. flat zones), and that the levelings can be applied to the original image at different levels of simplification using any criterion like size, shape, or contrast.

fast marching methods: A fast marching level set method is presented for monotonically advancing fronts, which leads to an extremely fast scheme for solving the Eikonal equation [18]. This technique has been applied to a wide collection of problems, including construction of geodesics on surfaces, computer vision, and shape-from-shading [17].

watersheds: One of the most useful methods among the mathematical morphology algorithms is a watershed transform. It is the classical morphological method for segmentation and has been used in several applications like medical imaging, robot vision, image sequences, and so forth [3]. The idea of using the watershed method for image segmentation is that the watershed of the surface tends to follow the high ground of the original image in terms of the intensity level. Therefore, finding the watershed of the magnitude of the gradient of an image ensures these lines will follow the edges (or regions of high ground) in the image. This achieves a useful result for image segmentation;

polygon filling: Polygon filling is often executed in the image or frame buffer. It is assumed that a polygon with proper closed borders are given and that inside the polygon no pixel has the colour value with which it is to be filled. For the seed fill algorithm we need a starting point which is inside the polygon. Starting at this *seed* position, the algorithm proceeds in all directions and sets each pixel to the required fill colour until the border is reached;

robot planning: Research in robot planning has considered the interleaving of planning and execution for example, as well as the issue of planning sensor actions as a way of gaining information for planning. A major theme in robot planning is also the integration of predictive and reactive planning as a way of making activity more resilient in the face of a changing environment. Predictive planning often applies the queue-based pattern in the configuration space using a visibility graph and through free space decomposition techniques.

13 Related Patterns

Parallel and Sequential patterns [5] are also used in mathematical morphology. Although these patterns are less eficient than the queue-based pattern, they are essential for several applications ranging from distance transforms to morphological reconstruction. A queue-based pattern can only be used when the notion of propagation is embedded in the solution.

References

[1] S. Beucher. *Segmentation d'Images et Morphologie Mathématique*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, Fontainebleau, 1990.

[2] S. Beucher. Watershed, hierarchical segmentation and waterfall algorithm. In J. Serra and P. Soille, editors, *Mathematical Morphology and Its Applications to Image Processing*, pages 69–76. Kluwer Academic Publishers, The Netherlands, 1994.

[3] S. Beucher and F. Meyer. The morphological approach to segmentation: the watershed transformation. In E. R. Dougherty, editor, *Mathematical Morphology in Image Processing*, chapter 12, pages 433–481. Marcel Dekker, New York, 1993.

[4] Edmond J. Breen and Ronald Jones. Attribute openings, thinnings, and granulometries. *Computer Vision and Image Understanding*, 64(3):377–389, 1995.

[5] M. C. d'Ornellas. *Algorithmic Patterns for Morphological Image Processing*. PhD thesis, Univesiteit van Amsterdam, 2001.

[6] M. C. d'Ornellas and R. v.d. Boomgaard. Generic algorithms for morphological image operators - a case study using watersheds. In H. J. A. M. Heijmans and J. B. T. M. Roerdink, editors, *Mathematical Morphology and Its Applications to Image Processing*, pages 323–330. Kluwer Academic Publishers, The Netherlands, 1998.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In O. M. Nierstrasz, editor, *ECOOP'93: Object-Oriented Programming - Proceedings of the 7th European Conference*, pages 406–431. Springer, Berlin, Heidelberg, 1993.

[8] F. C. A. Groen and N. J. Foster. A fast algorithm for cellular logic operations on sequential machines. *Pattern Recognition Letters*, 2:333–338, 1984.

[9] L. Ji, J. Piper, and J. Tang. Erosion and dilation of binary images by arbitrary structuring elements using interval coding. *Pattern Recognition Letters*, 9:201–209, 1989.

[10] F. Meyer. From connected operators to levelings. In H. J. A. M. Heijmans and J. B. T. M. Roerdink, editors, *Mathematical Morphology and Its Applications to Image Processing*, pages 191–198. Kluwer Academic Publishers, The Netherlands, 1998.

[11] F. Meyer. The levelings. In H. J. A. M. Heijmans and J. B. T. M. Roerdink, editors, *Mathematical Morphology and Its Applications to Image Processing*, pages 199–206. Kluwer Academic Publishers, The Netherlands, 1998.

[12] L. Najman and M. Schmitt. Watershed of a continuous function. *Signal Processing*, 38:99–112, 1994.

[13] L. Najman and M. Schmitt. Geodesic saliency of watershed contours and hierarchical segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(12):1163–1173, 1996.

[14] D. Noguet. A massively parallel implementation of the watershed based on cellular automata. In *IEEE Conference on Application Specific Array Processors*, July 1997.

[15] D. Noguet, A. Merle, and D. Lattard. A data dependent architecture based on seeded region growing strategy for advanced morphological operators. In P. Maragos, R. W. Schafer, and M. A. Butt, editors, *Mathematical Morphology and Its Applications to Image and Signal Processing*, pages 235–243. Kluwer Academic Publishers, 1996.

[16] J. B. T. M. Roerdink and A. Meijster. The watershed transform: definitions, algorithms, and parallellization strategies. *Fundamenta Informaticae*, 41:187–228, 2000.

[17] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *National Academy of Sciences Journal*, pages 1591–1595, 1996.

[18] J. A. Sethian. Theory, algorithms, and applications of level set methods for propagating interfaces. *Acta Numerica*, pages 309–395, 1996.

[19] P. Soille. *Morphological Image Analysis*. Springer-Verlag, Barcelona, 1999.

[20] P. Soille and L. Vincent. Determining watersheds in digital pictures via flooding simulations. *Visual Communications and Image Processing '90*, pages 240–250, 1990.

[21] B. J. H. Verwer. *Distance Transform: Metrics, Algorithms and Applications*. PhD thesis, Delft University of Technology, 1991.

[22] L. Vincent. *Algorithmes Morphologiques à Base de Files d'Attente et de Lacets. Extension aux Graphes*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, 1990.

[23] L. Vincent. Morphological area openings and closings for grey-scale images. In Y-L. O, A. Toet, D. Foster, H. J. A. M. Heijmans, and P. Meer, editors, *Proceedings of the Workshop "Shape in Picture", 7–11 September 1992, Driebergen, The Netherlands*, pages 197–208, Berlin, 1992. Springer.

[24] L. Vincent. Morphological algorithms. In E. R. Dougherty, editor, *Mathematical Morphology in Image Processing*, chapter 8, pages 255–288. Marcel Dekker, New York, 1993.

[25] L. Vincent. Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms. *IEEE Transactions on Image Processing*, 2:176–201, 1993.

[26] L. Vincent and E. R. Dougherty. Morphological segmentation for textures and particles. In E. R. Dougherty, editor, *Digital Image Processing Methods*, pages 43–102. Marcel Dekker, New York, 1994.

[27] L. Vincent and P. Soille. Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):583–598, 1991.

[28] L. J. Van Vliet and B. J. H. Verwer. A contour processing method for fast binary neighbourhood operations. *Pattern Recognition Letters*, 7:27–36, 1988.

FEM Simulator Skeleton

Maria Lencastre (mlpm@cin.ufpe.br)[1]
Felix C. G. Santos (fcgs@demec.ufpe.br)
Isledna Rodrigues (ira2@cin.ufpe.br)
Mechanical Engineering Department, Federal University of Pernambuco
Rua Acadêmico Hélio Ramos, S/N, Recife, PE 50740-530 – Brazil

Abstract

This paper proposes a pattern, called FEM Simulator Skeleton, which describes a general set of classes, and their interactions, for guiding the development of Simulators Models based on the Finite Element Method (FEM). This pattern is intended to help the design and implementation of simulators based on what is here called algorithm skeletons, defined in section 6. By simulators we call a computational system, aimed at obtaining approximate solutions to systems of coupled partial differential equations, together with a set of restrictions (differential-algebraic relationships involving one or more vector fields). The problem is first divided into different pre-defined levels of abstraction (global solution, blocks of groups, groups of phenomena, phenomena) where the simulator designer must supply algorithm skeletons for each one of those levels. A simple example is shown in section 8 of this paper, which describes the pattern applicability in the development of a simulator for the dynamics of a rigid body attached to an elastic beam (with temperature dependent constitutive relation), where both are also submitted to thermal loads. The main advantage of the pattern is its high level of abstraction, reusability and modularity in the design of simulators for several coupled multi-physics phenomena.

1. Introduction

A Simulator can be defined as an indispensable problem-solving tool, which can be used for obtaining an approximate solution of many real world problems. In [17], Banks defines some steps that guide the model builder in a thorough simulation study process: (a) Statement of the problem; (b) Identification of questions that are to be answered by the simulation study, including the various scenarios that will be investigated; (c) Model conceptualisation, where the real world system under investigation is abstracted by: a conceptual model; a series of mathematical and logical relationship, concerning the components and the structure of the system; (d) Data Collection; (e) Model Translation, where the conceptual model constructed in step c is coded into a computer–recognizable form, an operational model; (f) Verification, which is concerned at whether the operational model is performing properly; (g) Validation, which measure the accuracy of the conceptual model as a representation of the real system.

One of the most complex problems with which scientists and engineers are concerned nowadays, related to simulation software, is the lack of tools which could fulfil their research requirements in terms of: flexibility in building different solution strategies; providing support for implementation of more suitable numerical methods; guarantee of superior quality on software component's design, implementation and analysis. The main issue behind this is the fact that, due to the increasing complexity of the models and numerical methods, the building of simulation software has become a major part of the scientists' and engineer's work. As a consequence, the tasks of method validation and verification and shifting from one method to another require considerably more time spent in the development and implementation of the software than in just some years ago. The same is also true in the

software industry, and even more dramatically there, due to the needed sophistication required by a very competitive market [8].

Nowadays simulation systems supporting coupled multi-physic phenomena can be important predictive tools in many industrial activities. However, the need for more suitable numerical tools, which could more appropriately simulate a large amount of coupled phenomena, and the need for computational environments, which could help the building of those tools, is still a reality. Simulations using FEM can become very complex, particularly when the designer wants to guarantee high level of abstraction and reuse of the developed solutions. Those requirements comprise the main strategies in saving the production costs of high quality simulation software. This paper shows that a simulation implementation complexity for coupled phenomena can be greatly reduced with the use of predefined structures, due to FEM polymorphism. Of particular importance for us is the simulation of chemo-thermo-mechanical interactions (including control mechanisms), which occur inside a given system and between such a system and its surrounding environment.

The FEM Simulator Skeleton considers four levels of computation for FEM simulator conception. It supports abstractions for different phenomena coupling in a single strategy, identifying which parts can be more reusable than others and proposing a hierarchical and modular solution. The pattern also suggests a physical phenomenon abstraction, called here computational phenomenon. The main objective with such an abstraction is to make it easier the representation of data sharing and dependence between different phenomena.

The pattern's description is based on suggestions found in [16]. In section 2 the pattern name is supplied. Section 3, details the context in which existent problems might inhibit further developments, and to which the pattern solution applies. Section 4 presents the design challenge through a question. Section 5 shows pattern forces, that is, the patterns design trade-offs, what pulls the problem in different directions, towards different solutions. Section 6 explains how to solve the problem. Section 7 describes the pattern applicability. Section 8 presents an example of usage, that is, a general simulator scenery and a possible problem that can be solved by the defined simulator. Section 9 details the resulting context, telling which forces the pattern resolves and which forces remains unresolved by the pattern, and it points to other patterns that might be the next ones to be considered. Section 10 talks about known users. Finally, section 11 presents some conclusions and references.

2. Name

FEM Simulator Skeleton, which means a pattern for modelling FEM simulators based on algorithm skeletons for coupled phenomena.

3. Context

When a designer defines a computational model for a mathematical formalism, using the FEM in the context of coupled phenomena, he has to deal with problems like data dependence and data sharing (Figure 3.1). Such issues are not so trivial to treat in a homogeneous way because it is strongly dependent on the specific problem being considered. Thus it becomes difficult to provide reasonable high levels of abstractions, which could represent the main components, properties, relationships and operations involved. Without that, even when making use of sophisticated FEM libraries, the tasks involved in building and assessing the performance of new methods could become very costly and time consuming due to the lack of modularity and reuse. Also as far as we are concerned, there is no standardized solution for

the control of coupled phenomena simulations, making the integration of reusable components a very difficult task in this context.

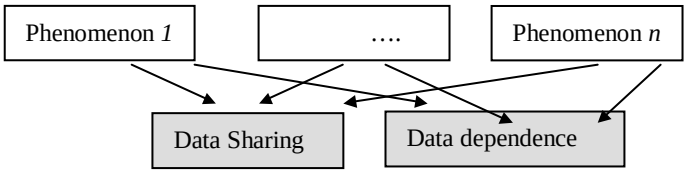


Figure 3.1 Phenomena relationship

There are 4 steps frequently identified in a FEM simulation process: Model, Pre-processing, Simulation and Post-processing [18]. Here we give a brief description of the function considered for each step [8,9]:

1) *Modelling*, where the simulator structure conceptualisation is defined, based on the designer strategies (skeleton algorithms, global scenery and so on);
2) *Pre-processor*, where the simulator is built, that is, where the simulator dynamic data structures are generated and assembled, based on the designer definitions, in order to create the required simulator, for the desired mathematical and computational formalism;
3) *Simulation Processor*, represents the simulation run, where the main computation effort would be, in general, the computation of the discrete vector fields for all phenomena, which means the solution of several coupled systems of (frequently non-linear) algebraic equations for each time step (if time dependent). In this step, different groups of input data are processed on the pre-defined simulator. The output from this processor allows for the model verification.
4) *Post-processor*, where the solution is processed in order to obtain the quantities of interest for the user and for the needed visualization.

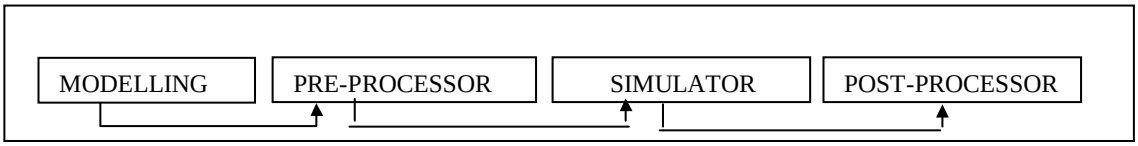


Figure 3.2 Simulation Process

In this pattern we are concerned with the conception of the Simulation Processor, the Modelling phase. We assume that the simulator building and assembling will be based on a variable designer data model, which describes: the initial scenery, algorithm skeletons and numerical methods, phenomena, geometry and so on. The initial scenery defines the class of problems that the simulator will be able to tackle in a broad sense. The simulator model is able of considering the use of many procedures (for instance: Time Loop; Adaptation Iteration; Time Step Estimation; Solution of Algebraic Systems; Error Estimation; etc), which may be either present or not, depending on the configuration of the initial Scenery. Thus the simulator model is provided with a global solution strategy, several phenomena details and their inter-relationships, together with iterations on solution schemes for blocks, groups, and so on. Those pieces of data for the simulator model will then be mapped to appropriate structures during the Pre-processor phase. The result of the Pre-Processor is the final Simulator, which will be activated, when desired, to compute the approximate solutions and to send them to the Post-Processor. The Post-Processor will generate the final results in the way designed and implemented by the user.

4. Problem

How a complex simulator for coupled multi-physics phenomena based on the Finite Element Method (FEM) can be structured in such a way that guarantees high level of reuse and modularity?

5. Forces

The FEM Simulator Skeleton pattern tries to solve forces, which are related to high costs in complex simulation systems development, especially in the direction of complexity management and software quality achievement. Nevertheless, this pattern also considers the automatic articulation of solution strategies for coupled multiphysic phenomena and their possible replacement. In what follows we describe the evolved forces in the context of FEM coupled phenomena simulators modelling:

a) High complexity: there is a lack of standard abstractions that help the simplification and organization of complex structures of data and code related to coupled phenomena simulations in the FEM context. The relationships among phenomena are strongly problem-dependent and solution algorithm dependent.
b) Reusability: numerical experiments are complex constructs, based on pieces of information such as strategies, auxiliary methods and pieces of data. They can be strongly reusable for large classes of problems.
c) Adaptability: due to the frequent improvement of auxiliary numerical methods or due to the need of comparing different methods, the simulator architecture must guarantee that it can suffer adaptations (to some extent) to support the required modifications without heavy reprogramming.
d) Strategy Independence: in order to allow the designer to specify the simulator features and strategies, there must exist flexibility in building different solution strategies.
e) Integrability: there is a need for an integral piece of software, which is able of monolithically solving a specified set of coupled phenomena. Some problems simply do not allow for an independent solution for each phenomenon. Furthermore, whenever different software components have to be used together for the simulation of coupled phenomena (for instance, in a partitioned, staggered way), problems concerning data transfer and integration frequently appear.

6. Solution

The main structure of the pattern for representing a general FEM simulator is composed of Simulator, Block of Groups, Group of Phenomena, Phenomenon and Algorithm Skeletons and MathMethods, see Figure 6.1. The FEM Simulator Skeleton pattern suggests a FEM simulator algorithms organization with 4 levels of computational demands: Global Skeleton, Block Skeletons, Group Skeleton and Phenomenon. These levels were defined due to the high number of repeated (similar) structures and the degree of reusability of the involved algorithms (see example in section 8).

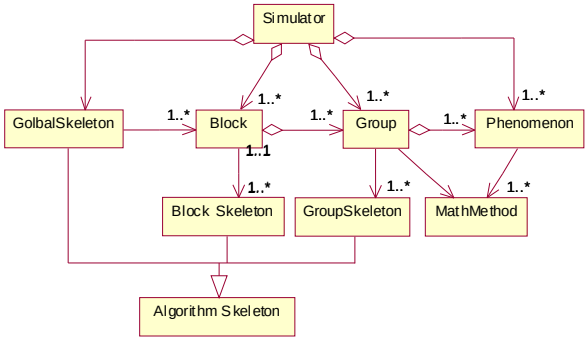


Figure 6.1 Participants of the Simulator Pattern

6.1 Participants

The FEM Simulator Skeleton pattern is composed of the following participants:

- *Simulator* represents a class of possible simulations and it is responsible for the control of the main process flow; thus it maintains the core of simulation through the Global Skeleton.
- *Algorithm Skeletons* are algorithms described by the simulator designer, corresponding to one of the levels of computation (Global, Block, Group), using the pattern-defined abstractions.
- *MathMethod* is a tool with a very specific purpose and is used by either Algorithm Skeletons or encapsulated procedures inside a Phenomenon. For instance, MathMethods are defined for numerical integration, mesh adaptation, error estimation and other tasks.
- *Global Skeleton* is the highest level of the solution scheme and it articulates the action of all Blocks. It is supposed to be strongly reusable.
- *Block* is a set of Groups of phenomena. Each Block has a set of skeletons called Block Skeletons. More than one block is justified, for instance, in the case where a problem can be partitioned into either independent or one-side dependent sets of groups of phenomena.
- *Block Skeletons,* where the Groups are required to perform a certain number of categories of procedures (for instance, partitioned - staggered - solution procedures involving groups of phenomena). When a Group is asked to execute a category of procedures (for instance, to compute a solution for its group of phenomena), it executes a very specific algorithm, which is a member of that category. Block Skeletons are supposed to be strongly reusable.
- A *Group* is a set of phenomena, which are going to be solved monolithically. A Group is provided with a set of Group Skeletons.
- *Group Skeletons* represent very specific procedures. Due to its problem- and method-specific definition and organization, the Group Skeletons are the less reusable among all Skeletons. Nevertheless, it may be implemented in such a way that it becomes able of considering a varying number of phenomena, depending on the requirements from the simulation design.
- A *Phenomenon* represents a complex system composed of data and tools. Its primary responsibility is to provide the contributions of each phenomenon to a Group System to be solved in each instant of the solution process. This level is the place where the **couplings** and other processes of **data sharing** and **dependence** are considered in the formation of

the needed vectors and matrices. It is the lowest level of the procedures in the solution schemes and thus it represents a tremendous effort in terms of programming, testing and validation. Therefore, the reusability of the tools located in the classes, which compose what we call a Phenomenon is fundamental in the saving of time and cost whenever one is programming new simulations.

6.2 Levels of Computation

The 4 levels of computation demands (skeletons and methods) are detailed in what follows.

a) *Global Skeleton* is the first level of computation and represents the global algorithm skeleton (the core of the simulator). The global algorithm skeleton articulates the procedures involving all blocks. The procedures here deals with a higher level of the simulation execution, like time loops, adaptive iterations, and so on. It also includes general requirements such as asking the blocks to obtain the block solution or to perform an adaptation procedure. There is no need for matrices and vectors manipulations in this level. The building of a Global Skeleton depends on a series of decisions about the whole classification of the simulation. A Global Algorithm Skeleton is unique for each simulator, but may be replaceable, producing another simulator. Global Algorithm Skeleton is the procedural structure representing the algorithm to be performed with demands defined still in a higher level. It does not make any requirements directly neither to a Group of phenomena nor to any phenomenon.

b) *Block Skeletons* are made in order to articulate the Groups of Phenomena in the execution of tasks demanded by the Global Skeleton. Each block has a set of skeletons (Block Skeletons), which satisfies the demands from the Global Skeleton by decoding them into demands for the groups in a previously defined order. A simulator may have a Block Skeleton changed without needing to change its Global Skeleton. Nevertheless, a well-designed Block Algorithm Skeleton is also very reusable and it is not supposed to be substituted even in the case of very severe changes in the solution algorithm in the level of the Group of phenomena. Block Skeleton defines solution procedures such as iterations in the case of operator splitting solution strategies (which involves all Groups), iterations in the case non-linear solvers (involving one or more Groups) and so on. It also transfers directly to its Groups some of the demands coming from the Global Skeleton (time step estimation, error estimation, etc.) and possibly post-processing the output from the Groups.

c) *Group Skeletons* are made in order to articulate the Phenomena in the execution of tasks demanded by the Block Skeletons. A Group is provided with a set of Group Skeletons, which represent very specific procedures and may not be very reusable. Its purpose is to segment (encapsulate) the parts from the solution scheme, which are specific of the particular solution method being used for a group of phenomena. Usually, the more reusable parts of the solution scheme are best located either in a Block Skeleton or in the Global Skeleton. In the Group Skeletons the quantities produced by the Phenomena Skeletons are manipulated in the way required by the solution method, which characterizes the Group. Thus the Group becomes specialized in the solution of any subset of a set of possible phenomena and so, all vectors and Matrices used in the solution are located in the Groups. The Group also needs to have knowledge of the couplings of its

Phenomena whenever building coupled terms. This is so because the coupled terms are built using a possibly already computed discrete vector field (possibly related to other group), which should be appropriately defined. Frequently, Group Skeletons make use of MathMethods, whenever there is a task, which can be encapsulated representing either a reusable or a replaceable procedure (solution of an algebraic system of equations, for instance).

d) *Phenomenon Procedures* represent the lowest level of all procedures in the simulation and are specific of all possible contributions its Phenomenon can provide to any solution scheme. Starting from the computation of the Global Skeleton and going through the two other levels of articulation, what remains to be defined are the contributions of each phenomenon to its Group solution scheme in a uniform parameterised way. The phenomena classes will be composed of phenomenon data and a group of numerical methods (MathMethods), which are replaceable (can be modified by the users through input data, like integration rules, for instance).

6.3 Computational Phenomenon Abstraction

A computational phenomenon, or simply a phenomenon (see Figure 6.2), is defined by it's vector field and weak forms defined in its geometric entity together with boundary conditions information. The boundary conditions are also implemented as fictitious phenomena, defined on the respective geometric entity of the boundary of its domain. A phenomenon has also MathMethods which implement, for example, Mesh generation, Integration Rules, Shape Functions, etc.

A computational phenomenon is considered as a set of processes, which produces matrices and vectors related to a specific weak form (as a whole or only parts of it) defined on a specific geometric domain. Each one of those matrices and vectors may be dependent on vector fields from other phenomena. The reference adopted in the definition of a computational phenomenon is based on the simulation region in which it is defined. Phenomenon is an abstraction and may also be used to represent restrictions (involving one or more phenomena), boundary conditions and other types of relationships and processes.

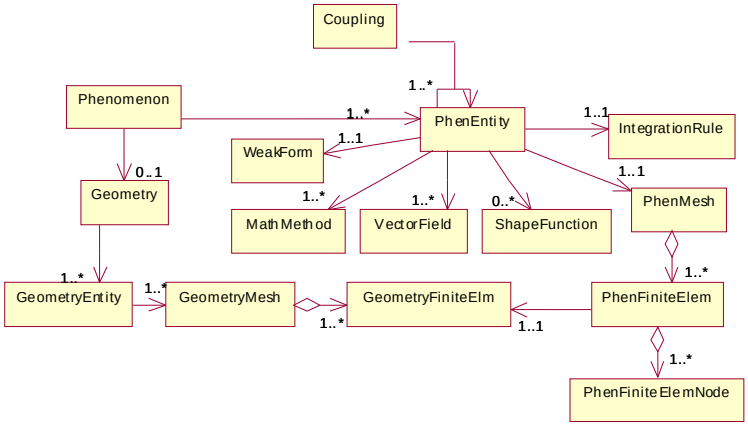


Figure 6.2 Phenomena classes

An original phenomenon may generate several computational phenomena by the time the modelling is finished. The generated phenomena are related to boundary conditions, which

are defined for the original one and implemented as fictitious phenomena. Fictitious phenomena defined in order to implement boundary conditions inherit many pieces of information from the original phenomenon, such as vector field, geometric and phenomenon mesh, etc.

6.4 Interaction

We can summarize the pattern major interaction in the following way: (a) the Global Skeleton articulates the procedures involving all blocks; it does not make any requirements directly neither to a Group of phenomena nor to any Phenomenon; (b) the Block Skeletons then define the activities of the groups; (c) the Group Skeletons in turn articulate the phenomena in their computations. This produces more clean and reusable Global and Blocks algorithm Skeletons leaving to the Groups algorithm Skeletons the responsibility of defining the specific problem dependent (non-reusable) procedures of the whole solution algorithm.

7. Applicability

We can say that the proposed pattern has great applicability in FEM simulation modelling especially when the following situation is very frequent:

- Several phenomena defined on the same geometric region, with either different meshes and different adaptation criteria or sharing meshes and other data;
- Interchange of data between phenomena is very frequent (data dependence)
- Assessment of solution quality may be different and sometimes interdependent (error estimation, adaptation, approximation properties) from one phenomenon to another
- The desired solution algorithms articulate separate groups of phenomena and those groups, in turn, consider sets of phenomena in the computation procedures (as it is the case in operator splitting (staggered) schemes).

8. Example of Usage

In order to make clear why the pattern was proposed, in section 8.1 we show the general scenery of a simulator model, which is described to the extent that it is needed for our explanation purposes. Many details will not be mentioned in the pursuing of explanation clarity. Section 8.2 details the pattern application to the proposed simulator scenery. The objective of this example is to give a better comprehension of the involved pattern participants, providing an illustration of the interaction between the computation levels of FEM Simulator Skeleton pattern. Section 8.3 provides some considerations related to the pattern application. In section 8.4 we show one real example, of problem formulation, that can be solved by the defined simulator.

8.1 A Simulator description

We consider the following global scenery example for a FEM simulator specification: a simulator capable of solving problems involving transient Phenomena; the phenomena context includes linear temperature-dependent elasticity, rigid body motion and linear heat transfer; Dirichlet restrictions are considered through Lagrange multipliers; the simulator process does not include estimation error and adaptation processes (see Figure 8.1).

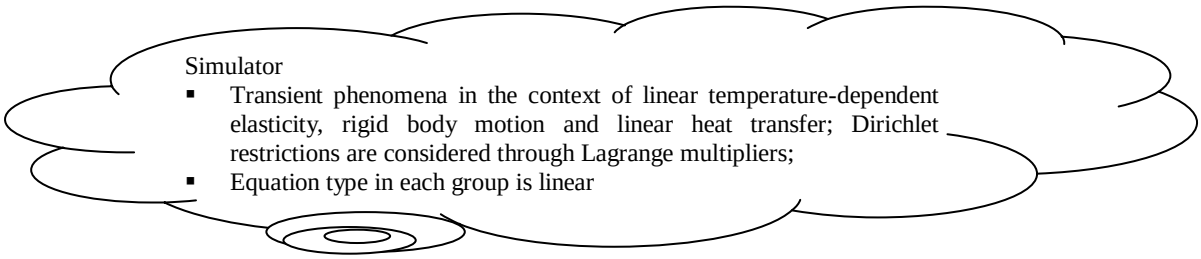


Figure 8.1 Simulator specification (global scenery)

The Block Scenery for each Block considers an iteration between the solution for the Lagrange multipliers and the solution for the phenomena themselves (assumes stabilization). Also, it assumes that there are two blocks: one for temperature and its Lagrange multipliers and other for the elasticity, rigid body motion and their Lagrange multipliers. This type of choice for the number of blocks is due to the fact that the present model of heat transfer does not depend on the result of the elasticity problem. Thus the heat transfer problem (and respective Lagrange multipliers) can be solved before solving the elasticity/rigid body motion problem (and respective Lagrange multipliers), which depends on the temperature.

The Group Scenery for the Groups of Phenomena is with matrix assembling; the inner procedure for each solver group is linear and iterative with Pre-conditioning and Equation type in each group is linear. No Front tracking.

8.2 Pattern Application

Usually, it is observed that an algorithm defined for the solution of a problem by the FEM method has repeated (similar) structures. Thus in the pursuing of a high degree of reusability, four levels of demands in the algorithm were devised: Global Skeleton, Block Skeleton, Group Skeleton, and Phenomena procedures. In the Block Skeleton we will assume that N^i_g is the number of groups for the i^{th} -block. In what follows we present the algorithm Skeleton and the Global skeleton.

Figure 8.2 describes the Global Algorithm Skeleton for the proposed Simulator. As it involves, for example, transient phenomena it includes tasks to compute initial time steps for blocks and also the computation of the next time step.

*I. From Blocks i = **1** until **2***
*I.0) Retrieve initial state for Block **i***
*I.I) Compute initial time step Δt_i for Block **i***
*I.II) Compute initial auxiliary data for Block **i***
II. Compute initial $\Delta t = min_{1 \le i \le 2} \{\Delta t_i\}$ and set time instant $t_1 = 0$
III. While $t_1 \le T_{max}$ do:
III.0) Set $t_0 = t_1$ and $t_1 = t_0 + \Delta t$
*III.I) For Block **i = 1** until **2***
*III.I.0) Solve for Block **i***
*III.I.I) Compute next time Δt_i for Block **i***
III.II) Compute next time step $\Delta t = min_{1 \le i \le 2} \{\Delta t_i\}$
III.III) Continue with time iteration
IV. End of the simulation

Figure 8.2 Global Algorithm Skeleton

In Figure 8.3, we represent a graph in order to help the understanding of the algorithms that compose the global skeleton. In the graph-like structure, each node on the graph means

either the definition of a loop, an iteration or a procedure. Figure 8.4 details the Block Skeleton for the proposed Simulator. It is composed of sub-skeletons that implement for example: Initial state for the Block (I.0), the solver for the block (III.0), and so on.

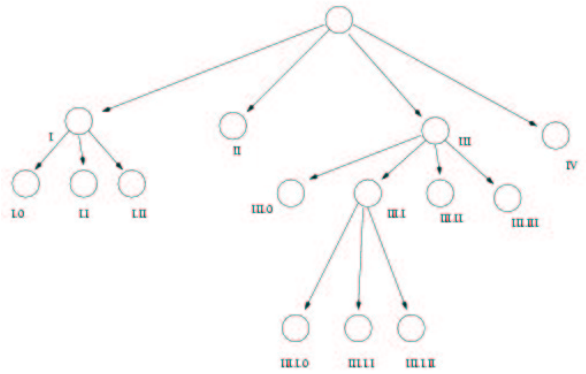


Figure 8.3 Global Algorithm Skeleton Graph

Is-B^r) Initial State for Block r (see (I.0)):
*Is-B^r.0) For **i = 1** until N^r_g*
Is-B^r.0.0) Ask Group i to compute Initial state for its phenomena
It-B^r) Initial time step for Block r (see (I.I)):
*It- B^r.0) For **i = 1** until N^r_g*
It- B^r.0.0) Ask Group i to compute Initial time step Δi
It-B^1.I) Set $\Delta t^1 = \min_{1 \le i \le N^1_p} \{\Delta_i\}$
Id-B^r) Compute initial auxiliary data for Block r (see (I.II))
Id-B^r.0) For i = 1 until N^r_g
Id-B^r.0.0) Ask Group i to compute its auxiliary data.
*Sl-B^r) Solve for Block **r** (see (III.0))*
*Sl-B^r.0) Initialise iteration state **k = 0** for Block **i***
*Sl-B^r.I) Set **k = 0**. While convergence for Block **r** is not achieved, do:*
Sl-Br.I.0) Compute the (k+1)th-solution based on the k th-solution for Block r
Sl-Br.I.I) Compute error between the solutions k and k+1 for Block r
Sl-Br.I.II) Compute auxiliary data for next step and increment k = k+1
*Sl-B^r.II) Accept last solution from iteration loop for Block **r***
Sk-B^r) Initialise iteration state k = 0 for Block r (see (Sl-B^r.0)):
Sk-B^r.0) For i = 1 until N^r_g
Sk-B^r.0.0) Ask Group i to initialise iteration state k= 0
Sl-B^r) Compute the $(k+1)^{th}$-solution from the k^{th} -solution for Block r (see (Sl-B^r.I.0)):
Sl-B^r.0) For i = 1 until N^r_g
Sl-B^r.0.0) Ask the Group i to compute the $(k+1)^{th}$ -solution from the k^{th} –solution for Group i
Er-B^r) Compute error between the $(k+1)^{th}$ -solution and the k^{th} -solution for Block r (see (Sl-B^r.I.I)):
Er-B^r.0) For i = 1 until N^r_g
Er-B^r.0.0) Ask Group i to compute error Ei^k for Group
Er-B^r.I) Compute Block error $E^{r,k}$ based on the Group errors $\{ Ei^k\}$ $1 \le j \le N^1_g$
Ad-B^r) Compute auxiliary data for Block r at k^{th} -iteration (see (Sl-B^r.I.II)):
Ad-B^r.0) For i = 1 until N^r_g
Ad-B^r.0.0) Ask Group i to compute the auxiliary data for this Group.
As-B^r) Accept last solution obtained in the iteration for Block r (see (Sl-B^r.II)):
As-B^r.0) For i = 1 until N^r_g
As-B^r.0.0) Accept last solution obtained for Group i and store it.
Nt-B^r) Compute next time step for Block r (see (III.I.I)):
Nt-B^r.0) Ask group i to compute next time step Δ_i
Nt-B^r.I) Set $\Delta t^r = \min_{1 \le i \le N^r_g} \{\Delta_i\}$

Figure 8.4 Block Skeleton for any Block r

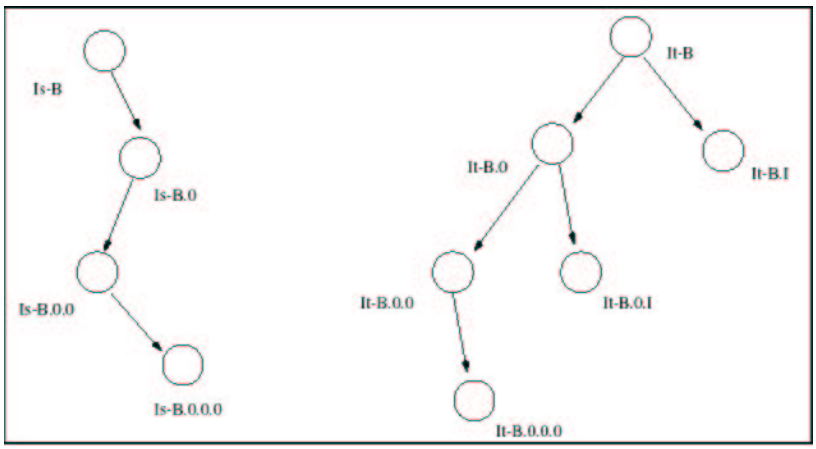


Figure 8.5 Example of subgraphs of the Block Skeleton Graph

In Figure 8.5, a part of the Block Skeletons is described in graph-like structures. So, the number of Block algorithms is ten, which are the same for both blocks in the current setting. Observe that those Block Skeletons articulate the groups in a very simple way, almost only sending to the groups the requests made by the Global Skeleton. Nevertheless, it should be noted that the decision of providing an iterative scheme involving the Groups was made and defined by the Block Skeleton. In this sense such a procedure is transparent to the Global skeleton and to the Group Skeletons.

The Group Skeletons is subtler in what concerns the articulation of their phenomena for providing the demands of the original solution algorithm. The detailed description of Group Skeletons is beyond the objectives of this example. However, it can be seen from the algorithm that each Phenomenon should provide Matrices and vectors for assembling. In order to make it simpler and uniform, consider that, for each matrix (let us say, **C**) a given phenomenon may provide at the finite element level, it will offer the following indexed options

0. **C**
1. **C . Vec**

If more than one matrix is provided (let us say $\mathbf{C}_1$ and $\mathbf{C}_2$) linear combinations will also be provided like, for instance,

2. **a** $\mathbf{C}_1$ + **b** $\mathbf{C}_2$

In the above **Vec** is a given vector and **a** and **b** are given scalars. Each one of those quantities a phenomenon offers to its Group may depend on vector fields from other phenomena (either from its Group or not). It is the responsibility of the current Group to indicate: (i) the quantity to be computed by its phenomena and (ii) in the case of coupling, what is the vector field (either from the current Group or not) the coupled phenomena should use in order to provide what is needed for the computation of the coupled quantity. With such an organization, demands to the Phenomena become very uniform, making them extremely reusable.

A final remark is related to the fact that this pattern was made possible by the way the data and tools are built in the Phenomenon level. It is in this level that the data dependence and sharing between phenomena are defined, leaving the Global Skeleton and the Block Skeletons free from those details. However, it is the Group Skeletons, which are the agents responsible to map the need of a phenomenon for quantities from other phenomena to the actual quantities, which are stored either in the current Group or in other Groups.

8.3 Considerations

This pattern considers that the class of problems, which define the applicability of a simulator, can be defined in a somehow clear way. For instance, considering only its Global Skeleton, the Simulator built in the example (section 8.2) is able of solving simulations in the class of dynamic problems with neither adaptation nor error estimation. Now, considering its Block Skeletons, it is able of solving only linear (or very mild non-linear) problems with Dirichlet type of restrictions and using a staggered stabilized methodology. Those restrictions may involve one or more vector fields. The Group Skeletons are very specific to the solution scheme used and even slight modifications may cause a need to redesign and re-program them. As it was said just before, the **couplings** and other process of **data sharing** and **dependence** are considered in the phenomenon level leaving the Global Skeleton and the Block Skeletons free of having to consider them. Since the Group Skeletons are the least reusable, it may (and frequently does) deal with specifying the right quantities a coupled phenomenon should retrieve from its own Group. This reflects a coupling between Groups, which has been described earlier and is related to the specifics of the solution methodology being used by the Group.

8.4 Example of the Simulator Applicability

An example of a problem that can be solved by the defined simulator (section 8.1 and 8.2) is described in what follows. Consider the geometry defined in Figure 8.6. It is composed of two sub-domains Ω_1 and Ω_2. The physical phenomena defined therein are (transient state): linear elasticity with temperature dependent constitutive equations in Ω_1; rigid body motion of Ω_2 (this body has a certain distributed density ρ_M) and heat transfer in Ω_1 and Ω_2. Let a point $\rho_M \in \Omega_2$ be a reference point for the rigid body. It is very convenient if such a point could be the centre of mass of Ω_2, because the weight and inertia terms will not generate moments around ρ_M. The proposed simulator will build the global linear system related to all the geometric mesh elements, for each phenomenon, and solve this system.

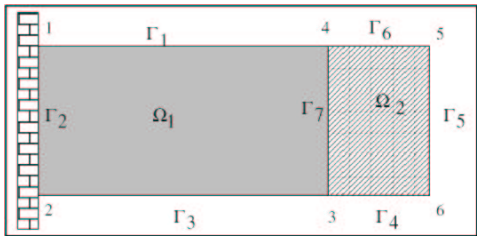


Figure 8.6 Whole domain

For the present example of problem formulation, to be applied to the defined FEM simulator we can consider the following defined [10]:

a) Blocks: block 1, composed of groups 1 and 2; block 2, composed of groups 3 and 4.
b) Groups: group 1, phenomena represented by vector fields T_1 and T_2 (heat transfer in Ω_1 and Ω_2); group 2, phenomena represented by vector field μ_q (Lagrange multiplier in Γ_7, due to restrictions between T_1 and T_2); group 3, phenomena represented by their vector fields w_1 and w_2 (elasticity in Ω_1 and rigid body motion in Ω_2); group 4, composed of the phenomena represented by their vector fields μ and μ_f (Lagrange multipliers in Γ_2 and Γ_7, respectively, due to restrictions in w_1).

9. Consequences

It is worthwhile observing that a FEM Simulator Skeleton pattern is not restricted to a given implementation of Blocks, Groups and Phenomena. Their abstract behaviour and interaction are independent of a specific implementation. When dealing with the building of a specific Simulator, the implementation of the Global Skeleton and of the Blocks Skeletons should reflect the needs for the solution of a large class of problems, which constitutes its strategy. Thus each built Simulator, based on the proposed pattern, should be able of solving completely different problems, defined on completely different geometries and considering completely different sets of phenomena, provided the problem is still within its applicability range.

9.1 Forces solved by the pattern

The FEM Simulator Skeleton pattern supports:

- Higher level of abstraction for the main concepts of FEM Simulation Skeleton pattern modelling, reducing the complexity and improving the correctness of the systems (simulators) that will be developed.
- Higher level of hierarchical modularity for the system process organization, by the use of global skeletons, blocks and group skeletons.
- A solution, which may consider monolithic coupled phenomena simulation.
- The higher levels of code reusability are found in the Phenomena, Global and Block skeleton structures, followed by Group of phenomena. The less reusable is the group of phenomena, because it is the place more sensitive to modifications, whenever changes in the numerical method and type of simulation are desired.
- Reliability of the computer-generated predictions is improved by the use of pre-defined strategies, numerical methods and templates.
- A higher level of maintainability is acquired once the pattern: separates different levels of computation and high reusability of the first two levels of computation is guaranteed.

9.2 Negative Consequences

In the FEM Simulator Skeleton pattern some negative consequences can be identified: the model builders require special training, that is, the designer must understand the proposed abstractions; designers will only achieve higher levels of reusability if they know how to articulate their strategies and problems; the simulator performance can decrease due to the extra imposed levels of abstraction.

9.3 Forces unsolved by the pattern

Some forces are still not solved or not even treated in the present work:

a) Automatic programming: this is desired due to the great volume of code that must be reprogrammed in a single application of coupled phenomena.
b) Expertise level: there are lots of standard situations and states, which are neither assisted nor guaranteed.
c) Performance: generally the simulations are very computer time consuming. So the performance must be taken seriously into consideration.

d) Scalability: simulations frequently require large volume of data, which can be partitioned and processed by many processors in a distributed memory environment. So, scalability with respect to the number of processors is important.
e) Portability: the simulations code should have high levels of reusability. So, it is important for it to be portable in order to take advantage of different existing expertise, using different computational environments, which frequently should interact in building multi-physics simulations.
f) Reliability: the reliability of computer-generated predictions is a great concern to specialists.
g) Simulation Pre-processing: pre-processing of input data is an important task, since the simulator structures require a complex mapping of the real input data. Also, the data structure may ease the burden on the global algorithms complexity in what concerns data sharing and data dependence between different phenomena.

9.4 Patterns that might be the next ones

Patterns that might be the next ones to be considered are related to:

- Pre-processing of input data (it was considered in [11]);
- Support for solution independence, in order to allow the designer to specify the system features and strategies;
- Automatic Programming.

9.5 Related patterns

The authors did not find any pattern that is specifically about algorithm hierarchical modularisation for simulations based on FEM. There are some works, however, which present some level of abstraction and modularisation [3,4,7]. Specifically, in the simulation of coupled phenomena based on FEM, there are some works under development [8,9,10], but not yet in a pattern form.

10. Known users

Computational Mechanics has had a profound impact on science and technology over the past three decades, due to its effectiveness in solving problems of interest to society and on providing deeper understanding of natural phenomena. The field has been enormously successful to date because of its unprecedented predictive powers, making it possible the simulation of complex physical events and the use of these simulations in the design of engineering systems [10]. This is done through the so called “computer modelling”: the development of discretized versions of theories of physics (and other fields), which are amenable to digital computation, together with complex processes of manipulating these digital representations to produce abstractions of the way real systems behave [1]. The Finite Element Method (FEM) has been frequently used in the field of Computational Mechanics, which has come to rely heavily on this technique. Gradually FEM is becoming the most popular analysing procedure within various fields of design [2].

Thus due to tremendous ongoing activity in the fields of application of the FEM, there is a need for tools which could help the development of simulators with a high reusability degree in both the academic and industrial worlds. The expected users of this pattern are

scientists and engineers who already deal with development of FEM codes in some level, or, at least, have a basic knowledge of that method.

11. Conclusion

The FEM Simulator Skeleton pattern gives support in the conception of FEM simulators. This pattern makes it possible to separate complex procedures from simpler ones and strongly re-usable software components from less re-usable ones. Besides, it opens the way to automatic programming of FEM simulators for coupled phenomena. One immediate benefit is the enhancement of re-usability.

References

[1] Committee on Theoretical and Applied Mechanics, "Research Directions in Computation al Mechanics", a report of the United States, September 2000.

[2] Tworzydlo, Oden T. J. "Knowledge- Based Methods and Smart Algorithms in Computational Mechanics", Engineering Facture Mechanics, Vol.60 No5/6, 1995.

[3] Langtangen et al, "Increasing the Efficiency and Reliability of Software Development for System PDEs", Modern Software Tools for Scientific Computing, pages 247-268 Birkauser, 1997 .

[4] Langtangen et al, "Finite Element Pre-processors in Diffpack", Report 1999-01.

[5] OMG, "Unified Modelling Language Specification Version 1.4", September 2001.

[6] Oren T., King D., "Requirements for a Repository Based Simulation Environment", Proceedings of the 1992 Winter Simulation Conference. ed. Swain, Goldsman, et, 1992.

[7] Parker S., Weinstein D. and Christopher J. "The SCIRUN Computational Steering Software System", Modern Software Tools for Scientific Computing, 1997

[8] Santos F., Lencastre M. and Almeida I. "Data and Process Management in a FEM Simulation Environment for Coupled Multi-physics Phenomena", Fifth International Symposium on Computer Methods in Biomechanics and Biomedical Engineering Rome, 2001

[9] Santos F., Lencastre M. et al. "FEM Simulation Environment for Coupled Multi-Physics Phenomena", Simulation and Planning In High Autonomy Systems - AIS'2002; Theme: Towards Component-Based Modelling and Simulation, Lisboa, Portugal, 2002.

[10] Santos F., Lencastre M. and Araújo J. "A Process Model for FEM Simulation Support Development", Summer Computer Simulation Conference - SCSC'2002, US Grant Hotel- San Diego, California, July, 2002

[11] Tanir and Servic "A Standard Simulation Environment: A Review of Preliminary Requirements", Proceed. Winter Simulation Conf. ed. Swain, Goldsman et al, 1994.

[12] Rucki and Miller "An Algorithm Framework for Flexible Finite Element-based Modelling", Computer Methods Application Mechanic Engineering, 136 (1996) 363-384.

[13] Tworzydlo and Oden "Knowledge- Based Methods and Smart Algorithms in Computational Mechanics. " Engineering Facture Mechanics. Vol.60 No5/6, 1995.

[14] Kortright "Modeling and Simulation with UML and Java", Nicholls State Univ. LA, 1997.

[15] Gamma, E. et al. "Design Patterns – Elements of reusable object-oriented software". Addison-Wesley, 1994.

[16] Coplien, J. "Foundation of Pattern Concepts and Pattern Writing". Bell Labs / USA and University of Manchester / UK. Simpósio Brasileiro de Engenharia de Software –SBES'2001, Brazil, 2001.

[17] Banks, J. "Handbook of Simulation - Principles, Methodology, Applications and Practice". John Wiley & Sons, Inc. Georgia Institute of Technology, Atlanta, Georgia. 1998.

[18] Cook R. D. " Finite Element Modeling for Stress Analysis", by John Wiley and Sons, Inc., 1995

APPENDIX

This appendix presents the following paper "PDC: Persistent Data Collections Pattern". PDC is a design pattern that contains a set of classes to obtain a better maintenance and reuse levels when using persistence mechanisms to develop an object-oriented application. This paper was submitted and workshopped in the first version of the SugarloafPLoP conference (SugarloafPLoP'2001). Unfortunately, we could include it in the SugarloafPLoP'2001 hard copy proceedings, but we are glad to publish its final version in the SugarloafPLoP'2002 proceedings.

PDC: Persistent Data Collections pattern

Tiago Massoni Vander Alves Sérgio Soares Paulo Borba*
Centro de Informática
Universidade Federal de Pernambuco

Introduction

The object–oriented applications layer architecture [2, 3] allows the distribution of classes into well–defined layers, according to each crosscutting concern of an application (business, communication, data access, etc.) to obtain separation of concerns. Elements from different layers communicate only through interfaces. However, we have to refine these layers by filling them with specific classes. The complete set of these classes, related to business and data access concerns, was transformed into a design pattern, called PDC (Persistent Data Collections), which is presented in this paper.

Brief

Provide a set of classes and interfaces in order to separate data access code from business and user–interface code, promoting modularity.

Context

When developing persistent object–oriented information systems applications using specific Application Programming Interfaces (APIs) that lead to interwoven code making maintenance and reuse difficult.

Problem

Obtain better maintenance and reuse levels when using persistence mechanisms to develop an object–oriented application.

Forces

- Developers should be able to address the business aspects of an application independently from persistence operations.

*Supported in part by CNPq, grant 521994/96–9. Electronic mail: {tmasson,vralves}@us.ibm.com, {scbs,phmb}@cin.ufpe.br. Av. Professor Luis Freire s/n Cidade Universitária 50740–540 Recife PE Brazil.

- *Ad hoc* implementations directly using specific Application Programming Interfaces (APIs) usually lead to interwoven code that is hard to maintain. For example, a Java [8] program can use the JDBC API (Java Database Connectivity API [14]) for manipulating persistent data within business code.
- The type of persistent storage or vendor may change over the life of an application.
- Business classes may be reused by other applications.
- It may be non-trivial to deal with some aspects from persistent systems, such as enabling connections to database platforms and managing transactions efficiently.
- The system performance should not be affected.

Solution

The basic idea of PDC is to avoid mixing data access code with business code from domain-related objects, leading to extensibility and reusability. For this purpose, we propose the separation of design classes in two types:

- classes describing business logic objects.
- classes for data manipulation and storage, with specific persistence code.

The communication between these two types of classes is carried out through interfaces, which guarantee independence between the business layer and the data access layer. Business code will be the same, regardless of how data access operations are implemented.

PDC suggests the use of persistent data collections, which contain code for manipulating a group of persistent objects of an application. These collections represent a clear distinction between the "data" and the "data set", being the core of our solution. Our solution is complemented by ideas taken from other well-defined design patterns, as Facade, Abstract Factory and Bridge [7]. The goal is to reduce the impact caused by modifications in the system functional and non–functional requirements.

As in the example in Figure 3, for each important domain object which will be persistent in the application (like `Account`), we create two other classes: the business collection (`AccountRecord`) and the data collection (`AccountRepositoryJDBC`) classes, representing business and persistent collections of domain objects, respectively. Furthermore, each persistent domain class must inherit from the `PersistentObject` class, indicating that its objects will be stored persistently.

The `Bank` class encapsulates all services offered by the application (applying the Facade pattern [7]). The object from this class calls methods on all business collection objects of the application (as `AccountRecord`), in order to implement the services. The business collection in turn uses persistence-related services from its corresponding persistent data collection (as the `insert` and `search` methods).

The `PersistenceMechanismJDBC` class is used by `Bank` and persistent data collections (as `AccountRepositoryJDBC`) for performing database platform services, such as connection and transaction management. These issues are addressed by specific methods in the persistence mechanism.

In order to request services from the data access layer, the business objects send messages to data access objects only through interfaces, which provides extensibility for the design of the application. In the example, the `IAccountRepository` interface separates business collections from persistent data collections, and the `IPersistenceMechanism` interface isolates specific persistence mechanism services from its business clients, such as the `Bank` class.

As in the example above, we can use PDC to structure the application using a set of specific classes, separating business and user–interface concerns from persistence concerns. Such application is easier to maintain and to extend, since its core functionality is decoupled from data access code. In addition, classes from the application can also be reused by other applications.

Structure

Figure 1 details the structure of PDC, using an UML class diagram [4]. The class names denote the element of the pattern itself, including classes with the "Interface" stereotype, which denote interfaces containing only method signatures to be implemented by the indicated classes.

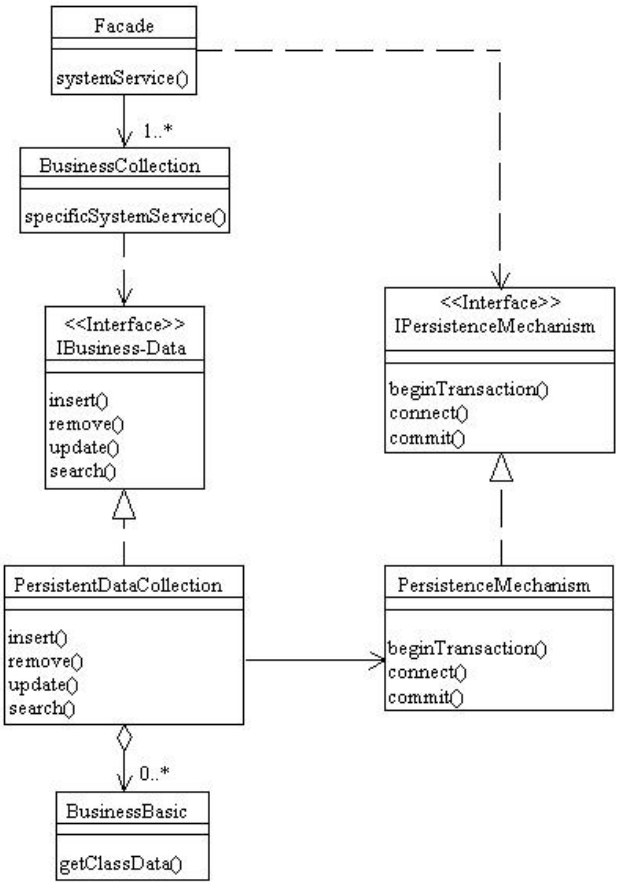


Figure 1: Class diagram of PDC.

The participants of the pattern are presented as follows, along with their matching elements of the example presented in Figure 3:

- **Facade.** This class provides a simple interface to all services of a complex system [7]. A facade offers a simple default view of the system that is useful for most clients. It keeps references to the several `BusinessCollection` objects of the application, and delegates calls to them. Additionally, it implements the Singleton pattern, thus exactly one instance of this class will be active during execution. This element is represented by the `Bank` class in the example.

- **BusinessBasic.** This class represents a business basic concept, reflecting clearly the problem domain (for instance, account, client, investment). If we choose this class to inherit from an abstract class containing abstract data access methods (see Implementation Section), the `BusinessBasic` class has to implement those methods. Using this approach, although some data access code is placed within a business class, the business code of the class does not depend on the data access code. Such code on a business basic class can be easily removed or replaced, with no impact on business code. In the example, this class is represented by the `Account` class.

- **BusinessCollection.** This class represents a grouping of objects from a significant business basic class, on the business' perspective. It contains methods for inserting, querying, updating, and deleting business objects, with verification and tests of preconditions related to the object manipulation. Furthermore, the `BusinessCollection` class also contains methods directly related to the application domain. This element is represented by the `AccountRecord` class in the example.

- **PersistentDataCollection.** This class contains methods for manipulating persistent objects of a specific business basic class. The code for these methods depends on a specific API for accessing some persistence platform, thus any changes to this platform will cause direct impact on this class, but absolutely no impact on business code (since the `IBusiness–Data` interface isolates these changes). The `PersistentDataCollection` class implements methods from a `IBusiness–Data` interface and depends on services from the `PersistenceMechanism` class in order to perform database operations, more specifically for finer granular transactions and database connections. In the example, the role of this class is played by the `AccountRepositoryJDBC` class.

- **IBusiness–Data.** This interface establishes a communication protocol between `BusinessCollection` objects and `PersistentDataCollection` objects. A business collection class depends on this interface for storing and retrieving objects from the database. This approach promotes modularity, since changes to the data access code do not have impact on business code. In the example, this interface is represented by `IAccountRepository`.

- **PersistenceMechanism.** This class contains methods that implement specific services related to a database platform, such as connecting to and disconnecting from the database, and transaction management. Methods related to connection management open and maintain a database connection for a service from the application, making this connection available to one or more `PersistentDataCollection` objects involved in the accomplishment of the service. Methods related to transaction management open, confirm or abort transactions, in order to provide consistency among all operations used to accomplish an application service. The code of these

methods depends on a specific persistence API. This class is represented by the `PersistenceMechanismJDBC` class in the example.

- **IPersistenceMechanism.** This interface is defined in order to provide independence between the business classes and the `PersistenceMechanism` class (which implements this interface). Therefore, if we change the database platform, we have to replace the old `PersistenceMechanism` object by a new object, but this modification does not have impact on business classes. The `Facade` class depends on this interface for invoking transaction methods. The example presents an interface with the same name.

Dynamics

Figure 2 shows a sequence diagram [4] of a typical scenario for the use of PDC, using the approach of data access methods encapsulated into a business basic class (see Implementation Section). The `Facade` object creates a `PersistenceMechanism` object, whose services will be requested during execution. Next, a service on the `Facade` object is called, which in turn begins a transaction (invoking a method on the `PersistenceMechanism` object) and delegates the call to a `BusinessCollection` object in order to perform this service (a querying operation that retrieves data from the database). The `BusinessCollection` object performs all validation and tests on the input data, then invokes an operation to manipulate persistent data on the corresponding `PersistentDataCollection` object (through the corresponding business–data interface). The latter creates an empty `BusinessBasic` instance and fills it with database information (calling `deepAccess`, which in turn executes queries through services offered by the `PersistenceMechanism` object, as the `executeQuery` method), returning the resulting object to the `Facade` object. In the end of the operation, the `Facade` object confirms the end of a database transaction, invoking `commitTransaction` on the `PersistenceMechanism` object.

Consequences

The use of PDC offers the following benefits:

- *Support for independent implementation.* PDC's layer architecture allows to address the business aspects independently from persistence operations. This abstraction is promoted by interfaces between the business layer and the data access layer.

- *Maintainability.* The pattern's structure increases the system maintainability by separating business code from data access code. Therefore, changes in the data access classes should not interfere in the business classes.

- *Extensibility.* The pattern makes it easier to seamlessly change the database technology or vendor, minimizing or even eliminating impact on business code. Interfaces between the business layer and the data access layer promote the desired extensibility for the application.

- *Use of several persistence platforms.* The resulting code is able to support storing objects into several persistence platforms, such as files, relational and object–oriented databases, by creating a number of implementations for the persistence

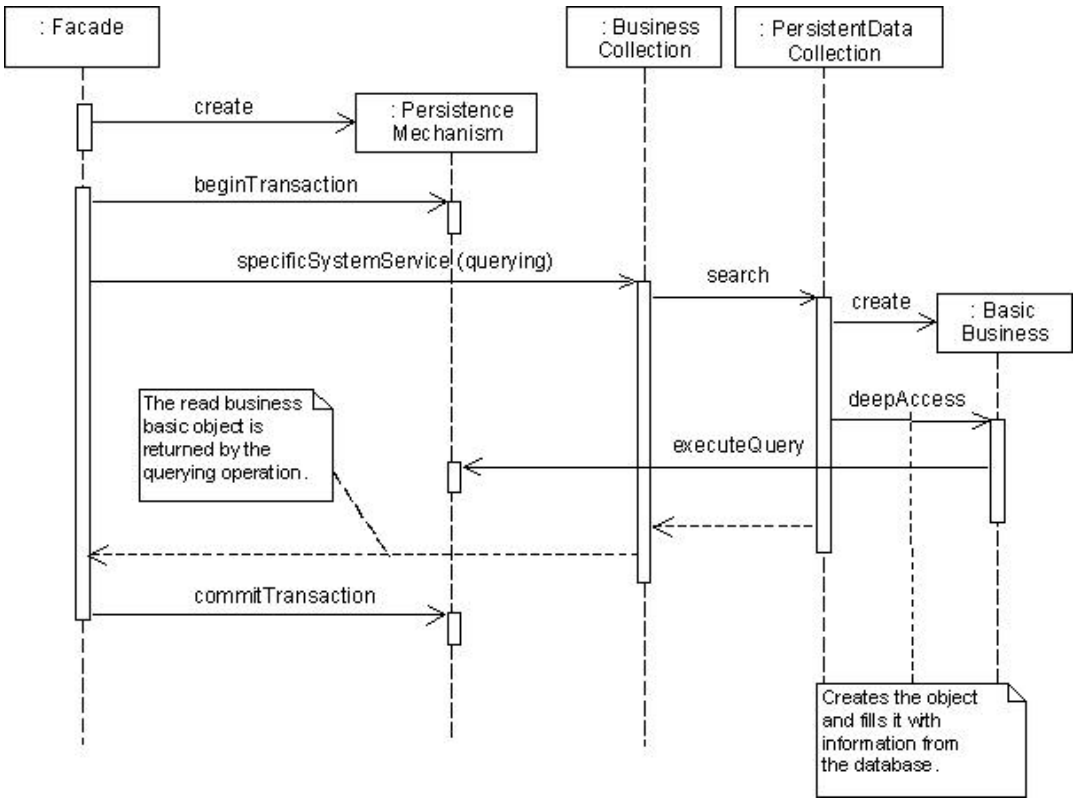


Figure 2: Dynamics of PDC.

mechanism class and for each persistent data collection class; all of these classes must implement the corresponding interfaces.

- *Reuse.* Due to the structure provided by the pattern, business classes can be easily reused by another application based on other database technologies. In addition, changes to data access issues are simpler, since they are restricted to data access code.

- *Abstraction.* As the pattern abstracts the persistence problem by using interfaces, persistence implementation may use complex algorithms or APIs to deal with some non-trivial aspects from persistent systems, such as enabling connections to database platforms and managing transactions efficiently.

- *Support for progressive implementation.* During early phases of the application development, functionally complete prototypes are constructed, where business collection classes depend on business–data interfaces, but the latter are implemented by volatile data collections (storing objects in memory only). Later, data access code can be added seamlessly, replacing volatile data collections by specific persistent data collection objects, then adding a persistence mechanism object. Such approach enables addressing the business problems independently from persistence operations, simpler validation of user requirements, and simplification of tests [9].

The liabilities of the pattern are:

- *Increased number of classes.* For each significant business basic class, we have to create up to three additional classes and one interface. However, their structure is simple and their generation can be simply automated by tools.

- *Increased indirection.* In order to introduce the layer architecture we must use different kinds of classes that delegate some calls to others, which may decrease system performance. In fact, this lost of efficiency is minimal, since these indirections are locally executed, and the additional execution time is irrelevant when compared to the overhead of the IO operations that read from and write to the persistence mechanism.

Implementation

Here we consider how to implement PDC using JDBC as the data access API for using relational database services. Consider the following implementation issues:

- *Java platform.* The pattern elements must be implemented in the Java programming language, since JDBC is part of the Java platform.

- *Inheritance in the business basic class.* Most code for manipulating objects using JDBC can be contained in business basic classes, within methods inherited from an abstract class (`PersistentObject` in our banking example). It can be considered a miscellaneous of business and data access code, even though those inherited data access methods are not invoked by business code (as mentioned earlier). One alternative for such situation is to transfer all code for manipulating persistent business basic objects to the persistent data collection classes. The disadvantage of such approach is that changes in a business basic class will also reflect in the corresponding persistent data collection class; it is necessary to implement a new persistent data collection for each new platform. On the other hand, in this approach changes in the persistent platform will not affect the business basic classes.

- *Transactions.* Using JDBC, we can easily implement transactions using database services. We must use the `setAutoCommit`, `commit` and `rollback` methods on the `Connection` class in order to implement a transaction when implementing a sequence of operations, which must be executed as a single one.

- *Business basic subclasses.* A business basic class can be specialized in business basic subclasses, depending on the business rules. In the case of business collection and persistent data collection classes (including business–data interfaces), we can choose from two design alternatives: one is to create a class for each business basic subclass; another is to use only one class, in order to avoid duplicate code. A detailed discussion about this topic is presented in a related work [15].

- *Concurrency control.* One concurrency problem arises when using a connection pool to manage the connections with the persistence mechanism. Each execution flow (thread) must obtain a connection from the connection pool before communicating

with the persistence mechanism. Usually there is a single connection pool containing all the connections of the system, and thus this poll is accessed concurrently. Moreover, we need to apply some concurrency control to the system. Examples of others situations in which concurrency control should be addressed are interference by business rules (system policies), unsafe data types, and other race conditions [12].

- *Volatile data collections.* We can use this type of class for storing objects in a non–persistent manner, in order to support progressive implementation. Using this approach, we can abstract from persistence or any other non–functional requirement, when implementing functional prototypes for the application. These prototypes can be useful for validating user requirements and simplifying tests. This class also implements its corresponding business–data interface, but its methods use in–memory data structures like arrays or lists to manipulate business objects.

- *Abstract factories.* Variations of PDC can include classes which represent abstract factories [7], in order to increase extensibility and reusability of business classes. An abstract persistence factory class can be introduced, containing a method for creating a persistence mechanism object, and such method can be implemented by a subclass of the abstract factory, the concrete factory. The facade object can call this method to instantiate the persistence mechanism, without making a explicit call to its constructor method. The same idea can be used for creating persistent data collections, isolating the business classes (facade and business collection classes) from the instantiation code. In both cases, the information needed by the concrete factories to instantiate the objects is placed in simple text or XML configuration files.

Sample Code

We now provide a brief sketch of the implementation of the main elements of PDC using Java and the JDBC API, in the banking application example introduced in Figure 3. First, we present a business basic class, `Account`, which reflects directly the problem domain. The `public` modifier in classes and methods is omitted by brevity.

```
class Account extends PersistentObject {
    private Number number;
    private double balance;
    void credit(double value) {  balance = balance + value;  }
    ...
    /* Data access operations */
    void insert() throws StoringException {
        try {
            String sql = "insert into account values (";
            sql += "ID = "+super.getId();  // get the object id
            sql += "NUMBER = "+this.getNumber();
            sql += "BALANCE = "+this.getBalance();
            super.pm.executeUpdate(sql);
        } catch (SQLException e) { throw new StoringException(); }
    }
```

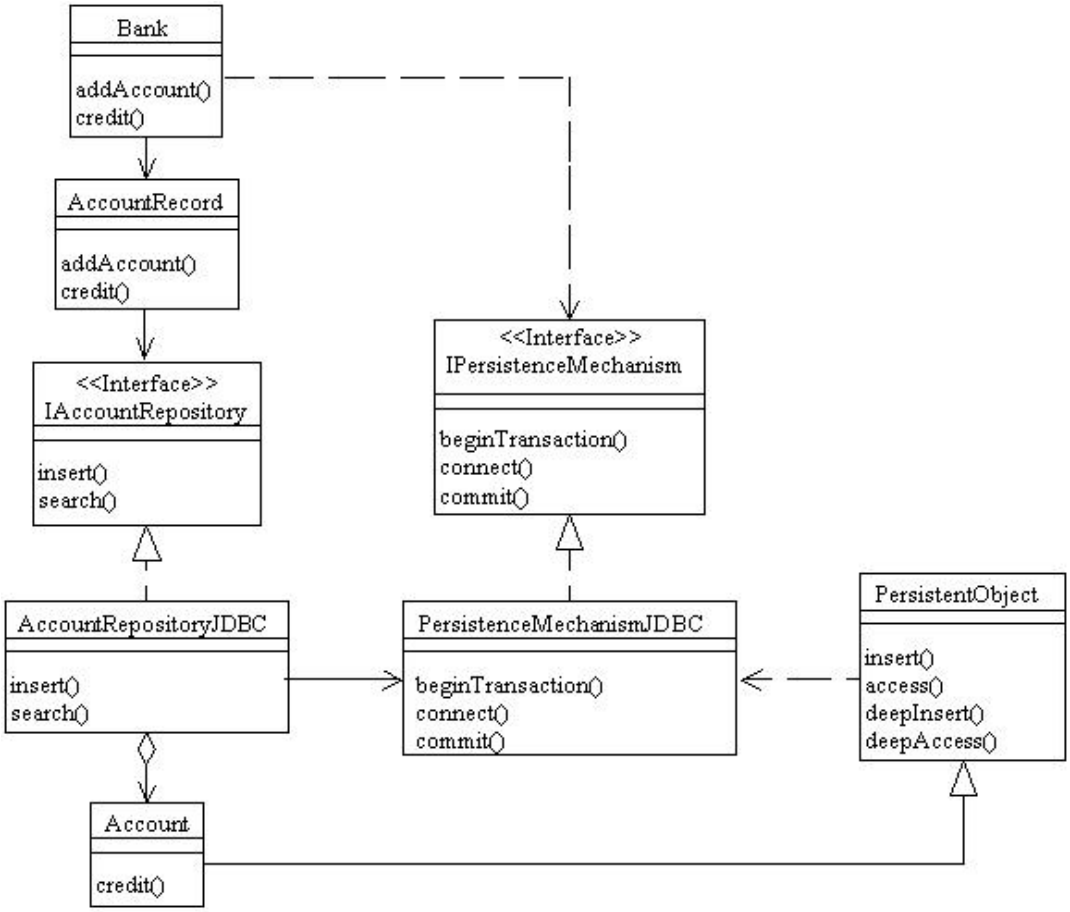


Figure 3: Example of PDC applied to a banking application.

Two of the attributes and one business operation, `credit` (containing only business code and not invoking any data access method), are presented above. In another portion of the class, there are data access methods inherited from the `PersistentObject` class, containing specific code for database operations in this class (as the `insert` method). Any exception related to the data access API (`SQLException`) is replaced by a general database exception (`StoringException`).

In addition, this class contains methods with the `deep` prefix, which are special operations for manipulating attributes which are references to other objects or collection of objects (as the `number` attribute). The `deepInsert` method in the `Account` class has an `IPersistenceMechanism` interface parameter receiving a reference to a persistence mechanism object in order to perform the corresponding database operation:

```
    void deepInsert (IPersistenceMechanism pm)
           throws StoringException {
       super.pm = pm;
       this.number.deepInsert(pm);
       this.insert();
    }
    ...
}
```

Notice that `deepInsert` is called first for the attribute, before the `insert` for the `Account` object. This order is followed in operations to write data to the database, due to a restriction of relational databases, which forces the code to insert rows in auxiliary tables first (`number` attribute), then insert a row in the main table (`Account` object). In this way, the relationships can be established with no errors. This order does not need to be followed in operations querying the database. Operations deleting data from the database depend on the referential integrity defined for the tables involved.

Although there is business code along with data access code in the same class, the business methods do not depend on the data access methods, since the former do not invoke the latter. Therefore, we can insert and remove data access methods with no impact on business code (a process easily automated by tools). The `PersistentObject` class is presented below:

```
abstract class PersistentObject {
    protected long id;
    protected IPersistenceMechanism pm;
    abstract void insert() throws StoringException;
    abstract void deepInsert(IPersistenceMechanism pm)
        throws StoringException;
    abstract void access() throws StoringException;
    abstract void deepAccess(IPersistenceMechanism pm)
        throws  StoringException;
    ...
}
```

where the `id` and the `pm` attributes denote the object identity of a persistent object and a persistence mechanism object to perform database operations, respectively. The abstract data access methods in this class must be implemented by all business basic classes, which will be made persistent. The `StoringException` exception is raised when a problem occurs in any database operation.

In order to represent a set of business basic objects on the business' vision, we use a business collection class. We present the class `AccountRecord`, which represents a set of bank accounts:

```
class AccountRecord {
    private IAccountRepository accountsRep;
    AccountRecord(IAccountRepository accountsRep) {
        this.accountsRep = accountsRep;
    }
```

where the constructor of `AccountRecord` receives as argument an object which implements a business–data interface, and two of the business operations for this class, `addAccount` and `credit`, are also presented. The first method inserts an `Account` object into the database, raising an exception if an account with the same number already exists.

```
    void addAccount(Account account)
            throws StoringException, DuplicateAccountException {
        if (this.accountsRep.exists(account.getAccountNumber()))
            throw new DuplicateAccountException();
        else  this.accountsRep.insert(account);
    }
```

The second method queries the database for a given account. If the query is successful, a value is added to the account's balance and the account is updated in the database. However, if the account does not exist in the database, an exception is raised.

```
    void credit(Number accountNumber, double value)
            throws StoringException, UnknownAccountException {
        if (accountsRep.exists(accountNumber)) {
            Account account = accountsRep.search(accountNumber);
            account.credit(value);
            this.accountsRep.update(account);
        }
        else  throw new UnknownAccountException();
    }
    ...
}
```

The database is represented by the attribute `accountsRep`, a business–data interface with data access operations. This interface is as follows:

```
interface IAccountRepository {
    void insert(Account account) throws StoringException;
    Account search(Number accountNumber) throws StoringException;
    void update(Account account) throws StoringException;
    boolean exists(Number accountNumber) throws StoringException;
    ...
}
```

where the `update` method is important to maintain consistency between in–memory (volatile) and persistent objects. Other methods on this interface could be complex queries (for instance, returning a set of objects) and methods for sequential querying.

A class implementing a business–data interface is a persistent data collection class. In our example, this class implements its methods invoking data access methods defined in the business basic classes. In our example, the `AccountRepositoryJDBC` class is presented as follows:

```
class AccountRepositoryJDBC implements IAccountRepository {
    private PersistenceMechanismJDBC pm;
    void insert(Account account) throws StoringException {
        account.deepInsert(this.pm);
    }
```

Note that the `pm` attribute stores a persistence mechanism object, which is passed as an argument for the database operations on `Account` objects, as in the `search` method.

```
    Account search(Number accountNumber) throws StoringException {
        Account ac = new Account(accountNumber);
        ac.deepAccess(this.pm);
        return ac;
    }
    ...
}
```

On the other hand, if it is desired to develop a functional prototype first, we can implement a business–data interface using a volatile data collection. In the banking application, we can create a class which stores and retrieves `Account` objects from an array. The objects will be maintained in the array only during the current execution.

The facade class of the pattern is represented by the `Bank` class in this application:

```
class Bank {
    private IPersistenceMechanism pm;
    private AccountRecord accounts;
    Bank() throws PersistenceMechanismException {
        PersistentFactory factory = PersistentFactory.getFactory();
        this.pm = factory.createPersistenceMechanism();
        this.accounts = new AccountRecord(
             AccountDataFactory.getFactory().createDataCollection(pm));
    }
    void addAccount(Account account)
            throws StoringException, AccountAlreadyExistsException {
        this.pm.beginTransaction();
        try {   this.accounts.add(account);  }
        catch (Exception e) {
            this.pm.cancelTransaction();
            throw e;
        }
        this.pm.commitTransaction();
    }
    void credit(String accountNumber, double value)
            throws StoringException, UnknownAccountException {
        this.pm.beginTransaction();
        try {  this.accounts.credit(accountNumber,value);  }
        ...
    }
    ...
}
```

This persistence mechanism object is instantiated in the `Bank`'s constructor, in order to initialize the system, being stored in an `IPersistenceMechanism` interface attribute. All the initialization process is performed using a `PersistenceFactory` class, which reads a configuration file and creates the right specific persistence factory object for the application. This object will then create the specific persistence mechanism object for the `Bank` class, promoting extensibility of the business code (the facade class does not instantiate the persistence mechanism object directly). See the Implementation section.

`Bank` uses services from its `AccountRecord` attribute, delegating calls to the latter in its methods. This attribute is initialized by passing as argument a new persistent data collection object, which implements a business–data interface and receives a persistence mechanism object. In order to maintain separation between business and data access code, this persistent data collection object is instantiated by a specific data factory for JDBC, which in turn was first instantiated by a static method (`getFactory`) in an abstract `AccountDataFactory` class (see Implementation section). In the `addAccount` and `credit` methods, the `Facade` class invokes methods on the persistence mechanism object for beginning and confirming a transaction, or canceling it if some exception occurs.

The `IPersistenceMechanism` interface, which is used by `Bank`, is presented as follows:

```
interface IPersistenceMechanism {
    void beginTransaction() throws PersistenceMechanismException;
    void commitTransaction() throws PersistenceMechanismException;
    void cancelTransaction() throws PersistenceMechanismException;
    void connect() throws PersistenceMechanismException;
    void disconnect() throws PersistenceMechanismException;
    ...
}
```

where `PersistenceMechanismException` is the exception raised when some error occurs in one of those operations. A persistence mechanism class implements this interface using specific database API operations, as in the following example:

```
class PersistenceMechanismJDBC implements IPersistenceMechanism {
    void beginTransaction() throws PersistenceMechanismException {
        try {
            // requests a connection from a connection pool
            Connection conn = this.requestConnection();
            conn.setAutoCommit(false);
        }
        catch (SQLException e) {
            throw new PersistenceMechanismException();
        }
    }
    ...
}
```

This class implements the `beginTransaction` method using services from the JDBC API. First, a connection to the database is requested from a connection pool (allowed by JDBC). If there is not any opened connection, a new one is created. Then a transaction is initialized in the context of the connection. Any `SQLException` raised is replaced by a general exception, in order to guarantee isolation between business and data access code.

Known Uses

Several organizations have been using PDC as a design pattern for many real software projects. Most of these projects have aimed at developing from simple to complex ap-

plications, and satisfactory results have been collected in such situations. Some of these systems are presented as follows:

- A system to manage clients of a telecommunication company. The system is able to register mobile telephones and manage client information and telephone services configuration. The system can be used over the Internet.
- A system for performing online exams. This system has been used to offer different kinds of exams, such as simulations based on previous university entry exams, helping students to evaluate their knowledge before the real exams.
- A complex supermarket system. A system that is responsible for the control of sales in a supermarket. This system will be used in several supermarkets and is already been used in other kinds of stores.
- A system for registering health system complaints. The system allows citizens to complaint about health problems and to retrieve information about the public health system, such as the location or the specialties of a health unit.
- This pattern is also used in undergraduate and graduate courses on object–oriented programming at the Center of Computer Science of the Federal University of Pernambuco. Several kinds of systems (such as games, academic control systems, and sales systems) have been developed in these courses.

In addition, the pattern is one of the basic patterns of the Progressive Implementation Method (Pim) [5]. Pim is a method for the systematic implementation of complex object–oriented applications in Java. In particular, this method supports a progressive approach for object–oriented implementation, where persistence, distribution and concurrency control are not initially considered in the implementation activities, but are gradually introduced, preserving the application's functional requirements [1, 9, 11, 15]. Pim relies on the use of specific architectural and design patterns for structuring object–oriented applications, in order to promote modularity and separation of concerns [10]. PDC is the design pattern applied for dealing with persistence.

Related Patterns

- *Crossing Chasms [6].* In their set of patterns for object–relational integration, Brown and Whitenack deal with the definition of database schemas for relational databases, supporting the object model. These patterns can be useful in PDC (for setting up the database tables), since they have distinct objectives (PDC aims at structuring the application in layers for a seamless introduction of persistence).
- *Persistent Layer and other patterns [16].* Yoder's patterns and PDC have very similar objectives in obtaining separation of concerns between business and data access code. Many of the ideas presented in the Yoder's patterns can be combined into elements of PDC in a practical way (for instance, Transaction Manager and Connection Manager can be instantiated as the PDC's persistence mechanism class). However, Yoder's patterns do not separate definitions of "data" and "data set", as defined in our persistent data collections, and assuming to be applied specifically

to relational databases. We believe that PDC can be applied almost directly to a number of persistence platforms, including object databases and files.

- *Abstract Factory [7].* This pattern is applied in PDC to implement a persistence factory class for creating persistence mechanism objects, which is used by a facade class. Factories also can be used for creating persistent data collection objects transparently for the business collection classes (see Implementation section).

- *Facade [7].* The facade class of PDC is a direct implementation of the Facade pattern.

- *Singleton [7].* Usually only one facade object is required in an application. Thus facade objects are often implemented as Singletons.

- *Bridge [7].* This pattern is used in PDC as the business–data and persistence mechanism interfaces, which play the role of a bridge between the business and the data access layers.

- *Concurrency Manager [13].* This pattern can be used in PDC to control concurrent situations, such as interferences by business rules (system policies), unsafe data types, and other race conditions.

Acknowledgements

We would like to give special thanks to our shepherd in this paper, Rosana Teresinha Vaccare Braga, from ICMC-USP, for making important suggestions for improving this pattern. We also thanks Jorge L. Ortega Arjona and Gunter Mussbacher for the suggestions made at the conference.

References

[1] Vander Alves. Progressive Development of Distributed Object-Oriented Programs. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, February 2001.

[2] Scott Ambler. *Building Object Applications that Work*. Cambridge University Press and Sigs Books, 1998.

[3] Scott Ambler. *The Object Primer*. Cambridge University Press, 2001.

[4] Grady Booch et al. *The Unified Modeling Language User Guide*. Object Technology. Addison-Wesley, 1999.

[5] Paulo Borba, Saulo Araújo, Hednilson Bezerra, Marconi Lima, and Sérgio Soares. Progressive implementation of distributed Java applications. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 40–47, Los Angeles, USA, 17th–18th May 1999.

[6] K. Brown and B. Whitenack. Crossing Chasms: A Pattern Language for Object-RDBMS Integration. In J. Vlissides et. al. (eds.), Pattern Languages of Program Design 2. Addison-Wesley, 1996.

[7] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[8] James Gosgling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[9] Tiago Massoni. A Software Process with Support to Progressive Implementation (in portuguese). Master's thesis, CIn – Federal University of Pernambuco, February 2001.

[10] David L. Parnas et al. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of ACM*, 15(12):1053–1058, December 1972.

[11] Sérgio Soares. Progressive Development of Concurrent Object-Oriented Programs (in portuguese). Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, February 2001.

[12] Sérgio Soares and Paulo Borba. Concurrency Control with Java and Relational Databases (in portuguese). In *V Brazilian Symposium of Programming Languages*, 23th–25th May 2001.

[13] Sérgio Soares and Paulo Borba. Concurrency Manager. Technical report, State University of Rio de Janeiro—UERJ, Rio de Janeiro, Brazil, 3th–5th October 2001. To appear.

[14] Sun Microsystems. Java Database Conectivity Specification, 2000. Available at ftp://ftp.javasoft.com/pub/jdbc.

[15] Euricélia Viana. Integrating Java with Relational Databases (in portuguese). Master's thesis, Centro de Informática, UFPE, 2000.

[16] J.W. Yoder, R.E. Johnson, and Q.D. Wilson. Connecting Business Objects to Relational Databases. In *Proceedings of the 5th Conference on the Pattern Languages of Programs*, Monticello-IL-EUA, August 1998.