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.

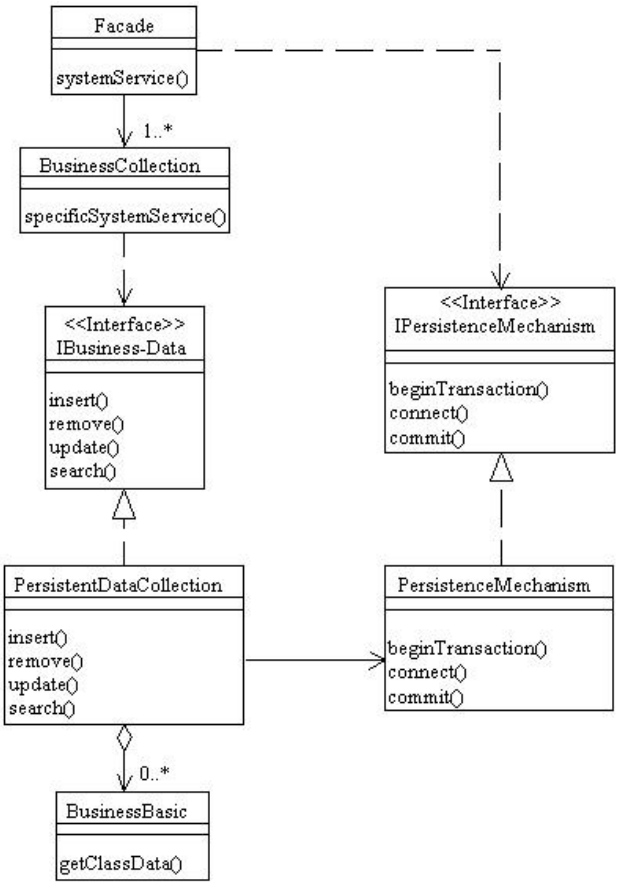


Figure 1: Class diagram of PDC.

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

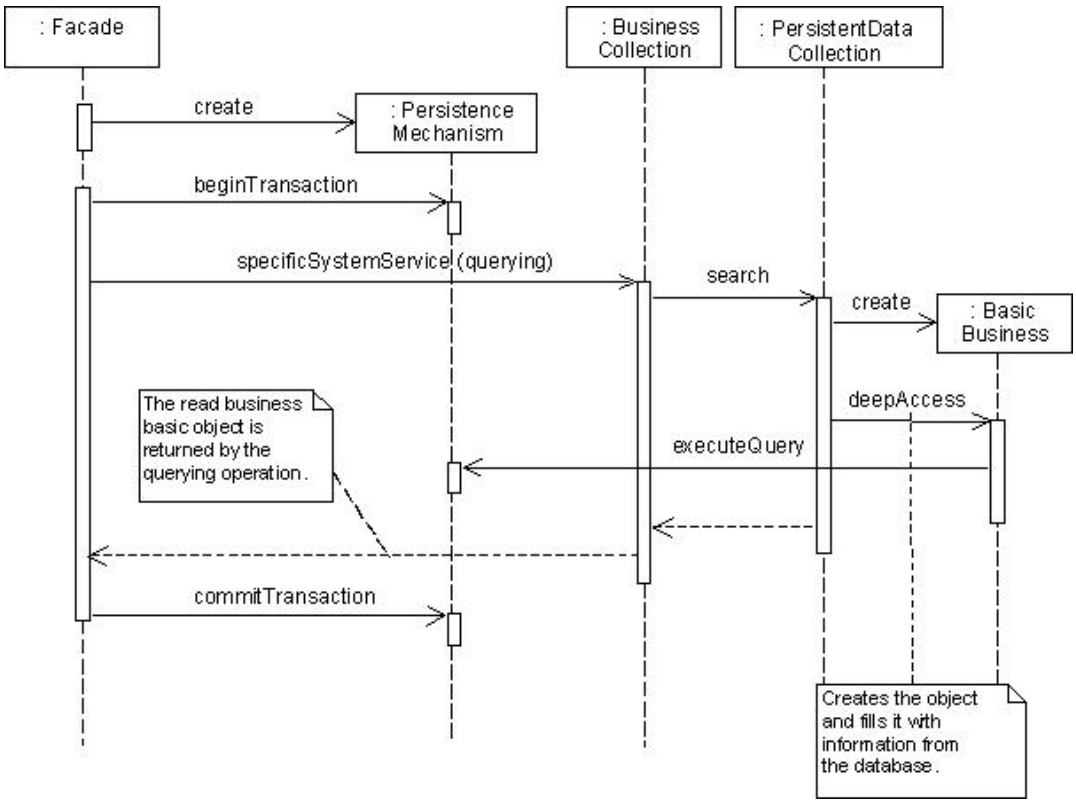


Figure 2: Dynamics of PDC.

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(); }
    }
```

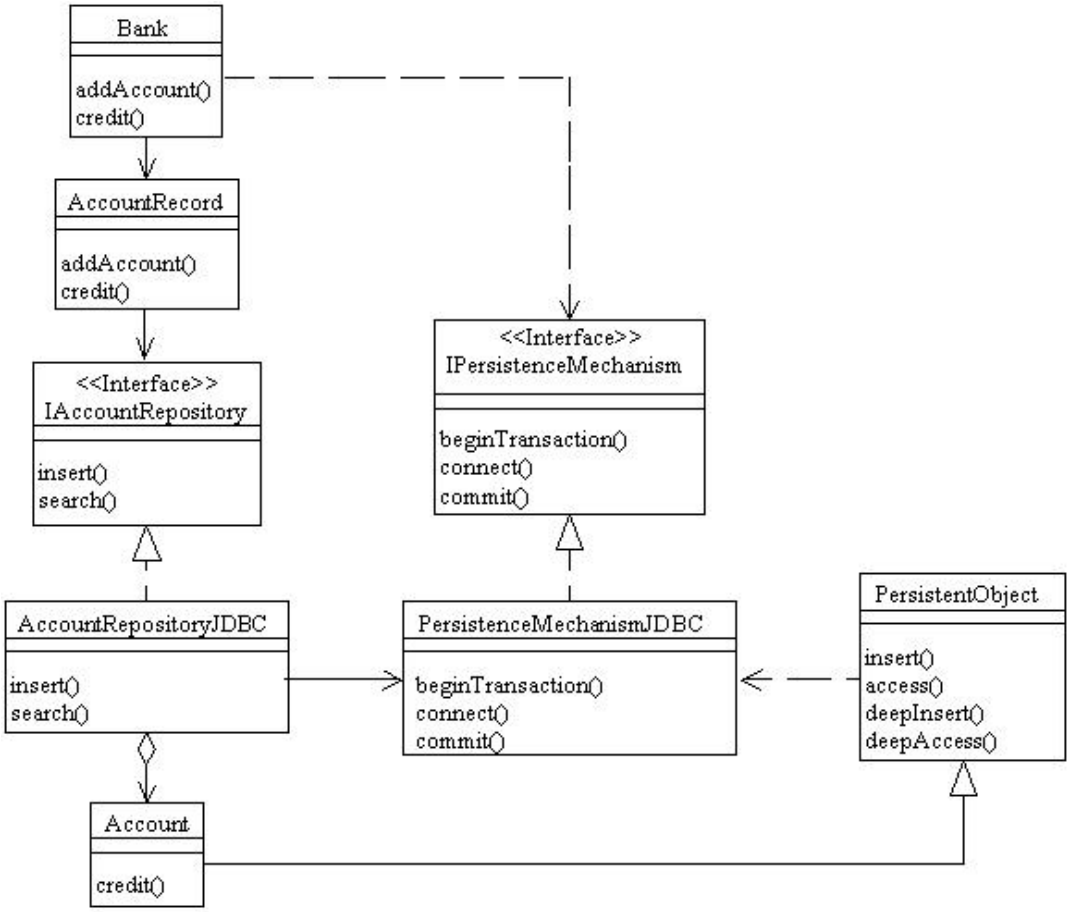


Figure 3: Example of PDC applied to a banking application.

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 ICMCSC-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.

Uma Linguagem de Padrões para o Projeto Sintático de Linguagens de Descrição de Arquitetura

Cidcley Teixeira de Souza, Paulo Roberto Freire Cunha
Universidade Federal de Pernambuco
Centro de Informática
Av. Prof. Luiz Freire, s/n - Cidade Universitária
50732-970, Recife, PE, Brasil
{cts,prfc}@cin.ufpe.br

Jerffeson Teixeira de Souza
University of Ottawa
School of Information Technology and Engineering
K1N 6N5, Ottawa, ON, Canada
jsouza@site.uottawa.ca

Resumo

O objetivo das linguagens de descrição de arquitetura (ADLs) é permitir a especificação de arquiteturas de software de forma que seja possível a realização de análises estruturais e comportamentais nessas arquiteturas antes de sua implementação. Esse artigo apresenta uma linguagem de padrões que captura as principais decisões necessárias à construção de uma ADL. O enfoque principal desses padrões é a definição, organização e representação de elementos sintáticos para essas linguagens.

Keywords: Arquitetura de Software, Linguagens de Descrição de Arquitetura, Padrões de Projeto.

1 Introdução

De uma forma simplificada, arquitetura de software pode ser definida como sendo a representação da "estrutura dos componentes de um programa/sistema, seus inter-relacionamentos, princípios e regras que governam seu projeto e evolução ao longo do tempo" [Gar00]. O desenvolvimento de um projeto arquitetural permite a visualização de algumas propriedades gerais do software antes que este seja implementado, garantindo que algumas de suas características sejam explicitadas.

Contudo, a construção de arquiteturas de software para grandes aplicações é uma tarefa complexa. A utilização de padrões tem auxiliado nessa tarefa possibilitando uma abordagem mais precisa na definição dessas arquiteturas. Em Kane [DHKM97], por exemplo, são apresentados padrões que tratam da organização e gerenciamento de arquiteturas de softwares; em Meszaros [Mes97] padrões são utilizados para capturar o processo de definição de arquiteturas; em Shaw [Sha96] são apresentados padrões para a organização e definição de estilos de interação entre elementos arquiteturais.

Para formalizar a representação de projetos arquiteturais foram criadas as ADLs *(Architecture Description Languages)*. Essas linguagens fornecem um *framework* sintático e um conjunto de ferramentas que possibilitam a especificação de projetos arquiteturais e a realização de diversos tipos de análises nessas arquiteturas. Exemplos de ADLs incluem Darwin [MDEK95], UniCon [SDD95], Meta-H [BV93], Wright [AG97], Acme [GMW97] entre outras. O Apêndice A apresenta mais informações sobre arquitetura de software e ADLs.

Entretanto, a ploriferação de ADLs tem dificultado a decisão de qual linguagem adotar para realizar um determinado projeto arquitetural. Principalmente se considerado que cada ADL possui características específicas e ferramentas que realizam tarefas também específicas.

Além do mais, cada projeto exige que tipos diferentes de informações sejam especificadas para os diversos elementos arquiteturais que compõem o projeto, e tanto a escolha de quais informações utilizar como a forma com que essas informações serão representadas estão sujeitas às características da ADL adotada no projeto.

Essa especificidade das ADLs acaba tornando necessário a adoção de mais de uma dessas linguagens para a realização de um mesmo projeto arquitetural, de modo a se conseguir atingir todos os objetivos do projeto. Contudo, fatores como tempo e custos necessários para se dominar essas linguagens acabam sendo, em alguns casos, proibitivos no desenvolvimento de alguns projetos arquiteturais com ADLs.

Nesse caso, pode ser razoável se optar pela construção de uma nova ADL pela própria equipe de desenvolvimento, onde as característica dessa linguagem possam ser definidas de acordo com o perfil da equipe que vai utilizá-la. Com uma ADL definida pela própria equipe, os projetos arquiteturais serão desenvolvidos mais rapidamente, visto que não haveria a necessidade da adoção de outras ADLs, e até mesmo o tempo de adaptação da equipe à nova linguagem seria extremamente reduzido.

Para facilitar o processo de desenvolvimento de uma nova ADL, esse artigo apresenta uma linguagem de padrões que captura o processo decisório relativo ao projeto da sintaxe de ADLs. Esses padrões auxiliam a definir aspectos importantes relativos à forma com que a sintaxe de uma ADL deve ser projetada, de modo a ser de fácil manuseio e flexível no que diz respeito à definição de informações sobre arquiteturas de software.

1.1 Motivação

A consecução de um bom projeto arquitetural pode ser um fator determinante no sucesso ou fracasso do desenvolvimento de um software complexo. Entretanto, essa fase do ciclo de vida do software é normalmente desprezada pela maioria dos projetistas. Isso se deve principalmente ao fato da expressividade limitada dos diagramas do tipo "caixas-e-linhas"que são normalmente utilizados para representar os projetos arquiteturais.

Uma outra solução é a utilização de uma ADL (*Architecture Description Language*), que é uma linguagem específica para de modelar projetos arquiteturais. Contudo, as notações da maioria dessas linguagens são difíceis, o que pode afetar os prazos para o desenvolvimento do software, que são normalmente apertados.

Nesse sentido, uma solução viável para esse problema é a construção de uma ADL que seja mais fácil de ser manipulada pela equipe de desenvolvimento. Essa solução trás resultados bastante satisfatórios, principalmente a médio e longo prazos.

Tendo essa motivação em mente, e contando com a experiência de desenvolvimento de uma ADL, a linguagem ArchML [SC01], apresentamos nesse artigo uma linguagem de padrões que ressalta os principais problemas na realização do projeto sintático de uma nova ADL.

Essa linguagem de padrões tem como base na sua proposta a dedução das motivações e do processo de decisões envolvido na criação das estruturas sintáticas de diversas ADLs disponíveis atualmente. Além disso, todo esse processo decisório foi validado através de sua aplicação na construção de ArchML. O que nos faz concluir que esses padrões realmente capturam a filosofia embutida nos processos de decisão relativos ao projeto sintático de uma ADL.

1.2 Estrutura da Linguagem de Padrões

Os problemas tratados nessa linguagem de padrões e as soluções propostas para estes problemas são resumidos na Tabela 1. A Figura 1 apresenta as dependências entre os padrões propostos nesse artigo. Nessa figura, estão apresentadas em linhas cheias as dependências diretas entre os padrões, que indicam a necessidade de se ter concluído o padrão para se iniciar o seguinte. Por sua vez, a linha tracejada define dependências que não possuem ordem de precedência ou padrões que devem ser desenvolvidos em paralelo. A ordem com que esses padrões devem ser aplicados é a mesma com que eles são descritos ao longo do artigo.

O padrão *Projetar ADL*, apresentado na seção 2, é o padrão principal da linguagem. Ele é quem justifica a criação de uma nova ADL e identifica os padrões a serem utilizados para o projeto sintático dessa ADL. O padrão *Elementos Arquiteturais Básicos*, apresentado na seção 3, identifica quais os tipos de elementos arquiteturais devem ser descrito por uma ADL. A estruturação de uma especificação arquitetural a partir dos elementos que contituem sua arquitetura é definida no padrão *Organização das Especificações*, tratado na seção 4. Na seção 5 é apresentado o padrão *Informações dos Elementos Arquiteturais*, que descreve como se decidir a forma com que uma ADL deve especificar as informações para cada elemento arquitetural. O padrão *Estrutura Hierárquica de Informações* apresentado na seção 6 define a forma de como se organizar a apresentação das informações sobre os elementos arquiteturais.

Padrão	**Problema**	**Solução**
Projetar ADL	Como minimizar esforço e custo na especificação de arquiteturas de software quando há a necessidade de se utilizar diversas ADLs ?	Desenvolva uma nova ADL que contemple as principais necessidades identificadas pela equipe de desenvolvimento para a representação de seus projetos arquiteturais.
Elementos Arquiteturais Básicos	Que tipos de elementos arquiteturais devem ser representados pela ADL de forma a se criar especificações arquiteturais fáceis de ser construídas e mantidas ?	Defina tipos de elementos básicos para representar elementos arquiteturais, onde cada tipo de elemento deverá ter uma funcionalidade específica na arquitetura.
Organização das Especificações	Como organizar a especificação da arquitetura de um software a partir das especificações de seus elementos arquiteturais constituintes ?	Especifique em apenas um arquivo tanto os elementos arquiteturais como a arquitetura do software em si.
Informações dos Elementos Arquiteturais	Quais informações deverão estar disponíveis na ADL para a especificação de cada elemento arquitetural ?	Desenvolva a ADL de forma a deixar o projetista livre para decidir quais informações representar em cada elemento arquitetural. Entretanto, identifique algumas informações que devem ser obrigatórias para cada elemento e torne todas as demais opcionais.
Estrutura Hierárquica de Informações	Como a ADL deve representar as informações dos elementos arquiteturais de forma a facilitar o entendimento dessas informações ?	Crie um conjunto de contextos para cada informação e, dentro de cada contexto, defina de forma hierárquica as informações pertinentes ao contexto.

Tabela 1: Problemas e Soluções dos Padrões de Projeto Sintático de ADLs.

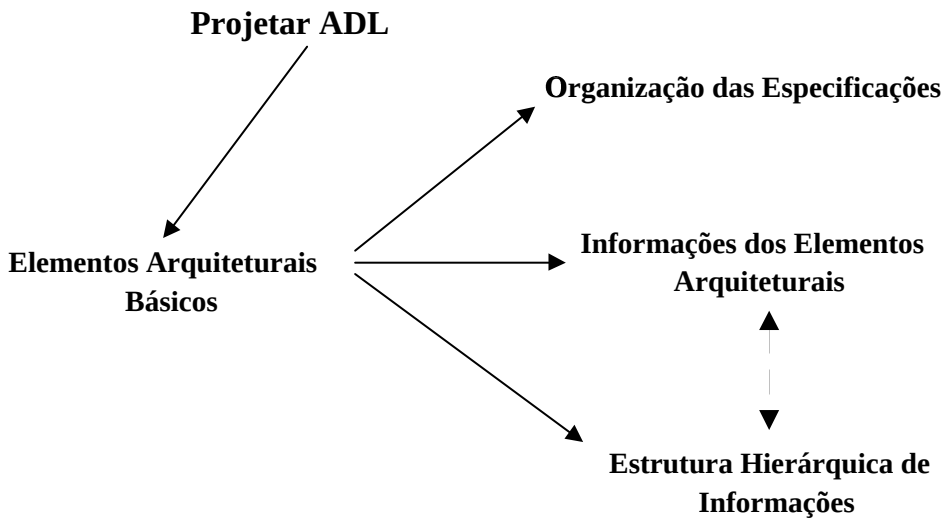


Figura 1: Dependências da Linguagem de Padrões.

2 Projetar ADL

Contexto
Você está trabalhando em um sistema de software complexo no qual um grande número de requisitos devem ser considerados. Além disso, é necessário a realização de diversos tipos de simulações e análises nesse sistema. Para suavizar a passagem da fase de elicitação e análise dos requisitos para a fase de implementação deve ser especificada uma arquitetura para o software. Contudo, para se conseguir uma representação arquitetural que permita a realização das análises requeridas é necessário a utilização de uma ADL. Entretanto, é bastante difícil encontrar uma ADL que sozinha possua todas as características necessárias para representar essa arquitetura e forneça todas as ferramentas necessárias para a realização das análises requeridas. Nesse sentido, surge a necessidade de se utilizar mais de uma ADL para especificar a arquitetura desse sistema.

Problema
Como minimizar esforço e custo na especificação de arquiteturas de software quando há a necessidade de se utilizar diversas ADLs ?

Forças

- Quanto mais complexo é o projeto, mais difíceis são as análises a serem realizadas, o que exige a utilização de ADLs que possuam ferramentas específicas.
- Ao se utilizar ADLs diferentes para realizar tarefas específicas em um projeto arquitetural surge a necessidade de treinar a equipe de desenvolvimento para entender a sintaxe e as ferramentas de cada nova ADL a ser utilizada, o que aumenta o tempo de desenvolvimento do projeto.

- Os custos de um projeto aumentam significativamente com o treinamento da equipe e com a necessidade de adaptação da arquitetura para cada nova ADL utilizada em um determinado projeto arquitetural.

- A equipe de desenvolvimento pode optar em adotar uma ADL específica que mais se aproxime das características do projeto e da formação dessa equipe. Entretanto, isso limita a expressividade dos projetos às características da ADL escolhida.

Solução
Desenvolva uma nova ADL que contemple as principais necessidades identificadas pela equipe de desenvolvimento para a representação de seus projetos arquiteturais. Essas características devem ser identificadas e refinadas a partir do acúmulo de experiência dessa equipe, adquirida com a especificação de arquiteturas de software utilizando outras ADLs existentes.

Para se criar uma nova ADL, diversos aspectos devem ser considerados. Dentre eles, o poder de representação das informações requeridas pela equipe de desenvolvimento para a especificação dos elementos arquiteturais é um dos mais importantes. A forma com que a ADL fragmenta a representação de arquiteturas complexas em elementos arquiteturais mais simples tem influência direta na manutenibilidade dessa arquitetura.

Contexto Resultante
A aplicação desse padrão resulta na decisão de se construir uma nova ADL que melhor se adapte às necessidades específicas dos projetos arquiteturais de uma determinada equipe de desenvolvimento.

Racionalização
Embora seja mais caro se produzir uma nova ADL, o custo de treinamento da equipe para entender determinadas ADLs, ou adaptar uma arquitetura para uma ADL específica, pode ser muito mais alto. Principalmente se for considerado que cada projeto tem características diferentes sendo que, normalmente, há a necessidade de se utilizar uma ADL que se adapte melhor a cada um dos projetos.

Ao se desenvolver uma nova ADL, a equipe terá a facilidade de adaptá-la às suas necessidades sempre que for necessário, sem necessidade de treinamento para entender a ADL. Com o passar do tempo, e com a acúmulo de experiência da equipe, a ADL estará estável o suficiente para suprir toda a demanda de análise arquitetural necessária sem gasto extra. Além disso, as ferramentas poderão ser desenvolvidas pela própria equipe, de modo a se conseguir realmente os resultados desejados.

Consequências
Algumas das vantagens do padrão Projetar ADL são as seguintes:

- **Uniformidade na representação arquitetural.** Com apenas uma ADL todos os projetos arquiteturais da equipe terão apenas uma linguagem de representação, o que facilita a compreensão dos projetos e cria um vocabulário comum entre os membros da equipe de desenvolvimento.

- **Agilização do projeto arquitetural.** Com a adaptação da equipe de desenvolvimento à nova ADL, todos os novos projetos que se seguem serão mais rapidamente realizados.

- **Tempo menor de aprendizagem.** Se for necessário a mudança de membros da equipe, o tempo de treinamento para a incorporação desses novos membros será bastante reduzido.

- **Adaptabilidade.** Por ser um produto da equipe de desenvolvimento, a ADL poderá ser adaptada facilmente para novas necessidades que surgirem.

Algumas das desvantagens do padrão Projetar ADL são as seguintes:

- **Custo inicial alto.** A implementação de uma nova ADL vai requerer esforço, tempo e, portanto, um orçamento extra para essa tarefa. Entretanto, futuras atualizações tendem a ter custos bem reduzidos.

- **Implementação de ferramentas.** Além da ADL em si, as ferramentas que permitirão à realização de análises nas arquiteturas, também devem ser implementadas.

Padrões Relacionados
Para que a implementação de uma nova ADL seja viabilizada, vários aspectos devem ser considerados a priori. Esses aspectos dizem respeito principalmente às características sintáticas da ADL, onde a primeira decisão a ser tomada está relacionada com a forma com que os elementos arquiteturais são identificados pala ADL. O padrão *Elementos Arquiteturais Básicos* (seção 3) trata desse problema.

3 Elementos Arquiteturais Básicos

Contexto
De acordo com o padrão *Projetar ADL*, você decidiu criar uma ADL para especificar a arquitetura de um software complexo. Essa tarefa é iniciada pela definição dos tipos de elementos arquiteturais que a ADL deve possuir.

Problema
Que tipos de elementos arquiteturais devem ser representados pela ADL de forma a se criar especificações arquiteturais fáceis de ser construídas e mantidas ?

Forças

- A definição dos tipos de elementos arquiteturais a serem especificados para uma arquitetura depende muito da experiência da equipe de desenvolvimento.

- Os tipos de elementos arquiteturais a serem definidos para a ADL influencia diretamente a gerenciabilidade e a clareza das especificação realizadas por essa linguagem. Sendo que um aumento exagerado no número de tipos de elementos especificados

resulta em uma arquitetura de difícil gerenciamento. Por outro lado, a construção de uma especificação sem a definição clara dos tipos de elementos arquiteturais representados pode tornar a arquitetura ilegível.

- O potencial de reuso de elementos de especificações arquiteturais é definido pela forma com que esses elementos são descritos pela ADL. Sendo que quanto mais clara e precisa for a classificação dos tipos de elementos arquiteturais melhor será o reusabilidade dessas especificações.

Solução
Defina tipos de elementos básicos para representar elementos arquiteturais, onde cada tipo de elemento deverá ter uma funcionalidade específica na arquitetura, de forma a tornar clara a especificação como um todo. Desse modo os seguintes tipos de elementos arquiteturais podem ser identificados:

- Interfaces - Devem descrever as funcionalidades requeridas e fornecidas pelos elementos arquiteturais.
- Componentes - Devem descrever as partes da arquitetura que realizam algum tipo de computação.
- Conectores - Devem descrever as estruturas de comunicação entre os componentes.
- Configurações Arquiteturais - Devem descrever a forma com que componentes e conectores são organizados para formar uma arquitetura.

Contexto Resultante
A aplicação desse padrão identifica os tipos de elementos necessários à ADL para a construção de arquiteturas de software. Dessa forma, cada elemento a ser representado em uma especificação de arquitetura está relacionado a um tipo específico.

Racionalização
A definição de tipos de elementos arquiteturais em uma especificação facilita sobremaneira o entendimento da mesma. Nesse padrão, também foi adotada a abordagem de se especificar separadamente as interfaces dos componentes e dos conectores. Essas interfaces descrevem o que cada elemento fornece e utiliza de outros elementos. Com essa abordagem, existe a possibilidade de reuso de especificações de interfaces e de uma melhor checagem de tipos entre elementos arquiteturais distintos, além de tornar mais intuitiva a tarefa de particionar a arquitetura.

Usos Conhecidos
ArchML utiliza a solução completa apresentada nesse padrão para a decomposição de elementos arquiteturais. Essa linguagem possui uma sintaxe baseada em XML para representar os elementos arquiteturais.

A ADL Darwin utiliza os conceitos de componentes e arquiteturas, mas não representa conectores explicitamente, podendo esses serem simulados a partir de especificações

de componentes. Nessa linguagem as interfaces dos elementos são descritas dentro dos próprios elementos arquiteturais.

UniCon representa os elementos arquiteturais através de componentes, conectores e configurações. Entretanto, os conectores de UniCon são predefinidos como parte da própria ADL.

Wright também utiliza os conceitos de componentes, conectores e configurações. Entretanto, nessa ADL os conectores podem ser definidos pelo usuário e sua semântica é especificada através de CSP.

Variação

A abordagem clássica utilizada na definição de tipos de elementos arquiteturais em ADLs sugere a utilização dos seguintes elementos apenas: Componentes, Conectores e Configurações Arquiteturais. Nessa abordagem, as interfaces tanto dos componentes como dos conectores, são definidas internamente a esses elementos, não podendo ser reutilizadas em outras arquiteturas.

Conseqüências

Algumas das vantagens do padrão Elementos Arquiteturais Básicos são as seguintes:

- **Reusabilidade.** A separação da especificação de interfaces dos elementos arquiteturais permite a reusabilidade dessas interfaces em outros projetos.

- **Previsibilidade de comportamento.** A definição de tipos para cada elemento arquitetural, permite se inferir que comportamento um determinado elemento vai ter dentro de projeto arquitetural.

Uma desvantagem do padrão Elementos Arquiteturais Básicos é a seguinte:

- **Configurações mais complexas.** Além da definição das interações entre componentes e conectores, os próprios componentes e conectores devem indicar quais interfaces devem utilizar, isso pode tornar as arquiteturas um pouco mais complexas.

Padrões Relacionados

Um aspecto importante a ser considerado na especificação da sintaxe de uma ADL é a forma com que os elementos arquiteturais especificados são organizados para produzir a especificação de uma arquitetura. Esse problema é tratado no padrão *Organização das Especificações*, apresentado na seção 4.

Um outro aspecto também importante é a representação das informações sobre os elementos que compõem as arquiteturas. Essas informações devem ser de fácil compreensão e gerenciamento, e deve ser possível a inclusão de informações não previstas sobre determinados elementos arquiteturais sem ter que reescrever a ADL. Para se determinar os tipos de informações necessárias para representar elementos arquiteturais pela ADL utilize o padrão *Informações dos Elementos Arquiteturais*, apresentado na seção 5.

4 Organização das Especificações

Contexto
Foi definida a necessidade de se criar uma nova ADL, apresentada no padrão *Projetar ADL*. Também foram definidos os tipos de elementos arquiteturais básicos que essa ADL pode representar, relacionados no padrão *Elementos Arquiteturais Básicos*. Esses elementos arquiteturais devem ser utilizados para se criar a representação de arquiteturas de software.

Problema
Como organizar a especificação da arquitetura de um software a partir das especificações de seus elementos arquiteturais constituintes ?

Forças

- A organização da especificação da arquitetura de um software realizada em um único arquivo junto com a especificação de seus elementos arquiteturais constituintes, garante uma manipulação mais fácil dessa especificação, entretanto a reusabilidade dos elementos arquiteturais fica comprometida.
- Dependendo do tamanho da equipe que está produzindo a especificação arquitetural pode ser necessários desmembrar a especificação em partes menores. Sendo que quanto maior a equipe, maior pode ser a fragmentação necessária.
- A complexidade da especificação arquitetural e a estabilidade dos requisitos que a gerou, pode determinar a forma com que os elementos que compõem a arquitetura serão organizados. Sendo que quanto mais complexa for a estrutura do software mais necessário se faz a centralização da especificação.

Solução
Especifique em apenas um arquivo tanto os elementos arquiteturais como a arquitetura do software em si. Inicialmente realize a especificação de todos os elementos arquiteturais presentes na arquitetura. Em seguida especifique a arquitetura através das conexões entre esses elementos.

Contexto Resultante
Tanto os elementos arquiteturais como a configuração desses elementos formando a arquitetura são especificados em apenas um documento, cabendo à equipe a tarefa de organizar o trabalho de seus participantes de forma independente em cima de uma mesma fonte de informação.

Racionalização
Especificar tanto os elementos arquiteturais como a arquitetura do software em um mesmo arquivo texto facilita a compreensão da especificação e a visualização de inconsistências, o que pode agilizar a tarefa de especificação.

Usos Conhecidos
A organização de especificações arquiteturais em um arquivo único é um consenso entre as ADLs. A ADL UniCon, por exemplo, utiliza essa estrutura para organizar suas especificações. A Figura 2, apresenta uma especificação arquitetural escrita em UniCon. Como pode ser observado, tanto os elementos arquiteturais como a configuração desses elementos são definidos em apenas um documento.

```
COMPONENT Reverser {
   INTERFACE IS
      TYPE Filter
      PLAYER input IS StreamIn
         SIGNATURE (‘‘line’’)
         PORTBINDING (stdin)
      END input
      PLAYER output IS StreamOut
         SIGNATURE (‘‘line’’)
         PORTBINDING (stdout)
      END output
      PLAYER error IS StreamOut
         SIGNATURE (‘‘line’’)
         PORTBINDING (stderr)
      END error
   END INTERFACE

   IMPLEMENTATION IS
      ...
      USES stack INTERFACE Stack
      ...
      CONNECT reverse.iob TO datause.user
      ...
      ESTABLISH C-proc-call WITH
         reverse.stack init AS caller
         stack.stack init AS definer
      END C-proc-call
      ...
      BIND output TO ABSTRACTION
         MAPSTO (reverse.fprintf)
      END output
   END IMPLEMENTATION
END Reverser
```

Figura 2: Organização de Especificação Arquitetural em UniCon.

Variação
Elementos Arquiteturais em Arquivos Separados
Uma variação para esse padrão é a especificação separada dos elementos arquiteturias. Isso permite a construção de especificações arquiteturais de forma distribuída, além de possibilitar o reuso dessas especificações. Entretanto, a tarefa de gerenciamento da construção da arquitetura torna-se bem mais complexa.

Um exemplo da aplicação dessa variação é a linguagem ArchML, onde os elementos arquiteturais são especificados em arquivos separados e a arquitetura completa é formada pela especificação da configuração em um outro arquivo específico. A Figura 3 apresenta o projeto de uma arquitetura em ArchML. Nessa figura, pode-se observar a definição de links entre instâncias de componentes e conectores sem a presença da especificação dos mesmos, que são especificados em arquivos externos, independentes e possivelmente distribuídos.

```
<?xml version="1.0"encoding="US-ASCII"?>
<!DOCTYPE System SYSTEM "../dtds/1.2/System.dtd≫
<System id="System≫
<Info>
   <Author>Cidcley T. de Souza</Author>
   <Version>1.0</Version>
</Info>
<Components>
   <Component id="CatFile"xlink:href="CatFile.xml"/>
   <Component id="RemoveVowels"xlink:href="RemoveVowels.xml"/>
</Components>
<Connectors>
   <Connector id="UnixPipe"xlink:href="Unix-pipe.xml"/>
</Connectors>
<Links>
   <Link>
      <From><Instance xlink:href="CatFile"PortName="output"/></From>
      <To><Instance xlink:href="UnixPipe"PortName="source"/></To>
   </Link>
   <Link>
      <From><Instance xlink:href="UnixPipe"PortName="sink"/></From>
      <To><Instance xlink:href="RemoveVowels"PortName="input"/></To>
   </Link>
</Links>
</System>
```

Figura 3: Organização de Especificação Arquitetural em ArchML.

Consequências
Algumas das vantagens do padrão Organização das Especificações são as seguintes:

- **Fácil compreensão.** Tendo centralizado a especificação de todos os elementos em um arquivo único, fica mais fácil se observar e manipular as características da arquitetura.

- **Facilidade de Evolução.** Qualquer modificação que seja necessária, pode ser melhor planejada se observando as características dos elementos arquiteturais individualmente.

Algumas das desvantagens do padrão Organização das Especificações são as seguintes:

- **Baixo reuso.** O reuso de elementos arquiteturais fica comprometido com a especificação dos mesmos dentro de uma especificação arquitetural única. Sendo que para que um elemento seja reusado, este deve ser copiado para dentro da nova especificação.
- **Falta de Independência.** A construção da arquitetura deverá ser realizada por apenas uma pessoa por vez, visto que só existe um único arquivo onde todos os elementos arquiteturais são especificados. Isso compromete a evolução dos elementos de forma independente da arquitetura, cabendo a quem for modificar algum elemento, realizar essa mudança dentro do código da arquitetura.

Padrões Relacionados
Pode-se utilizar o padrão *Software Architecture* [Mes97] para guiar a definição de instâncias de elementos arquiteturais para um determinado projeto.

5 Informações dos Elementos Arquiteturais

Contexto
Você tem que especificar os elementos arquiteturais em uma arquitetura de software. Cada elemento possui, além de um tipo específico (identificado no padrão *Elementos Arquiteturais Básicos*), diversos requisitos diferentes a serem representados, sendo que cada requisito pode necessitar de níveis de detalhamento diferentes.

Problema
Quais informações deverão estar disponíveis na ADL para a especificação de cada elemento arquitetural ?

Forças

- Especificar todos os requisitos dos elementos arquiteturais pode tornar a especificação complexa demais e desnecessariamente detalhada.
- Especificar poucos requisitos dos elementos arquiteturais pode tornar a especificação pouco expressiva.
- A quantidade de requisitos a serem representados é controlada pelas necessidades de cada projeto. Sendo que quanto mais complexas forem as análises requeridas por uma determinada arquitetura, maior será a quantidade de informações necessárias para realizar essas análises.

Solução
Desenvolva a ADL de forma a deixar o projetista livre para decidir quais informações representar em cada elemento arquitetural. Entretanto, identifique algumas informações que devem ser obrigatórias para cada elemento e torne todas as demais opcionais.

As informações obrigatórias são as necessárias para identificar as características básicas de cada elemento arquitetural. Já as informações opcionais são as que têm sua utilização definida de acordo com as necessidades de cada projeto.

Para organizar a representação dessas informações de forma a permitir um melhor entendimento das especificações, o padrão *Estrutura Hierárquica de Informações* deve ser utilizado.

Contexto Resultante
Os tipos de informações necessárias para capturar os requisitos dos elementos arquiteturais estão definidos, podendo esses serem representadas pela ADL. Entretanto, além de se definir quais informações são relevantes para se especificar um determinado elemento arquitetural, essas informações devem estar organizadas para que sejam mais facilmente entendidas e manipuladas.

Exemplo
A equipe de desenvolvimento deseja especificar um componente para uma arquitetura de software. Os requisitos levantados para esse componente indicam que ele possui as seguintes características:

- Representa um fornecedor de serviços (servidor);
- Aceita um número limitado de conexões simultâneas;

Baseado nesses requisitos podem ser observados dois tipos diferentes de informações a serem representadas pela ADL para se especificar esse componente. O primeiro tipo de informação diz respeito a identificação do componente em si, como por exemplo a utilização de um identificador único para representar o componente. Esse tipo de informação é considerada uma informação básica para o componente, e deve ser obrigatória para quaisquer outros componentes especificados pela ADL.

Um outro tipo de informação diz respeito às características do componente que podem variar de acordo com as necessidades de cada projeto. Por exemplo, o número máximo de conexões aceito pelo componente pode ser uma restrição referente às limitações do sistema de comunicação onde esse componente está inserido, sendo que essa informação pode não ser relevante se a aplicação especificada estiver totalmente centralizada em uma máquina apenas. Desse modo, a ADL deve representar essas informações de forma opcional, ou seja, devem haver elementos sintáticos que possam ser utilizados para representar essa informação sem se ter a obrigação de representá-las.

Usos Conhecidos
Todas as ADLs possuem mecanismos para a identificação de elementos arquiteturais. Seja através de um nome, ou por uma representação simbólica. Contudo, a representação de outros tipos de informações contextuais sobre os elementos não é uma característica comum a essas linguagens.

Entretanto, existem algumas exceções. A linguagem Acme, por exemplo, permite a representação de propriedades através de anotações dentro da especificação do elemento arquitetural. A Figura 4, apresenta a especificação de algumas propriedades de um componente em Acme. Uma peculiaridade de Acme é que essas propriedades são tratadas apenas como anotações, e são manipuladas apenas por ferramentas específicas.

```
Component server = {
   Port receive-request;
   Properties { idempotence :  boolean = true;
                max-concurrent-clients :  integer = 1 }}
```

Figura 4: Propriedades em Acme.

A definição de informações em elementos arquiteturais também pode ser observada na linguagem ArchML. A Figura 5 apresenta a especificação de um componente onde as informações obrigatórias identificadas são o nome do componente e a interface que ele utiliza.

Como informações opcionais podem ser definidas: restrições de instalação do componente, algumas propriedades do componentes, entre outras. Na Figura 5, uma propriedade *Property* representa o nome de um arquivo que implementa um determinado componente.

```
<?xml version="1.0"?>
<!DOCTYPE ComponentType SYSTEM "../dtds/1.2/ComponentType.dtd≫
<ComponentType id="CatFile≫
<Info>
   <Author>Cidcley T. de Souza</Author>
   <Version>1.0</Version>
</Info>
<Properties>
   <Property>
      <Name>Implementation</Name>
      <VarType Value="string≫</VarType>
      <DefaultValue>catfile.java</DefaultValue>
   </Property>
</Properties>
<Interfaces>
   <Interface xlink:href="FilterItf.xml≫</Interface>
</Interfaces>
</ComponentType>
```

Figura 5: Especificação de informações em ArchML.

Uma característica de ArchML é que essa linguagem permite a definição de quantas propriedades sejam necessárias, sendo que essas informações são todas utilizadas pelas ferramentas que utilizam a linguagem.

Conseqüências
Uma das vantagens do padrão Informações dos Elementos Arquiteturais é a seguinte:

- **Flexibilidade.** A quantidade de informações a ser utilizada na descrição de cada elemento é definida por quem está fazendo a especificação, de acordo com as necessidades do projeto.

Uma desvantagem do padrão Informações dos Elementos Arquiteturais é a seguinte:

- **ADL mais complexa.** A implementação da ADL para dar suporte a essa característica é um pouco mais difícil, visto que não se pode definir a priori quais elementos devem ser inseridos para a descrição de cada elemento arquitetural.

Padrões Relacionados
Tendo definido a forma com que a ADL deve permitir a especificação das informações dos elementos arquiteturais, o passo seguinte é se definir como a ADL realizará a organização dessas informações de forma a se conseguir um melhor entendimento e manipulação das mesmas. Para esse fim o padrão *Estrutura Hierárquica de Informações* deve ser utilizado.

6 Estrutura Hierárquica de Informações

Contexto
Você está especificando um elemento arquitetural para um software complexo. Esse elemento possui uma grande quantidade de informações para ser representada. De acordo com os requisitos do projeto em que esse elemento está inserido, há a necessidade de se representar todas as informações levantadas (o padrão *Informações dos Elementos Arquiteturais* mostra como a ADL deve representar essas informações). A especificação desse componente deve ser utilizada pelos outros participantes no processo de desenvolvimento do sistema.

Problema
Como a ADL deve representar as informações dos elementos arquiteturais de forma a facilitar o entendimento dessas informações ?

Forças

- Um elemento arquitetural sem uma estrutura bem definida para representar suas informações é difícil de ser entendido e, conseqüentemente, difícil de ser mantido.
- A mistura de informações dentro da especificação de um elemento arquitetural dificulta bastante a realização de análises sobre o elemento. Desse modo, quanto mais fácil for reconhecer as informações dentro da especificação dos elementos, mais fácil será sua manipulação.
- A criação de hierarquias para representar as informações dos elementos arquiteturais pode facilitar o entendimento da relação entre as informações. Entretando, se a profundidade dessa hierarquia for muito grande as informações não ficarão claras.

Solução
Crie um conjunto de contextos para cada informação e, dentro de cada contexto, defina de forma hierárquica as informações pertinentes ao contexto. Cada contexto deve servir como um delimitador de informações de um determinado tipo. Essas informações, por sua vez, devem ser organizadas de forma hierárquica dentro desses contextos.

Contexto Resultante
Com a aplicação desse padrão as informações necessárias para representar os elementos arquiteturais são organizadas contextualmente. Sendo que cada contexto representa um delimitador para tipos de informações diferentes.

Racionalização
Com a definição de contextos para organizar a representação de informações sobre elementos arquiteturais se diminui a profundidade da hierarquia de um determinado tipo de informação. Com isso, conseguimos todas as vantagens da representação de informações de forma hierárquica minimizando os riscos de tornar a especificação confusa.

Além disso, os contextos permitem se definir conjuntos de informações específicas, o que ajuda bastante a manutenção e a realização de análises sobre a arquitetura.

Exemplo
Suponha que se queira representar as seguintes informações sobre um componente de uma arquitetura de software.

1. Esse componente foi implementado por Cidcley;
2. Esse componente foi implementado no dia X;
3. O componente foi implementado em Java;
4. O arquivo que implementa o componente chama-se `Comp.class`
5. O PATH onde o arquivo deve ser instalado deve ser `/temp/Componentes`
6. Esse componente utiliza a Interface `Itf1`;

Para que todas essas informações possam ser mais facilmente representadas na especificação do componente de software através da ADL, elas devem ser separadas por tipos (como apresentado no padrão *Informações dos Elementos Arquiteturais*. Nessa caso, podem ser identificados três tipos distintos de informações: o primeiro tipo (itens 1 e 2) diz respeito ao gerenciamento do componente. O segundo tipo (itens 3, 4 e 5) são propriedades sobre a implementação do mesmo. O terceiro tipo (item 6) é uma informação sobre as interfaces utilizadas pelo componente.

Desse modo, podemos observar a existência de diferentes contextos relacionados às informações do componente. Assim, poderíamos definir os contextos `Gerencial`, `Propriedades` e `Interfaces`, onde cada tipo de informação, de acordo com esse exemplo, poderia ser respectivamente especificada.

Usos Conhecidos
A ADL MetaH possui construtores que permitem a definição de contextos para certas propriedades da aplicação. Essas propriedades são fornecidas através de atributos. Entretanto, a definição de informações são limitadas a escalonabilidade, confiabilidade e segurança. A Figura 6 apresenta a criação de contextos em MetaH para representar atributos de processos.

```
periodic process implementation P1.SIMPLE is attributes
   self'SourceTime := 100 μs;
   self'Period := 1 sec;
   self'SourceFile := "p1.a";
end P1.SIMPLE;
periodic process implementation P2.SIMPLE is attributes
   self'SourceTime := 50 μs;
   self'Period := 1 sec;
   self'SourceFile := "p1.a";
end P2.SIMPLE;
```

Figura 6: Contextos em MetaH.

ArchML por sua vez, permite a definição de diversas propriedades, além de possuir contextos específicos para definir informações gerenciais, de interfaces e de links entre elementos arquiteturais.

A Figura 7 apresenta trechos de códigos de especificação de contextos em ArchML com suas informações devidas. Nessa figura são apresentados alguns exemplos de contextos e de informações sobre esses contextos. Nela podem ser observados os contextos `Info`, `Properties` e `Links`. Alguns desses contextos podem ter outros contextos mais específicos, como e o caso de `Link`, que é um subcontexto de `Links`. Dentro de cada contexto ou subcontexto são armazenadas as informações sobre eles. Por exemplo, `Author` e `Version` são informações sobre o contexto `Info`.

Conseqüências
Algumas das vantagens do padrão Estrutura Hierárquica de Informações são as seguintes:

- **Legibilidade.** Os contextos criados permitem uma melhor visualização das informações representadas.
- **Manutenibilidade.** A adoção de um esquema hierárquico facilita a representação de dependências sobre informações e possibilita uma melhor manutenção das especificações.

Uma desvantagem do padrão Estrutura Hierárquica de Informações é a seguinte:

- **Classificação de informações.** Algumas vezes pode ser bem difícil se definir onde se colocar uma determinada informação.

```
<Info>
   <Author>Cidcley T. de Souza</Author>
   <Version>1.0</Version>
</Info>
   ...
<Properties>
   <Property>
      <Name>Implementation</Name>
      <VarType Value="string≫</VarType>
      <DefaultValue>remove.java</DefaultValue>
   </Property>
</Properties>
   ...
<Links>
   <Link>
      <From><Instance xlink:href="CatFile"PortName="output"/></From>
      <To><Instance xlink:href="UnixPipe"PortName="source"/></To>
   </Link>
</Links>
```

Figura 7: Contextos e hierarquias em ArchML.

7 Resumo da Linguagem de Padrões

Os padrões apresentados nesse artigo apóiam a tomada de decisões sobre o projeto sintático de ADLs. Com eles, os projetistas de ADLs podem construir linguagens mais fáceis de serem manipuladas e, principalmente, sintaticamente fáceis de serem utilizadas. Iniciando com o padrão *Projetar ADL*, que define a necessidade de se projetar uma nova ADL para a realização de tarefas de projetos arquiteturais específicos de uma equipe de desenvolvimento, todos os padrões tratam de como se definir os elementos sintáticos necessários para a representação de projetos arquiteturais.

Nesse contexto, o padrão *Elementos Arquiteturais Básicos* é apresentado para auxiliar a definição dos tipos de elementos arquiteturais a serem especificados por uma ADL de forma a se conseguir uma melhor reusabilidade desses elementos. Já o padrão *Organização das Especificações* trata da forma com que a ADL deve organizar os elementos arquiteturais de uma arquitetura para gerar especificações de fácil manutenção e evolução. Além disso, o problema da determinação dos tipos de informações que devem ser representadas para cada elemento especificado por uma ADL é tratado pelo padrão *Informações dos Elementos Arquiteturais*, que objetiva a criação de especificações flexíveis, no sentido de que novas informações possam ser facilmente introduzidas sem ter que modificar a ADL. Por fim, é definido o padrão *Estrutura Hierárquica de Informações* que especifica a forma pela qual as informações dos elementos arquiteturais devem ser organizadas pela ADL, de modo a aumentar tanto a legibilidade das especificações como facilitar a manutenção das mesmas.

Agradecimentos

Os autores gostariam de agradecer à CAPES (*Fundação Coordenação de Aperfeiçoamento de Pessoal de Nível Superior*) pelo suporte financeiro concedido. Gostaríamos também de agradecer ao nosso "shepherd", Alexandre M. Braga, que com seus excelentes comentários e sugestões contribuiu diretamente para o aperfeiçoamento desse trabalho, e aos colegas: Rossana, Jugurta, Flávia, Gibeon e Márcio, pelas valiosas sugestões e análises realizadas durante o SugarLoafPloP.

Referências

[AG97] R. Allen and D. Garlan, *A Formal Basis for Architectural Connections*, IEEE Transactions on Software Engineering (1997).

[BCK97] Bass, Clements, and Kazman, *Software Architecture in Practice*, Addison-Wesley, 1997.

[BV93] P. Binns and S. Vestal, *Formal Real-Time Architecture Specification and Analysis*, IEEE Workshop on Real-Time Operating Systems and Software, 1993.

[DHKM97] Dikel David, Christy Hermansen, David Kane, and Raphael Malveaux, *Organizational Patterns for Software Architecture*, In Proceedings of PloP97, 1997.

[GAO94] D. Garlan, R. Allen, and J. Ockerbloom, *Exploiting Style in Architectural Design Environments*, In Proceedings of ACM SIGSOFT: The Second Symposium on Foundations of Software Engineering, 1994.

[Gar00] David Garlan, *Software Architecture: a Roadmap*, The Future of Software Engineering. In Proceedings 22nd International Conference on Software Engineering (ACM Press, ed.), 2000.

[GMW97] D. Garlan, R. T. Monroe, and D. Wile, *Acme: An Architectural Description Interchange Language*, In Proceedings of CASCON'97, 1997.

[GP95] D. Garlan and D. Perry, *Introduction to the Special Issue on Software Architecture*, IEEE Transactions on Software Engineering (1995).

[LAK+95] D. C. Luckham, L. M. Augustin, J. J. Kenny, J. Veera, D. Bryan, and W. Mann, *Specification and Analysis of System Architecture Using Rapide*, IEEE Transactions on Software Engineering, 1995.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, *Specifying Distributed Software Architectures*, Proceedings of the 5th European Software Engineering Conference, 1995.

[Mes97] Gerard Meszaros, *Archi-Patterns - A Process Pattern Language for Defining Architectures*, In Proceedings of PloP97, 1997.

[MORT96] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor, *Using Object-Oriented Typing to Support Architectural Design in the C2 Style*, In Proceedings of ACM SIGSOFT (Symposium on Foundations of Software Engineering), 1996.

[MQ94] M. Moriconi and X. Qian, *Correctness and Composition of Software Architectures*, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, 1994.

[SC01] Cidcley T. de Souza and Paulo R. F. Cunha, *Especificando Arquiteturas de Software em XML*, XXVII Conferência Latinoamericana de Informática, 2001.

[SDD95] M. Shaw, R. DeLine, and D.V.Klein, *Abstractions for Software Architecture and Tools for Support Them*, IEEE Trans on Software Engineering, 1995.

[Sha96] Mary Shaw, *Some Patterns for Software Architectures*, Pattern Languages of Program Design 2 (Addison-Wesley, ed.), 1996.

A Arquitetura de Software e ADLs

O termo "Arquitetura de Software" não possui uma definição que seja universalmente aceita pela comunidade de Engenharia de Software. Na realidade, existem diversas definições para essa disciplina, cada uma salientando alguma característica importante. De uma forma simplificada, por exemplo, Garlan [GP95] define arquitetura de software como sendo "a estrutura dos componentes de um programa/sistema, os inter-relacionamentos, princípios e regras que governam seu projeto e evolução ao longo do tempo". A definição de Bass [BCK97] diz que "a arquitetura de um programa ou sistema de computação é a estrutura ou estruturas de sistema que compreendem os componentes de software, as propriedades externamente visíveis desses componentes, e o relacionamento entre eles".

A despeito da falta de consenso na definição do termo *arquitetura de software*, a terminologia utilizada nessa área de pesquisa e a identificação dos elementos básicos na descrição dessas arquiteturas são bastante aceitas. A definição encontrada em Moriconi [MQ94] apresenta os conceitos representados por arquiteturas de software, considerando os seguintes elementos: Componentes, que representam objetos com existência independente; Interfaces, objetos tipados que são pontos lógicos de interação entre os componentes e seu ambiente; Conectores: objetos tipados que relacionam interfaces, componentes ou ambos; Configurações: uma coleção de restrições que une objetos em uma arquitetura específica.

Todavia, mesmo com todo o progresso alcançado até agora, a área de arquitetura de software ainda não atingiu sua maturidade como uma disciplina de engenharia [Gar00]. Embora seus conceitos básicos já estejam claros, ainda existem enormes desafios a serem vencidos. Contudo, é indiscutível a importância das arquiteturas de software no contexto do ciclo de vida de softwares complexos.

A.1 A Importância do Projeto Arquitetural

O processo de comunicação entre os participantes de um projeto de software é essencial para o sucesso do mesmo. Porém, esse processo é bastante difícil de ser coordenado, visto que diversos interesses devem ser considerados e acomodados de forma consistente. Gerentes, agências de financiamento, programadores, entre outros, possuem visões diferentes do mesmo sistema. Entretanto, é importante que essas pessoas possam comunicar suas necessidades, e que essas necessidades possam ser entendidas pelos outros. Um gerente financeiro, por exemplo, que possui um grande entendimento sobre o domínio de um problema a ser implementado, mas não entende nada de programação, deve ser capaz de se comunicar com o desenvolvedor, que entende de programação, mas pode não ententer nada sobre o domínio do problema.

O projeto arquitetural fornece essa ponte entre os requisitos do sistema e a implementação [Gar00]. De forma simplificada, um diagrama contendo as partes do software representadas por caixas e linhas, fornece uma descrição da funcionalidade do sistema, além de permitir a visualização das interconexões entre essas partes. Essa representação fornece uma forma de comunicação que permite que o gerente possa ver seus requisitos representados e que o programador possa verificar as diferentes formas nas quais as funcionalidades possam ser implementadas.

A.2 Linguagens de Descrição de Arquitetura

Durante muito tempo, os projetos arquiteturais foram tratados informalmente. Simples diagramas com descrições em linguagem natural eram utilizados para definir a funcionalidade de aplicações complexas. Contudo, essas notações eram imprecisas e ambíguas, e não permitiam a realização de análises mais complexas sobre o comportamento das aplicações.

Em resposta à informalidade dos projetos arquiteturais, surgiram as ADLs (*Architecture Description Languages*). As ADLs são linguagens utilizadas na descrição precisa de arquiteturas de software, possibilitando a realização de análises formais dessas arquiteturas. Atualmente é possível se encontrar um grande número de ADLs, algumas mais genéricas e outras desenvolvidas para domínios específicos.

Exemplos de ADLs incluem Aesop [GAO94], Darwin [MDEK95], C2 [MORT96], Rapide [LAK+95], UniCon [SDD95], Meta-H [BV93], Wright [AG97], entre outras. Cada uma dessas ADLs, fornece um conjunto de propriedades diferentes. Por exemplo, Aesop suporta a utilização de estilos arquiteturais; Meta-H é utilizado para o projeto de softwares de tempo-real para controle de aviação; UniCon suporta tipos heterogêneos de componentes e conectores.

Além de possuírem característica específicas, as ADLs também possuem sintaxes específicas, normalmente difíceis de serem entendidas pela grande maioria dos participantes do processo de desenvolvimento de software. Essa dificuldade no entendimento dos elementos arquiteturais descritos pelas ADLs foi a principal motivação do desenvolvimento da linguagem ArchML.

Design Patterns for Components Reuse

Eduardo Kroth
University of Santa Cruz do Sul - UNISC
Department of Computer Science
kroth@polaris.unisc.br

Carlos Alberto Heuser
Federal University of Rio Grande do Sul – UFRGS
Department of Computer Science
heuser@inf.ufrgs.br

Abstract

This paper addresses the software reuse, specially the problem of developing applications from pre-existing software components. These components have concrete and template methods, which offer a specific functionality to be used in the application development. The "use" contract establishes the rules for the use of component methods by application methods. The "implementation" contract establishes the rules for the implementation of abstract methods of a component. In order to allow the implementation of the contracts, integration architecture is proposed, formed by components, application classes and integration classes. The integration classes are specified as design patterns. This paper describes the development of applications that use components and shows the way the integration architecture must be constructed.

Keywords: software reuse, components, frameworks, design patterns, reuse contracts

1. Introduction

One of the software reuse techniques concerning a specific problem domain is referred to the implementation of common functionality in frameworks [JOH88]. A *framework* for building applications in a specific problem domain is called vertical framework in opposite to a *horizontal framework* which provides a common infrastructure for lots of problem domains (visual interface, communication, persistence,...) [FAY97]. The problem related to frameworks use, especially with the use of vertical frameworks is that they implement a whole problem domain. Because of this, vertical frameworks are generally big, complex and difficult to understand and use. To solve this problem, a variety of solutions has been proposed, as the use of visualization graph tools [MEU97], the contract use [COD97] or design patterns use [ODE97]. Framelets are little frameworks that resolve simple cases and they have been coded in components [PRE00].

This paper proposes the framework division on a set of mini-frameworks as an alternative, i. e.; each framework consists of a little set of interrelation classes. These frameworks can be treated as a software component [SZY98].

Using a three-layer software architecture (human-computer interface problem domain and data management), this paper is based on the problem domain layer. In its proposal, the domain problem is divided into three different layers concerning the interests of them [SIL96]. From the three layers, one is generic formed by components that offer solutions for little specific problems.

The other layer is composed by whatever component application. The middle layer makes the relations that exist among components and application classes.

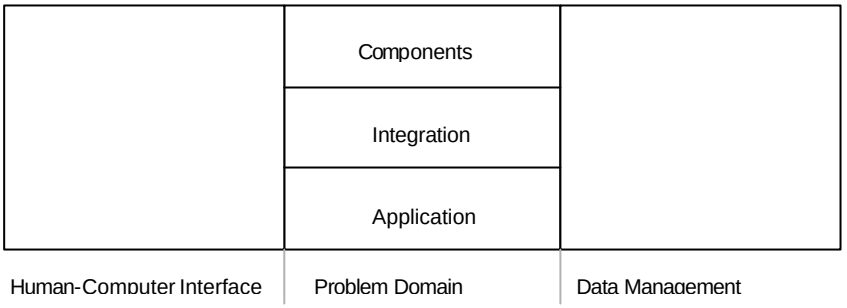


Figure 1 – Three-Layer Software Architecture

This paper presents a software architecture that allows associating the application classes to a component set as the one described in Figure 1. Besides that, the work describes a wizard, which allows building integration layer classes in an automatic way.

This paper is organized as it follows. Section 2 describes relations that there are among application and components. Section 3 presents the reuse contracts notation as components and application representation. In this chapter it is presented two new types of contracts. These contracts receive properties that increase the generation of an *integration architecture* developed in section 4.

2. Relations Among Components and Application

The following example describes possible relations among components and application. Figure 2 presents the component and application classes for a travel agency. The components are based on a set of analysis patterns described in [COA 97]. The components have abstract, template and concrete methods. Abstract methods are identified by italic form**.** Template and concrete methods are available for the application developer to relation them with application methods that have similar functionality. In Figure 2, components are inside packages and each package name identifies a component.

The application classes presented in Figure 2 are about a travel agency system. The application developer, actor responsible for the definition of components and application relations, develops these classes. An application can be built through the combination of a lot of components. The relation of an application and its components is characterized as it follows:

?? an application class may not need all class component methods,
?? an application class can use or implement methods from different components,
?? an abstract component method can be implemented by a concrete application method or by other concrete component method, and
?? the application methods can have different names from the ones presented in the components.

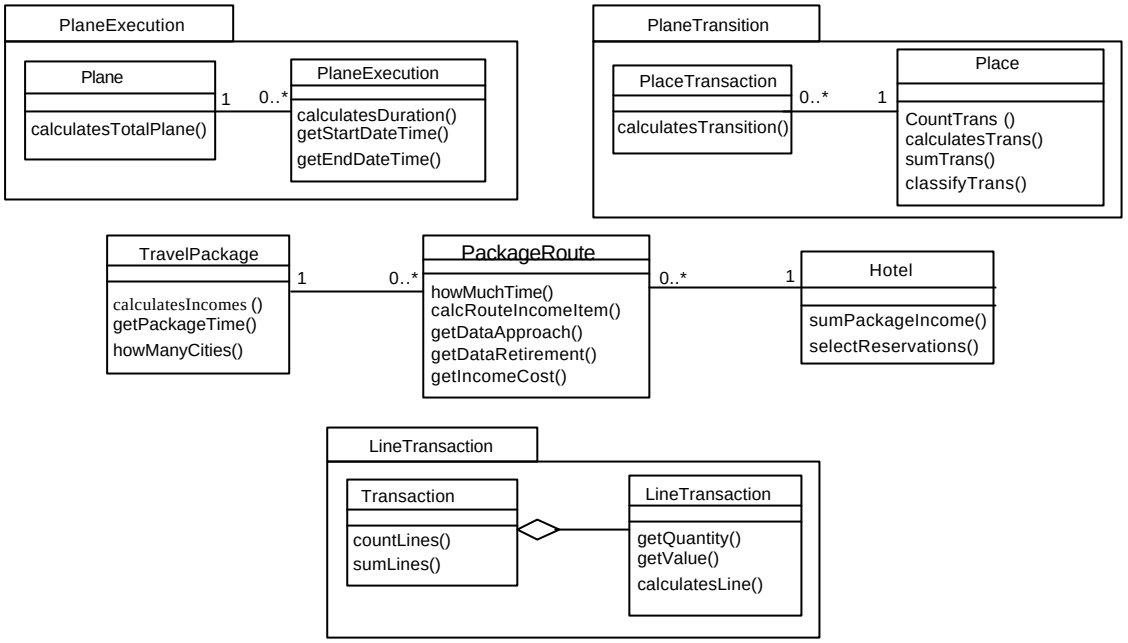


Figure 2 – Components and application classes example

Using the object model of Figure 2 some relations can be estabilished among the components and the application. The application developer establishes that the TravelPackage.calculatesIncome() method has the same functionality of the Transaction.addLines() method as the PackageRoute.howMuchTime() method in relation to PlaneExecution.calculatesDuration() method. These method relations are called use relation i.e., the TravelPackage.calculatesIncome() method uses the Transaction.addLines() method.

Abstract methods referred as template methods that have been chosen by the application developer need to be implemented. Thus, the application method or the method of other component that implements each referred method needs to be identified.

The PackageRoute.howMuchTime method uses the PlaneExecution.calculatesDuration. The latter is a template method and it invokes two abstract methods of the same class: getStartDateTime and getEndDateTime. Therefore, these two abstract methods need to be implemented if the PlaneExecution.claculatesDuration method is going to be used. This situation defines a component method being implemented by an application method.

Besides, component methods can be used to implement abstract methods. For example PlaceTransaction.calculatesTransaction is used by a Place class template method. In the studied case, this method is implemented by other component method: LineTransaction.calculatesLine.

This relation among methods is called *implementation* relation. An application method class or other component method can be implemented by an abstract component method.

3. Using reuse contracts to represent the relations

This paper uses the reuse contracts notation to represent the implementation and use relations established among components and application.

Reuse contracts [MEN96] is a technique for formal representation of components and application relation. The Programming Technology Labor research team of Free Bruxelas University, Belgica [DHO98] has been developing it. The notation was created in 1996 [LUC96], however it was not based in UML Recent papers [MEN98a, MEN 98b] adapted the notation to UML.

Reuse contracts use white-box components because they document the component and application classes relation. The component and application classes relation can be classified in basic types called *types of reuse contracts*. Originally the reuse contracts are: concrete and abstraction; extension and cancellation; refinement and coarsening [STE96]. In UML notation a stereotype is created for each contract and it is indicated in the dependency association.

This paper presents two new types of contracts. Each contract referred in this chapter is specified by some properties and by syntax. The syntax describes the write/read contract form. The properties relate the characteristics of each contract. The referred examples are about Figure 3.

3.1. Types of use contract

The use contract concerns the application methods that refer component methods aiming the use of component method implementation. For that, the application method functionality needs to be the same of the component method. Three properties characterize this contract and differ it of a simple inheritance: (i) possibility of component methods rename when used in application; (ii) partial use of component methods and, (iii) association of an application class with many component classes. The use contract syntax is as it follows:

Use <component method> used by <application method>

1st Property: Possibility to rename component methods . This property allows an application method name that is referring a component method, is not the same of the referred method, giving the application developer liberty in choosing another method name. This property causes an association among component and application i.e., the application method just calls the component method. For example the PlaneExecution.calculatesDuration() method is used by PackageRoute.howMuchTime() method and they do not have the same name.

2nd Property: Partial use of component methods. This property establishes liberty for the application developer in choosing the necessary component methods for the application class. Reuse contracts use the cancel contract to delete the desire method. Differing from the cancel contract, the use contract proposes information of the component methods chosen. In Figure 3, the Hotel application class does not use all the Place component class methods.

3rd Property: Association of an application class with many component classes. In an application class developing, the application developer can have the necessity of choosing more than one component to associate with an application. This necessity occurs when the application methods implementation are in different components. So, this property allows the relationship of an application class with many components. Besides, this property is necessary because the class developing solution is not stored in only one component. The property requires objects instantiation in the corresponding associated objects. For example, the application class PackageRoute has a relation with LineTransaction, PlaneExecution and PlaceTransaction component classes.

3.2. Types of implementation contract

The implementation contract concerns the abstract component methods implementation. This contract is complicated when a template component method is referred and consequently the abstract methods of the dependency list need to be implemented. The application developer needs to provide the abstract method implementation that can be in an application method or in another component method.
The syntax contract is: Impl <component method> - <application method>,
read as <component method> is implemented by <application method>.

The implementation contract presents two properties: (i) reference of an abstract method to a concrete method and (ii) possibility of different names among concrete and abstract methods.

1st Property: Reference of an abstract method to a concrete method. The implementation of an abstract component method has two options: (i) to use an application method or (ii) to use another component method. The first property says that the implementation relation needs to indicate which concrete method provides an implementation for abstract component method. For example, Figure 3 shows the PlaneExecution.getStartDateTime() needing an implementation and the application developer specifying the PackageRoute.getDateApproach() for that.

2nd Property: Possibility of different names among concrete and abstract method. The property determines that the implementation and component methods do not need to have the same names. The above example illustrates this property. When the application developer chooses another component to use in the implementation of a method, it is necessary to observe if the functionality specifications are enough.

In the example of Figure 3, the application developer defines that the PlaneExecution abstract methods class uses the PackageRoute application class implementation. The PlaneExecution abstract methods class is required by the PlaneExecution.calculatesTransaction method and they are methods that return needed information attributes for the called method.

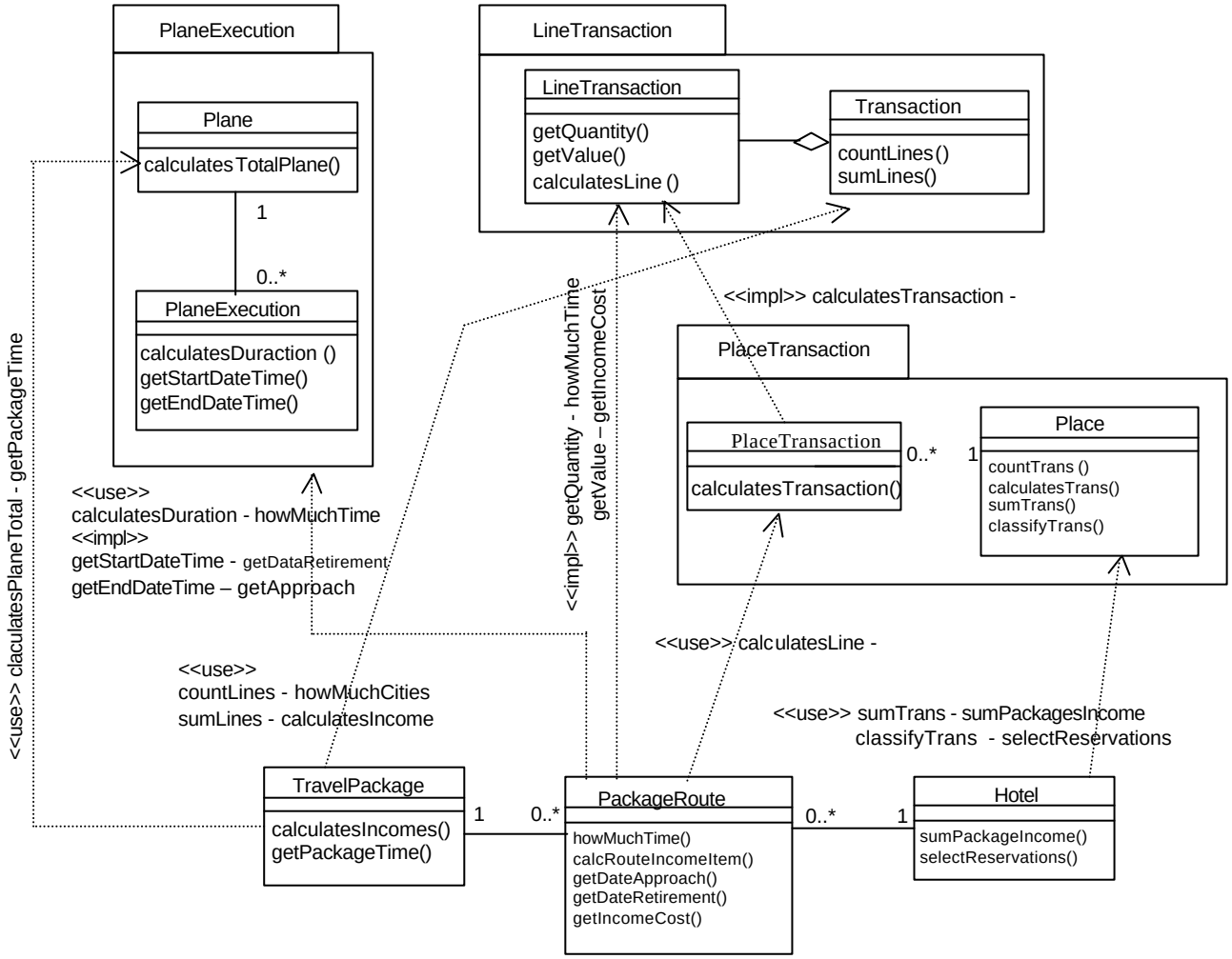


Figure 3 – Application of the contracts in the Travel system

4. Integration Architecture

This chapter presents classes architecture for the reuse contracts implementation proposed in the last chapter. The integration architecture is important because the Reuse Contracts notation is just used by the objects diagram i.e. not being directly implemented in programming languages. The new contract implementation is presented in the following chapter as design patterns.

4.1. Use Contract Pattern

Context

The use contract between component and application methods presents properties that require implementation. The use contract claims that the application class that uses this kind of contract

to establish a relation with component classes must provide some implementation functionality presented by the kind of contract.

Problems
This functionality implementation changes the original application class. In order not to change the original application class, the use contract functionalities need to be implemented in a specific built class. This class needs to be an abstract class individually built for each application class that has the use contact. The implementation of the components and application relation presents the follow requirements:

To instance and to delete the objects of associated components – all the instanced application object needs the instance of all associated components through the USE relation. When the application object is destroyed, all the objects of associated components need to be destroyed. The application object managers the 'life' of the objects of associated components and determines its instance and destruction.

In the above example, the ApplicationA application class has a relation with ComponentA, ComponentB and ComponentC classes through the use contract. This implies that every time that an ApplicationA application class object is instanced, the corresponding objects from the relating classes of components need to be instanced too.

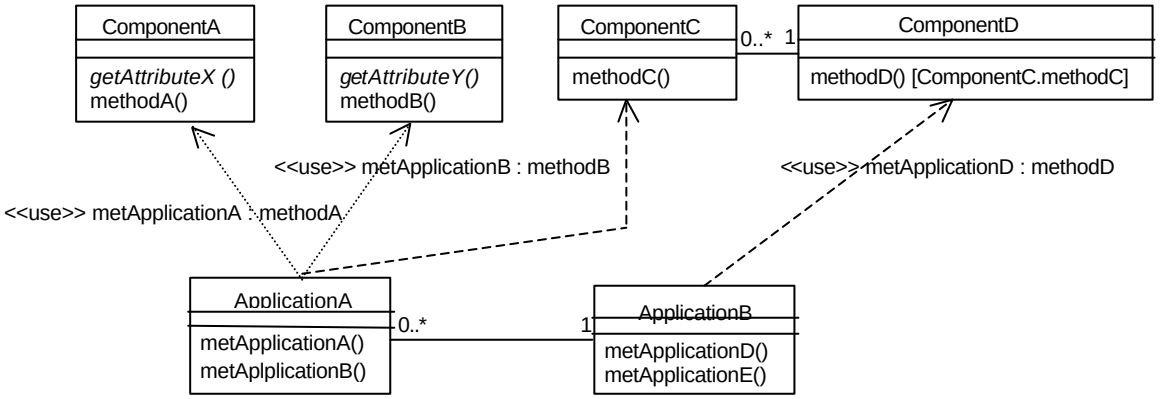


Figure 4 – Object Model using USE Contract

b) To have reference to the objects of related components – the application object needs to have a reference for each instanced object component. For example, the ApplicationA.metApplicationA must have references to ComponentA, ComponentB and ComponentC component classes.

c) To invoke the component methods – the application method indicated in the USE relation needs to be implemented to reference the corresponding method in the component object. For example, the ApplicationA.metApplictionA body method just needs a reference to invoke the ComponentA.methodA

d) To update the required application references – when a component method references another component method, the reference needs to be instanced between the classes. If the referenced class does not have an associated application class, then an application class that establishes a relation with it needs to be provided, to only instance and destroy its objects.

For example, one of the (ApplicationB.metApplicationD) methods references a component method (ComponentD.methodD). The referenced method depends on the ComponeneteC.methodC. The ComponentD.methodD method dependency concerns other method class, therefore the ComponentC class needs to be instanced and must have relation to any application class. Considering this, the application developer indicates that the AplicationA application class makes reference to the ComponentC class to just attend the dependency of the methods defined above.

Solution

Two other design patterns were used to build the router class solution: Decorator e Facade[GAM94]. The router class is an abstract class and exists for each application class that has a reuse contract with component classes. The application class inherits its router propriety class.

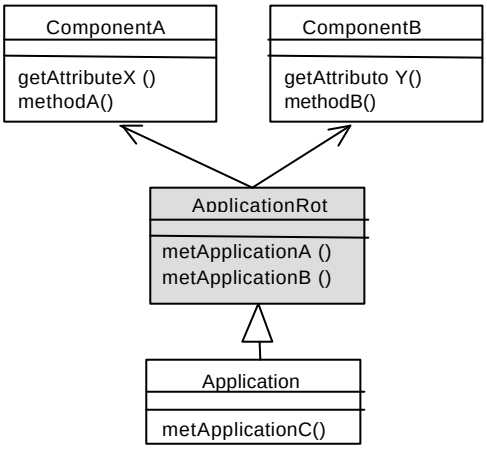


Figure 5 – Objects Model using router classes

The router class notation orders is as it follows: application class name followed by the "Rot" suffix. In Figure 5 the router class created to attend the association of Application (application) with ComponenteA and ComponentB (components) is called ApplicationRot.

A router class must have the instance and destruction functionality of the components implemented in its corresponding methods with the some functionality. Besides, the router class needs to have references for related components by the use contract kind. The methods established by the use contract kind must be in the router class, having the corresponding method call in the component.

4.2 Implementation contract pattern

Context

The implementation contract is used when abstract component methods are called by other component methods. Abstract methods need to be implemented in other specialized classes to not change the original component structure. The implementation contract indicates the concrete method for the abstract method implementation. This concrete method can be in an application class structure or in another component structure. Therefore, a component related with application classes can have abstract methods implemented in different classes, not only from the application but also from other components. This fact needs to be considered in this problem. Examples are in figure 6.

Problems

For the implementation contract implementation the following specifications are necessary:

a) to have reference to the related objects – when a component has implementation relation, the instanced object of this component must refer the objects where the related concrete methods are found. For example, the ComponentA object has an implementation relation with the ApplicationA object, therefore the component object has a reference for that application object. In the other example, the ComponentD object has an implementation relation with the object of another component, ComponentF. Therefore the ComponentD object has a reference to ComponentF object.

b) abstract methods refer the relation concrete methods – each abstract method existent in the implementation relation needs to be specialized. This specialization code must contain a reference to the concrete method defined in the relation. For example, when the ComponentA.getAttributeX method is specialized, it needs reference to the ApplicationA.getAttributeX1 method. In the other example, the ComponentD.methodD method refers to the ConponentF.methodF method.

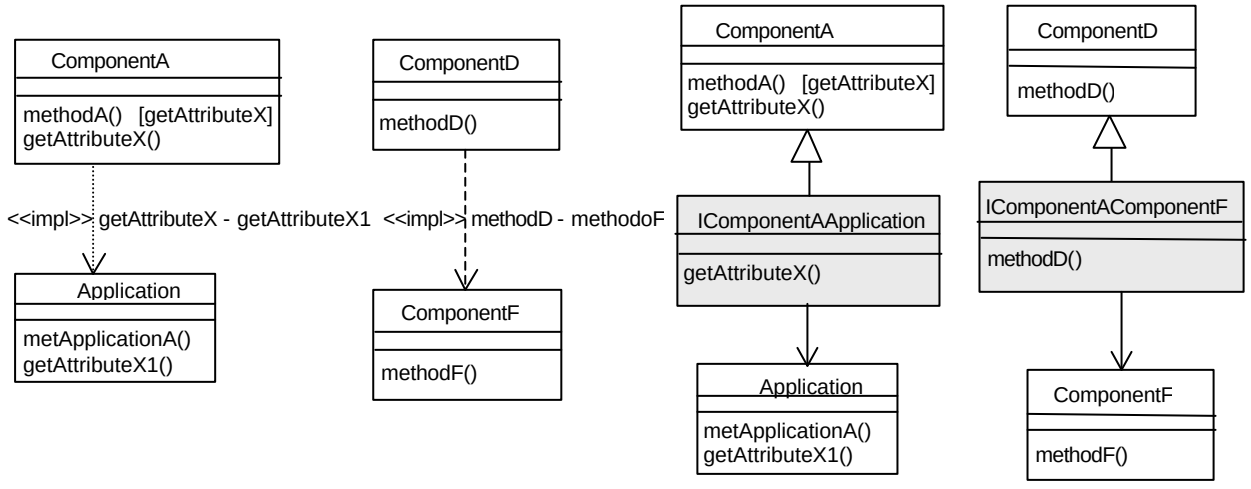


Figure 6 – Implementation relation examples and objects models using implemented classes

Solution

The implementation contract has its implementation based on the Adapter design pattern [GAM94]. For each implementation contract that associates a component class and a class responsible for the implementation, an implemented class is generated. The implementation class is resultant from this paper and it takes place in the model of the objects as a subclass of the component class. An implementation class name is composed by the suffix I (of implementation) added by the conjunction of application and component classes name. The component class is specialized by the implemented class, i.e. the instanced object in the component layer is the implemented class object.

Using Figure 6 as an example, the implementation relation establishes that ComponentA.getAttributeX is implemented by Application.getAttributeX1 method and ComponenetD.methodD method is implemented by ComponentF.methodF.

4.3 Case study using the integration architecture

Using the case study of Figure 3 – travel agency system using some pre-existent components – an object model can be generated using the integration layer to promote the implementation of use and relation implementation. For the use relation the router classes are built. In Figure 7, the implemented and router classes are edged by gray just to differ them from the others classes.

In Figure 8, the router classes are: HotelRot, PackageRouterRot and TravelPackageRot. Each router class is an abstract class and its structure is inherited by the corresponding application class. Router classes are referenced to the components established by the use relation.

Implementation classes correspond to the implementation relation established between component and application classes or from other components. Each implemented class is built for each combination of component class with its implementation class. The implemented classes built for the case study are: ILineTransactionItemPackageRoute, IPlaceTransactionItemLineTransaction and IPlaneExecutionPackageRoute.

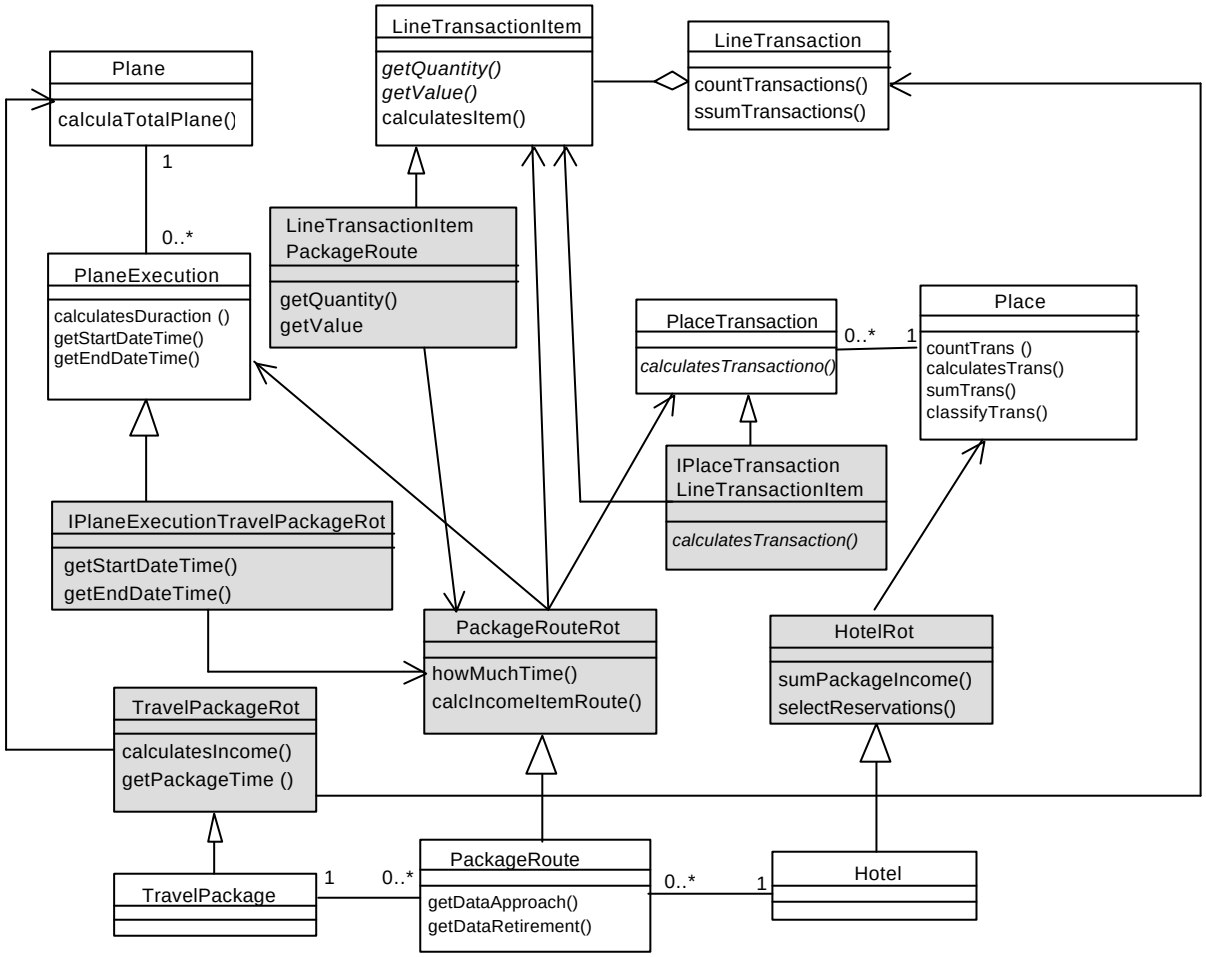


Figure 7 – Object model with the integrated architecture

6. Conclusions and Future Works

This paper presents (i) a technique aiming the specification of relation between components and application classes using a graph notation and (ii) software architecture to define relation implementation.

The technique uses software components as reuse objects. The used components are developed for the domain problem layer. These components must provide internal structure visualization and are used as white box components. Therefore components, application classes and integration layer form the problem domain layer.

When the application developer aims using component method functionality as implementation of an application class method, a use relation is used. This relation considers that not all-component methods must be used by application classes and application methods names do not have to be the same of the component. Besides that, an application class can use lots of components. Among these dependent methods, it is possible to have abstract methods requiring implementation. The application developer is responsible for abstract methods implementation and it can be done by application methods or by methods of others components. In this relation, when the abstract method is being implemented by another method, it is called implementation relation.

A notation is necessary to document the above relations. This paper uses the Reuse Contract as notation. The reuse contracts implementation is done through set of classes forming the integration layer. For each contract a specifically class is presented.

A router class is generated for each application class that has use relation with components. The router class is responsible for the object instantiation and destruction of relation components. The router class is formed by methods defined in the use relation. Each method defined in a use relation is implemented in the router class and refers to the corresponding component method. An implemented class is generated for each implementation relation existed between the component and the class that provides the method implementation. The implemented class is a component specialization and it has a reference to the implementation class provider.

For automatic classes generation of the *integration layer* this paper presents a wizard. The integration layer is generated from the relation established between components and application classes and it is specified by the reuse contracts. The wizard presents a textual interface allowing the application developer to specify the use and implementation relation between the components and application methods.

Component implementation was not evaluated in commercial tools based on component development (Microsoft/COM, OMG/Corba, and Java/RMI). Therefore, future works can concern integration and components layers implementation using those architectures.

References

[BOS97] BOSCH, J. Adapting Object-Oriented Components, Jyväskylä, Finland - Ed. Springer **Proceedings of the ...** ECOOP'97 Workshops, june 1997.

[COA97] COAD, P. et al. **Object models: strategies, patterns and applications**. New Jersey, Ed. Prentice Hall, 1997

[COD97] CODIENE, W. et al. From custom applications to domain-specific frameworks. **Communications of the ACM**, 40(10): 71-77.

[DHO98] D'HONDT, T.; et al. **Reuse Contracts**

Located in http://progwww.vub.ac.be/prog/pools/rcs/index.html (18 out 1999)

[FAY97] FAYAD, M.; SCHIMIDT, D. Object-Oriented Application Frameworks. **Communications of the ACM**. 40(10):71-77

[GAM94] GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. Massachusetts, Addison Wesley Publishing Company, 1994.

[JOH88] JOHNSON, R.; FOOTE, B. Designing Reusable Classes. **Journal of Object-Oriented Programming**, junho/julho 1988, vol. 1 number 2.p32-35

[KRO99] KROTH, E. et al. Software Assistente no Uso de Componentes. **Proceedings of the** XII Brazilian Symposium on Software Engineering – Tools Session, october, 1999.

[LUC96] LUCAS, C. **Documenting Reuse and Evolution with Reuse Contracts**. Doctoral Thesis, Science of Computing Departament, Vrije University , Bruxelas, Belgium. 1996

[MEI96] MEIJER, T. et al. Class Composition in FACE, a Framework Adaptive Composition Environment. **Proceedings of the...** ECOOP'96 Workshop Reader , july 1996.

[MEN96] MENS, K. et. al. Reuse Contracts: Managing Evolution in Adaptable Systems. **Proceedings of the...** ECOOP'96 Workshop on Adaptability in Object-Oriented Software Development, 1996

[MEN98a] MENS, K. et. al. Supporting Disciplined Reuse and Evolution of UML Models. **Proceedings of the...** UML'98 Workshop, Mulhouse, França, june 1998.

[MEN98b] MENS, K. et. al. Giving Precise Semantics to Reuse and Evolution in UML. **Proceedings of the...** ICSE'98 International Workshop on Principles of Software Evolution, Kyoto, Japan, 1998.

[MEU97] MEUSEL, M. Czarecki; Kopf, W. A model for structuring user documention of object-oriented frameworks using patterns and hypertext. **Proceedings of the ...** ECOOP'97 (LNCS 1241), pp.498-510, Springer-Verlag, 1997.

[ODE97] ODENTHAL, G.; Quiebel-Cirkel,K. Using Patterns for Design and Documentation. **Proceedings of the...** ECOOP'97 (LNCS 1241), pp. 511-529, Springer-Verlag, 1997.

[PRE00] PREE, Wolfgang; PASETTI, Alessandro. Two Novel Concepts for Systematic Product Line Development. In: The First Software Product Line Conference, 2000, Denver, Colorado. **Proceedings…** Located in: <http://www.softwareresearch.net/FrameworkMethodologyProjec/>. Acessed at 09 jul. 2000.

[SIL96] SILVA, A. R. et. al. Three-Layered Framework with Separation of Concerns. **Proceedings of the...** OOPSLA'96 Workshop on Exploration of Framework Design Principles, Califórnia, EUA, october, 1996

[STE96] Steyaert, P. et al. Reuse Contracts: Managing the Evolution of Reusable Assets. **Proceedings of the...** OOPSLA'96 Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM SIGPLAN Notices, vol. 31, nº. 10, october 1996, pp. 268-285

[SZY98] SZYPERSKI, C. **Component Software**. Harlow, Reino Unido, Addison Wesley Publishing Company,1998

Modelo de Cooperação para Aprendizagem Baseada em Projetos: Uma Linguagem de Padrões

Flávia Maria Santoro
flavia@cos.ufrj.br

COPPE-Sistemas/UFRJ
Caixa Postal 68511 - 21 945 270
Rio de Janeiro -Brasil

Marcos R. da Silva Borges
mborges@nce.ufrj.br

NCE/UFRJ
Caixa Postal 2324 - 20001-970
Rio de Janeiro - Brasil

Neide Santos
neide@ime.uerj.br

IME/UERJ
Caixa Postal 20550-013
Rio de Janeiro - Brasil

Abstract: Computer-supported collaborative learning (CSCL) environments arise as one of the most powerful and important educational applications. Moreover, CSCL holds many-sided features and building such environments is not an easy task. The developer of applications knows neither the educational domain, nor the collaborative strategies applied to the teaching-learning process. Teachers and students need a flexible environment to configure different collaborative projects. Thus, we understand that the design of CSCL environments should be based on a conceptual model, which allows the description of explicit collaborative processes. The proposal of this work is a Cooperation Model for Project-based Learning, described through a Pattern Language. The Model aims at supporting the development of collaborative environments in the domain of project-based learning.

Introdução

A construção de ambientes de aprendizagem cooperativa baseada em projetos não é uma tarefa trivial. O desenvolvedor de aplicações não conhece o domínio da educação e as nuances das estratégias cooperativas aplicadas ao processo ensino-aprendizagem, por outro lado, o professor/Facilitador e os Aprendizes precisam de um ambiente flexível, onde tenham apoio no uso da tecnologia computacional, e onde possam configurar diferentes projetos cooperativos de acordo com características específicas desejadas (Santoro et al., 2000).
A análise dos ambientes de aprendizagem cooperativa apresentados na literatura mostra que é possível identificar elementos comuns que levam a melhores ou piores resultados em termos do processo de cooperação. Esta linguagem de padrões visa apresentar alguns destes elementos e mostrar como utilizá-los para criar tais ambientes. Desta forma, o objetivo da linguagem de padrões é levantar problemas comuns aos ambientes, apontar soluções, mostrar como estas soluções são implementadas em alguns ambientes e como podem ser aplicadas no desenvolvimento de novos ambientes.

A Linguagem de Padrões

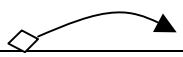
Vários aspectos estão envolvidos em um Modelo de Cooperação para Aprendizagem, e todos eles se relacionam à tentativa de produzir um processo cooperativo efetivo, traduzido em um **Fluxo Atividades**. Um processo cooperativo é definido pelo grau de **Interdependência** encontrado nas tarefas propostas.
Ao longo do processo, os aprendizes compartilham conhecimento sobre um determinado domínio. Portanto, é fundamental criar um entendimento comum sobre os objetos de estudo. Uma forma de garantir este entendimento é a estruturação ou **Representação Conhecimento**, que também irá facilitar a captura e recuperação da **Memória** do grupo.
A memória determina o armazenamento não só dos produtos gerados, mas também do desenrolar das atividades. Com isto, o grupo tem a possibilidade de aprender com através de trabalhos desenvolvidos anteriormente. Porém, surge uma questão: como e em que momento se fará o **Uso Memória**.

Mesmo em ambientes de aprendizagem cooperativa, os indivíduos necessitam de espaços para a sua produção pessoal e o ambiente deve prover **Apoio Trabalho Individual**. Porém, os membros do grupo devem sentir que o resultado faz parte de um todo através da **Integração Produtos Individuais**, afinal o resultado final é uma produção cooperativa.
Ao longo do processo, e em cada atividade especificamente, os participantes assumem funções ou responsabilidades diferentes. Estas funções são chamadas de **Papéis** e definem as relações, as formas de interação entre os participantes e o acesso a objetos compartilhados. Alguns exemplos mais típicos e genéricos são **Facilitador**, **Aprendiz** e **Coordenador**. Também é necessário definir como estes papéis são designados e mantidos através de **Critérios Nomeação**.
A **Coordenação** está relacionada ao controle do processo e ajuda ao estudante tanto em termos de conteúdo quanto de atuação no contexto das atividades propostas no ambiente. A Coordenação também envolve **Resolução Conflitos** e **Tomada Decisões**, que no caso destes ambientes devem ser analisados como processos auxiliares para que os aprendizes tomem decisões sobre o planejamento e a execução das tarefas que levarão à elaboração da solução de um problema proposto, promovendo sua aprendizagem, e mobilizam vários mecanismos cognitivos e afetivos (lógica, inferência, dedução, crença, dúvida, sutileza, envolvimento emocional).
Presentes em sistemas cooperativos de um modo geral, os mecanismos de percepção são os responsáveis pelo entendimento e consciência do grupo em relação aos participantes e às tarefas desenvolvidas. No caso específico de ambientes de aprendizagem deve-se prover **Percepção Espaço Trabalho**, **Percepção Tarefas**, **Percepção Interação Social** e **Percepção Conceitos**.
Qualquer ambiente educacional deve incorporar formas de suporte à avaliação da aprendizagem. Este suporte deve ser feito através da disponibilização de instrumentos apropriados inseridos no **Processo Avaliação Educacional**. É necessário avaliar **Resultados Individuais** e **Resultados Grupo**.

Relacionamento entre os Padrões da Linguagem

Para representar os tipos de relações entre os padrões na linguagem foi utilizado o trabalho de Gerber e Becker (2000), que propõem o uso de nós tipados para mostrar as diferenças semânticas dos diversos relacionamentos existentes entre os padrões e com isso facilitar a navegação entre eles. Os relacionamentos são classificados em quatro categorias resumidas na Tabela 1. A Figura 1 descreve o mapa de relacionamento entre os padrões da linguagem, os quais são apresentados em seguida.

Tabela 1 – Categorias de Relacionamentos entre Padrões

Relacionamento	Descrição	Representação Gráfica
é completado por	Um padrão é completado por outro quando ele divide um problema genérico em um grupo de sub-problemas resolvidos pelos padrões que o completam.	
é requisito para	Um padrão é requisito para outro quando ele tem que ser necessariamente usado antes do outro.	
leva a	Um padrão leva a outro quando ele deixa um problema não resolvido ou quando a solução aplicada gera um problema que pode ser resolvido pelo outro padrão.	
é refinado por	Um padrão é refinado por outro quando o último atende a um problema que é uma especialização do primeiro.	

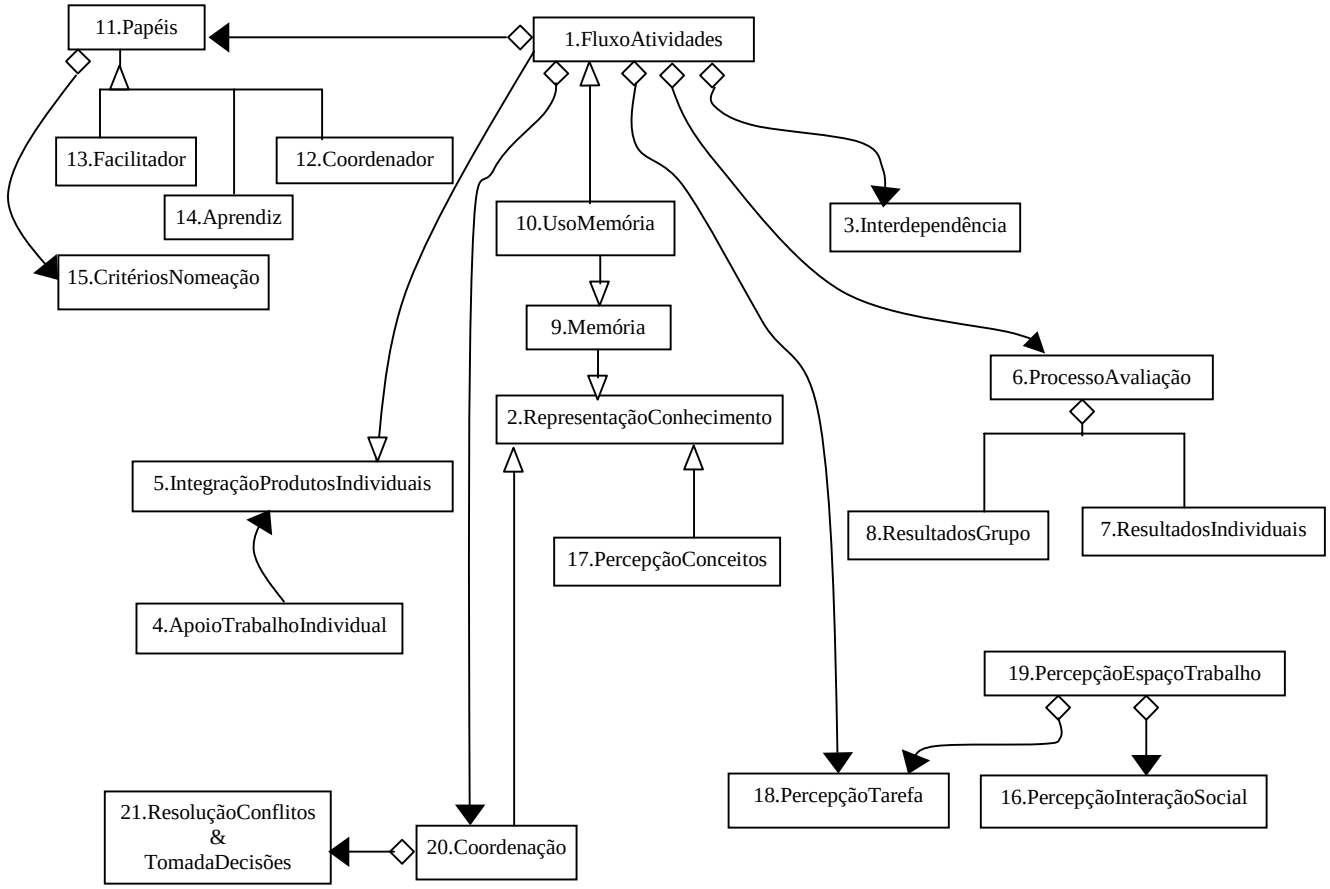


Figura 1 - Representação Gráfica da Linguagem de Padrões

1. Nome: Fluxo Atividades

Contexto:
Em ambientes cooperativos para aprendizagem baseada em projetos, diversas atividades são propostas a fim de que os alunos chegem ao objetivo educacional, ou seja, adquirir/construir conhecimento ao longo do processo. Em um projeto, há compromisso com a geração de produtos. Portanto, as atividades não são isoladas e desconectadas, mas compõem um fluxo necessário à execução do projeto. A composição deste fluxo vai determinar a característica pedagógica e funcional do projeto.

Problema:
Como definir e descrever o fluxo de atividades em ambientes de aprendizagem cooperativa baseada em projetos?

Forças:
O primeiro passo para a realização bem sucedida de um projeto é o seu planejamento. Para descrever um processo de trabalho, é necessário definir o relacionamento entre as diversas atividades: objetivos específicos, Papéis, Interdependência, regras, hierarquia, entradas/saídas, sub-produtos gerados, e ferramentas de apoio.
A noção de projeto (conjunto de atividades) como um todo é fundamental para a organização e Coordenação do trabalho de grupos. Por isso, é importante haver uma definição das tarefas apoiadas pelo ambiente e que estas tarefas levem à aprendizagem.

Solução:
Planeje as atividades de forma a levar os aprendizes gradualmente de uma perspectiva individual (levantamento de idéias, exploração) para uma perspectiva coletiva (argumentação, análise, comparação, decisão). O processo deve mostrar aos aprendizes a necessidade de interação para realização das tarefas. Lembre que nem sempre os indivíduos têm facilidade de trabalhar em equipe.
Crie ou permita que o grupo crie uma representação do processo, identificando principalmente como as interações entre os participantes deverão ocorrer, definindo assim o espaço cooperativo. Este espaço deve ter flexibilidade suficiente para ser redefinido sempre que o grupo avaliar esta necessidade.
O uso de um modelo de Workflow pode ser útil para representar o fluxo de atividades. Van der Veen et al. (1998) realizaram um estudo experimental sobre a aplicação de sistemas de workflow no contexto educacional e identificaram as semelhanças e diferenças entre processos educacionais e de negócios. Os resultados levaram à conclusão de que o uso de sistemas de workflow como apoio à aprendizagem baseada em projetos traz ganhos em relação às metas educacionais.

Usos conhecidos:
Estudos de caso realizados com o ambiente Zebu (Tiessen & Ward, 1999) levaram à conclusão da necessidade de prover ao professor mecanismos de suporte ao planejamento e visualização das atividades a serem propostas, de como deverá ocorrer o processo cooperativo e como deverá ser a participação dos alunos. Os mecanismos de suporte propostos têm como objetivo dar apoio ao professor no planejamento de atividades de aprendizagem interrelacionadas, que devam estimular os estudantes a participarem de um processo de engajamento progressivo em pesquisas.
Segundo Ferraris e Martel (2000), a regulação do espaço cooperativo traz os seguintes benefícios: facilita a organização dos participantes; favorece o seu comprometimento com a atividade conjunta; e aumenta a coesão do grupo. Sua função é definir como cada membro do grupo deverá participar da atividade cooperativa. O Modelo de Participação proposto por Ferraris e Martel (2000) é um modelo conceitual que descreve, formaliza e constrói o contexto da atividade cooperativa, os relacionamentos de dependência e a estrutura de trocas dentro do grupo.
No ambiente CLARE, a aprendizagem colaborativa sustenta-se em um modelo explícito - SECAI que “puxa” os aprendizes da posição externa, isolada e individual para a perspectiva interna, integrada e colaborativa em um artefato (Wan e Johnson, 1994).
O Modelo de Cooperação para Ensino/Aprendizagem de Disciplinas de Modelagem de Becker e Zanella (1998) é voltado para o domínio de conceitos de modelagem de dados através do desenvolvimento de exercícios, da crítica e discussão de alternativas para modelagem. O modelo provê um framework, onde estão definidos: (a) um processo, que ajuda os professores a definirem e estruturarem as atividades da classe; (b) papéis, a serem desempenhados por estudantes e pelo professor; e, (c) objetos compartilhados durante o processo.
Miao et al. (2000) propõem o uso de Protocolos de Aprendizagem, que são descritos como scripts computacionais para definir, guiar e controlar a interação social dos processos cooperativos em ambientes virtuais de aprendizagem.

2. Nome: Representação Conhecimento

Contexto:
Atividades cooperativas pressupõem compartilhamento de conhecimento e comunicação. Portanto, os objetos de estudo, que são as formas concretas através das quais ocorrem trocas, manipulação e produção de informação e conhecimento, são a base de funcionamento de ambientes de aprendizagem cooperativa.
Objetos compartilhados podem ser divididos em duas categorias: objetos de percepção e objetos de manipulação. Objetos de percepção são relacionados às informações necessárias à execução do processo de trabalho (registros de interações, visões sobre andamento das tarefas), e objetos de manipulação são relacionados aos produtos das tarefas executadas (documentos).

Problema:
Quais são os mecanismos necessários para fazer a representação de conhecimento eficiente em ambientes de aprendizagem cooperativa baseada em projetos?

Forças:
As interações em grupo e interpessoais envolvem o uso da linguagem na reorganização e na modificação dos entendimentos e das estruturas de conhecimento individuais, e portanto a aprendizagem é simultaneamente um fenômeno privado e social.
Para promover a aprendizagem através do compartilhamento é fundamental que se crie um entendimento comum sobre os objetos de estudo. As pessoas devem ter disponíveis mecanismos formais, estruturados, para representar um conhecimento, ou questionar uma colocação, de forma que todos os participantes tenham oportunidade de entender o que se está querendo comunicar. Representar conhecimento é facilitar a tradução do pensamento e das idéias dos participantes de interações em ambientes cooperativos.
A forma de representação está diretamente relacionada ao tipo ou área de conhecimento e ao uso que se fará desta informação. Além disso, deve relacionar uma informação ao contexto do processo cooperativo que está inserida, facilitando a captura e recuperação da Memória do grupo e a Coordenação do trabalho.
Em ambientes CSCL, a representação de conhecimentos permite também a possibilidade da existência de guias no processo de aquisição de conhecimento, uma vez que o próprio sistema interpreta as mensagens transmitidas nas interações entre os Aprendizes.

Solução:
Disponibilize uma linguagem comum para estruturação dos objetos de estudo. Esta linguagem deve conter construtores de três tipos: (a) dicionário de termos comuns determinados de acordo com o domínio específico do projeto, que vai auxiliar na equalização dos conteúdos intercambiados; (b) símbolos gráficos que identifiquem o tipo de informação ou mensagem que se deseja transmitir, criando um protocolo de comunicação nas interações, e (c) elementos que estabeleçam associações entre objetos, que permitem a formação de relacionamentos entre as informações, criando uma rede articulada.
Para que o ambiente não se torne muito rígido, tolhendo iniciativas particulares e criativas, flexibilize esta linguagem permitindo a inclusão de novos construtores nos três níveis. Desta forma, os membros do grupo vão desenvolver também um raciocínio sobre a lógica contida na elaboração de suas contribuições ao grupo.

Usos conhecidos:
O ambiente CLARE utiliza uma linguagem de representação de conhecimento semi-estruturada RESRA (Representational Schema of Research Artifacts) que implementa três construtores conceituais: primitivas nós, primitivas links, e formas canônicas (Wan e Johnson, 1994). Os estudantes utilizam esta linguagem para representar suas análises e conclusões sobre textos estudados. A partir daí, o grupo constrói um entendimento coletivo.
No Belvedere, idéias e relacionamentos são representados como objetos que podem ser apontados, ligados a outros objetos e discutidos, permitindo a construção de representações de relações lógicas e retóricas dentro de um debate. Sua interface se assemelha a um editor gráfico, que provê diagramas de argumentação disponibilizando formas geométricas para diferentes tipos e componentes de argumentos com links positivos e negativos, múltiplas formas de ligações, e possibilidades de anexos para acomodar argumentos complexos (Suthers e Weiner, 1995).
O ambiente PIE provê suporte a representações textuais e gráficas que ajudam estudantes a articularem suas intuições sobre probabilidade e embasá-las no processo de construção de argumentos que reflitam um entendimento padronizado (Enyedy et al., 1997).
Algebra Jam (Singley et al., 1999) usa uma tipologia de mensagens, que tem como objetivo prover níveis de interpretação mais compreensíveis de uma conversa entre os estudantes e com um tutor. O tipos de mensagens provêem uma forma estruturada que resume as possíveis intenções de comunicação entre os indivíduos.

3. Nome: Interdependência

Contexto:
Através de vários estudos e experiências realizados com grupos pequenos, pesquisadores da área de educação identificaram duas características que tornam os grupos bem sucedidos na situação de aprendizagem. Estas características são chamadas de interdependência positiva e responsabilidade individual (Johnson et al., 1990; Slavin, 1990).
Interdependência positiva significa que os membros de um grupo sentem que necessitam "caminhar juntos" para realizar uma tarefa, ou seja, bons resultados e maus resultados obtidos por um membro da equipe têm o mesmo efeito para toda a equipe. Responsabilidade individual significa que todos os membros de um grupo devem participar ativamente para que o grupo venha a ser bem sucedido.
Sendo assim, um processo realmente cooperativo, no sentido de que as pessoas têm um objetivo comum e necessitam interagir para alcançá-lo, é definido pelo grau de interdependência encontrado nas tarefas a serem realizadas.

Problema:
Como definir uma atividade em um ambiente de aprendizagem, garantindo que ela só poderá ser realizada de forma cooperativa, ou seja, quais são os elementos ou características que definem interdependência positiva na realização de uma tarefa?

Forças:
A estruturação de atividades cooperativas visando melhores resultados em termos do trabalho em grupo também deve ser aplicada no caso de ambientes cooperativos para aprendizagem apoiados por computadores.
Portanto, a aplicação dos fundamentos teóricos dos estudos realizados por educadores pode ser aplicada na implementação de ambientes cooperativos para aprendizagem apoiados por computadores.

A cooperação deve sustentar-se em um modelo explícito de cooperação, onde a natureza do processo cooperativo esteja clara para os participantes. Este modelo deve ser flexível para comportar diversas formas de cooperação, de coordenação e de comunicação.
Os objetivos e tarefas de um grupo devem ser projetados de forma que os aprendizes acreditem que formam uma equipe: o esforço de cada membro é importante e indispensável para o sucesso do grupo; e cada membro tem sua contribuição particular para dar ao grupo, de acordo com seus recursos, conhecimentos prévios, Papéis, e responsabilidade nas tarefas.

Solução:
Analise os elementos de interdependência da tarefa proposta, ou seja questione quais são as características da atividade indicadoras da necessidade do trabalho em grupo para sua realização. Uma vez estabelecida a legitimidade da cooperação, defina regras de cooperação – um conjunto de normas através da qual a tarefa deve ser desempenhada garantindo a aplicação de trabalho cooperativo. As regras devem ser criadas de forma a vincular o trabalho de um membro de um grupo aos demais, garantindo que o resultado final só poderá ser atingido, se todos trabalharem de forma cooperativa, compreendendo como será a Integração Produtos Individuais.
Johnson et al. (1990) sugerem que as tarefas propostas devem possuir um mais dos seguintes elementos de interdependência positiva:

1. Interdependência de Objetivo
   O grupo deve possuir um objetivo comum.
   Exemplo: Um grupo deve ler e criticar uma composição escrita por outro grupo para uma apresentação final.
2. Interdependência de Papéis
   Cada membro tem uma tarefa a cumprir, de forma que o objetivo final seja alcançado somente se cada fizer a sua parte. Neste caso, deve haver interação entre os membros do grupo e pode-se fazer revezamento entre os papéis assumidos por cada um.
   Exemplo: Um membro do grupo lê uma passagem de um texto, outro deve escrever um resumo sobre o que foi lido, e outro deve revisar aquilo que foi escrito.
3. Interdependência de Inimigo Externo
   Membros do grupo cooperam para defender-se de um “inimigo” comum. Neste caso, pode-se criar jogos onde haja competição entre grupos, existindo, porém dentro do mesmo grupo cooperação para chegar ao final da competição.
   Exemplo: Jogos de perguntas e respostas, onde o progresso de cada membro do grupo é avaliado e acrescenta ou não pontos para toda a sua equipe.
4. Interdependência de Recursos
   Os membros do grupo possuem recursos diferentes e estes devem ser compartilhados para realização de uma tarefa.
   Exemplo: Na dinâmica Jigsaw (Aronson,1978), grupos de estudantes possuem informações sobre partes diferentes de um tema. Os membros de cada grupo que possuem as mesmas partes se reúnem, discutem o assunto e pensam qual seria a melhor forma de explicá-la para os outros. Feito isto, retornam ao seu grupo original e cada um deverá explicar sua parte para os outros e então elaborar um produto final contendo todas as partes.
5. Interdependência de Identidade
   O grupo cria uma identidade própria dependendo da atividade na qual está envolvido.
   Exemplo: Estudantes de ciências podem dar o nome de um grande cientista ao seu grupo.
6. Interdependência de Recompensas

O grupo trabalha junto na expectativa de receber algum tipo de recompensa. As recompensas não podem se tornar motivo de sensação de injustiça dentro do grupo.

Usos conhecidos:
A definição deste padrão está baseada em estudos com grupos de aprendizagem na área de educação, onde observou-se resultados de sucesso e falha com o uso de técnicas e dinâmicas de estruturação do trabalho: Aronson, 1978; Johnson et al., 1990; Slavin, 1990.
O ambiente PIE (Gifford e Enyedy, 1999) apresenta regras de interação em cada uma das tarefas. Em estudos sobre o uso do ambiente, concluiu-se que as regras e a divisão de trabalho propostas tornaram mais efetiva a organização das atividades e consequentemente os resultados da aprendizagem dos estudantes.
No trabalho desenvolvido por Tiessen e Ward (1999), cabe ao professor apoiar os estudantes na construção coletiva de seu trabalho. Isto é feito através da configuração/estruturação de atividades que tornem as contribuições individuais fundamentais para o alcance das metas do grupo e que estimulem os estudantes a realizarem seu trabalho em conjunto com seus parceiros.

4. Nome: Apoio Trabalho Individual

Contexto:
Atividades de aprendizagem propostas em ambientes cooperativos são realizadas pelos membros do grupo no decorrer de um processo estabelecido. Estas atividades podem ser cooperativas ou individuais de acordo com o Fluxo Atividades definido pelo processo. Na realidade, existem momentos para trabalho em grupo, e outros para produções individuais. Mesmo nas tarefas cooperativas, deve existir um espaço para expressão individual.

Problema:
Que tipo de mecanismos devem ser disponibilizados em ambientes de aprendizagem cooperativa baseada em projetos para apoiar o trabalho individual?

Forças:
A aprendizagem é um processo inerentemente individual, não coletivo, que é influenciado por uma variedade de fatores externos, incluindo as interações em grupo e interpessoais. Aprender cooperativamente implica na troca entre indivíduos, e assume que, de alguma maneira, o todo é maior do que as partes individuais, de modo que a cooperação pode produzir ganhos superiores à aprendizagem solitária.
Aprendizagem cooperativa não significa necessariamente aprender em grupo, mas muitas vezes em poder contar com outras pessoas para apoiar sua aprendizagem e dar retorno se e quando necessário, no contexto de um ambiente não competitivo (Kaye, 1991).
Pesquisas na área de CSCW (Computer-Supported Cooperative Work) também apontam a necessidade de apoio à realização de tarefas individuais em ambientes cooperativos.

Solução:
Apresente ferramentas que permitam ao Aprendiz elaborar visões privadas sobre os objetos de estudo trabalhados no contexto do ambiente, deixando livre a opção de mostrá-las ou não ao outros membros do grupo, no momento que achar conveniente.
Estas ferramentas podem também estar presentes como funcionalidades das ferramentas cooperativas, de forma a disponibilizar espaços individuais, por exemplo, para rascunhos, dentro das tarefas de grupo.

Usos conhecidos:
O ambiente WebGuide (Stahl, 1999), provê funcionalidades que permitem que cada estudante manipule e analise as idéias discutidas pelo grupo, selecionando, editando, arranjando e relacionando e resumindo notas livremente de acordo com sua perspectiva pessoal, sem afetar as visões das outras pessoas.
O Collaboratory Notebook (O'Neill e Gomez, 1994) implementa blocos de anotações privados, onde os participantes podem fazer seus relatos pessoais sobre a experiência coletiva, sem necessariamente apresentar ao grupo.
Janelas com visões sobre aspectos identificados individualmente em um texto científico são disponibilizadas no ambiente CLARE (Wan e Johnson, 1994) para cada usuário. Outras janelas apresentam resumos das visões de outros usuários do sistema.

5. Nome: Integração Produtos Individuais

Contexto:
Em ambientes de aprendizagem cooperativa, muitas vezes, os trabalhos realizados pelos alunos são produtos da união de seus esforços. A cooperação em ambientes educacionais pode significar divisão de tarefas em partes controladas por diferentes colaboradores, ou esforço conjunto para realização de uma tarefa sem divisão de trabalho.
A divisão de tarefas entre membros individuais ou entre pequenos grupos pode se traduzir em maior rapidez na execução de uma atividade. Por outro lado, a realização das tarefas sem divisão de trabalho pode ampliar o debate, o compartilhamento de idéias e de conhecimento entre os participantes, bem como acarretar aumento na qualidade do material produzido. De qualquer forma, a aprendizagem cooperativa nunca deve se resumir a juntar partes ou recortes de trabalhos individuais.

Problema:
Como os produtos, ou resultados de atividades individuais devem ser disponibilizados, apresentados e integrados no trabalho do grupo, de forma que fique clara sua contribuição?

Forças:
Se o objetivo de ambientes de aprendizagem cooperativa é o compartilhamento e a interação entre os participantes para construção coletiva de conhecimento, os esforços individuais devem ser integrados, de forma a se complementarem como parte de um todo consistente, entendido por todo o grupo.
Colaborar implica em objetivos compartilhados e intenção explícita de 'somar algo' , ou seja, criar alguma coisa nova ou diferente através da colaboração, se contrapondo a uma simples troca de informação ou passagem de instruções (Kaye, 1991). Apesar de sempre existir o Apoio Trabalho Individual em ambientes de aprendizagem cooperativos, os Aprendizes devem ter noção de como integrar os produtos destes esforços quando for necessário.
Uma contribuição individual pode ter duas funções em um trabalho coletivo: (i) servir como parte ou complementação da produção; (ii) esclarecer, explicar ou argumentar sobre algum conceito discutido pelo grupo. Para que uma contribuição individual seja efetiva para o grupo e possa ser utilizada em outras etapas do trabalho coletivo, ela precisa estar clara para todos os membros do grupo. Isto acontece quando se usa uma representação comum do conhecimento comunicado. A solução para representação comum de conhecimento está descrita no padrão Representação Conhecimento.

Solução:
Deixe claro quais são as atividades individuais, quem é o seu responsável e qual o produto a ser gerado por ela. Todas estas informações devem estar explícitas na representação do Fluxo Atividades. Desta forma, ficará claro para o grupo, antes mesmo do iniciar a execução das tarefas, os momentos onde serão introduzidas as contribuições individuais.
Em cada objeto de estudo deve ser estar representada de alguma forma a contribuição dos membros do grupo. Esta representação pode ser feita, por exemplo, através de cores diferentes associadas a cada membro, ou do nome ou um apelido associado a uma parte do objeto.

Usos Conhecidos:
No ambiente CLARE (Wan e Johnson, 1994), a definição do processo mostra claramente a primeira fase, onde cada membro do grupo trabalha particularmente no resumo de um texto, para em uma segunda etapa apresentar o material produzido ao grupo e promover comparações e debate. Desta forma, fica explícita a contibuição de cada um e o momento em que esta é integrada ao trabalho coletivo.
No CSILE (Oshima, 1997) e Collaboratory Notebook (O'Neill e Gomez, 1994), as contribuições individuais são identificadas através dos nomes dos membros do grupo associados às notas escritas por eles.

6. Nome: Processo Avaliação

Contexto:
A avaliação da aprendizagem é o conjunto de ações organizadas com a finalidade de obter informações sobre o que foi assimilado pelo estudante, de que forma e em quais condições. Para tanto, é preciso elaborar um conjunto de procedimentos investigativos que possibilitem o ajuste e a orientação adequada. A avaliação deve funcionar por um lado como um instrumento que possibilite ao avaliador analisar criticamente a sua prática; e por outro, como instrumento que apresente ao avaliado a possibilidade de saber sobre seus avanços, dificuldades e possibilidades. Neste contexto estão inseridos os objetivos educacionais, ou seja, os conceitos ou habilidades que se pretende ensinar, e as próprias atividades projetadas.

Problema:
Como definir o processo de avaliação da aprendizagem, no contexto do desenvolvimento de um projeto em um ambiente cooperativo apoiado por computador?

Forças:
O processo de avaliação começa pelos objetivos do programa educacional, ou seja, o seu cerne está em determinar em que medida os objetivos pretendidos estão sendo realmente alcançados.
O processo de avaliação está diretamente relacionado ao tipos de atividades educacionais propostas, e portanto à teoria de aprendizagem na qual estão baseadas.
Na ótica de uma teoria sócio-cultural e construtivista, não é possível avaliar os conhecimentos construídos desvinculando-os do processo em que foram constituídos. Por isto, a avaliação deve ser contínua e deve permitir ao professor identificar e criar Zonas de Desenvolvimento Proximal (Vygostky). A avaliação é parte do processo e portanto deve estar definida no contexto do Fluxo Atividades. Através das diversas formas de avaliação, o professor pode dar feedback aos Aprendizes ao longo do processo.

No espaço educativo sócio-construtivista, os processos são mais relevantes que os produtos, e a realidade não deve ser reduzida somente à observação das concepções finais. A avaliação qualitativa deve ultrapassar a avaliação quantitativa, sem contudo dispensá-la.
O processo de avaliação de aprendizagem pode ser definido em algumas etapas:

1. Análise dos objetivos educacionais que formam um conjunto de especificações para avaliação.
2. Identificação de situações que dão ao aluno a oportunidade de expressar o comportamento implicado pelos objetivos educacionais.
3. Estabelecimento de instrumentos de avaliação.
4. Definição dos termos ou unidades de medida que serão utilizados para apresentar o resultado que se obteve com a avaliação.

Solução:
Em um ambiente cooperativo de aprendizagem baseada em projetos, disponibilize meios para que o avaliador possa especificar os momentos nos quais algum tipo de intervenção com fins de avaliação deverá ser realizada. Estas intervenções deverão estar inseridas no desenvolvimento das atividades propostas, através do Fluxo Atividades. Deve existir a possibilidade de serem feitas intervenções diferentes para cada membro do grupo, de forma que a avaliação possa ser individualizada.
As intervenções podem ser apresentações de dados coletados e armazenados no ambiente (Memória) sobre o desenvolvimento de cada indivíduo (Resultados Individuais) e do grupo como um todo (Resultados Grupo) no processo de aprendizagem, em vários aspectos: cognitivos, afetivos, que envolvam relações sociais, de cooperação, de participação, de poder de argumentação, crítica e criação.
A partir daí, deve-se examinar instrumentos de avaliação disponíveis ou desenvolvê-los especificamente para servir aos propósitos determinados em cada intervenção.

Usos Conhecidos:
Pesquisas na área de educação (Tyler, 1974; Secretaria Municipal de Educação do Rio de Janeiro, 1996) identificam etapas clássicas para um processo de avaliação, relacionado-o diretamente aos objetivos educacionais. Alguns ambientes apoiados por computadores adotam parte desta proposta e apontam solução específicas, tais como formas de representar os objetivos e ferramentas de apoio a algumas das etapas descritas (Leite e Omar, 1999; Tarouco e Hack, 1999).

7. Nome: Resultados Individuais

Contexto:
Em um ambiente cooperativo de aprendizagem apoiado por computadores, deve ser possível fazer avaliações acerca do desenvolvimento de cada membro de um grupo. Desta forma, pode-se ter noção do nível de desenvolvimento alcançado por cada um, de acordo com os objetivos pretendidos (que por sua vez podem ser diferentes para cada indivíduo).

Problema:
Como avaliar o resultado da aprendizagem de cada indivíduo trabalhando em ambientes cooperativos de aprendizagem baseada em projetos?

Forças:
Em ambientes construtivistas, baseados em teorias sócio-culturais, a avaliação deve ser baseada na observação do progresso individual, não só em relação ao ganho de novos conceitos, mas também em relação ao desenvolvimento de habilidades sociais.
O primeiro passo para implantar um Processo Avaliação é descrever a correlação entre os objetivos traçados com a definição daquilo que deverá ser avaliado. Feito isto, o avaliador deve ter meios para identificar situações onde serão aplicados determinados instrumentos de avaliação.
O ambiente deve prover diversos instrumentos de avaliação tanto quantitativa, quanto qualitativa, para serem utilizados de acordo com a necessidade prevista pelo avaliador. A prova única (igual para todos) deixa de ser o instrumento principal de avaliação. O avaliador deve ter meios para desenvolver exames individuais, de acordo com características de cada participante do processo. Os exames devem verificar o progresso do aprendiz e não somente uma resposta imediata.
A avaliação tradicional é sinônimo de testes com lápis e papel. Através deles, pode-se verificar a capacidade dos estudantes em analisar e tratar vários tipos de problemas verbais, vocabulário, leitura e outros gêneros de habilidade e aptidões facilmente expressos sob forma verbal. Porém, avaliação é muito mais do que isso. A avaliação deve ser reflexiva, crítica, emancipatória e deve buscar uma coerência na teoria e na ação. O ajustamento pessoal-social é avaliado com mais facilidade pela observação de pessoas em situações que envolvam relações sociais.

Solução:
Os instrumentos de avaliação em um ambiente de aprendizagem cooperativa devem capturar e apresentar informações relativas à observação das interações nos trabalhos em grupo e ao êxito na obtenção de soluções partilhada de problemas. Para isto, o ambiente deve prover fontes de informações sobre o processo de trabalho e os resultados obtidos. Permita que o avaliador configure que tipo de dados deseja monitorar.
O sistema deve permitir que os avaliadores façam anotações ou comentários estruturados (que podem ter algum tipo conceito associado) sobre o desenvolvimento dos alunos. A estruturação facilitará a apresentação e consulta de resultados.
Além disto, outros recursos poderão ser utilizados, tais como questionários, entrevistas e auto-avaliação. O avaliador deve ter a possibilidade de enviar perguntas aleatórias em determinadas situações configuradas por ele. O avaliador ou o próprio aluno podem ser responsáveis pela criação da estrutura de tópicos a serem avaliados.

Usos Conhecidos:
Trabalhos na área de educação apontam para a utilização de alguns instrumentos de avaliação na linha qualitativa e individualizada, de acordo com paradigmas educacionais mais modernos.
O Virtual Campus fornece meios para comunicação, compartilhamento de conhecimentos e armazenagem de informações em uma sala virtual de aprendizagem cooperativa. A avaliação de participação individual no ambiente Virtual Campus (Maher, 1999) pode identificar não somente a quantidade de contribuição, mas também o conteúdo do que foi apresentado. As informações armazenadas podem fornecer indicadores do tipo de colaboração e da extensão das interações entre os participantes. A avaliação é baseada nas informações observadas durante o processo de aprendizagem sobre o nível de participação de cada membro do grupo.

O ambiente de aprendizagem chamado “Virtual School” consiste em um notebook colaborativo que permite personalizar ou compartilhar espaços de trabalho para planejamento, organização, desenvolvimento e fazer anotações sobre projetos científicos. Uma única interface integra um conjunto de ferramentas de groupware com vários mecanismos de comunicação síncronos e assíncronos (Insenhour et al., 2000). As ferramentas de comunicação construídas no Virtual School incluem fóruns de discussão estruturados, e-mail, chat em tempo real, e vídeo conferência. Um servidor foi implementado com o objetivo de coordenar os usuários.
Filosofias de avaliação quantitativa e qualitativa são aplicadas a todos os níveis de métodos e dados. Os métodos são entrevistas, questionários, observações diretas, vídeos, e sistemas logs. Além disso, várias informações são capturadas como anotações, conversas de chat,, e-mail, que serão muito úteis para uma posterior avaliação.
Mühlenbrock e Hoppe (1999) propõem um framework para aprendizagem cooperativa apoiada por computador que monitora e gerencia as interações entre os grupos em cenários locais e distantes. Fornece mecanismos adaptáveis para processos de análise automatizados assim como para visualização e feedback.

8. Nome: Resultados Grupo

Contexto:
Em um ambiente cooperativo de aprendizagem apoiado por computadores, deve ser possível fazer avaliações acerca do desenvolvimento alcançado pelo grupo como uma entidade única. Desta forma, estará sendo verificado o resultado do trabalho em equipe e como cada participante se situa neste contexto.

Problema:
Como avaliar o resultado da aprendizagem de um grupo como um todo em ambientes cooperativos de aprendizagem baseada em projetos?

Forças:
Em ambientes de aprendizagem baseados em teorias sócio-culturais, a avaliação deve ser feita através da observação do progresso individual, não só em relação ao ganho de novos conceitos, mas também em relação ao desenvolvimento de habilidades sociais.
As características de Interdependência presentes neste tipo de ambiente mostram a importância de se prover meios para avaliação do grupo, uma vez que o trabalho realizado foi resultado de esforços conjuntos. Para se avaliar os resultados de um grupo, este precisa ser considerado uma entidade única. Neste caso é necessário traçar objetivos de aprendizagem para o grupo e prover instrumentos que permitam a avaliação destes objetivos. Um dos objetos a serem avaliados é o próprio produto construído pelo grupo.
Através do Processo Avaliação o avaliador deve definir instrumentos de avaliação qualitativa e quantitativa.

Solução:
Procure definir relatórios de apresentação dos produtos gerados pelo grupo ao longo da execução do projeto, de forma a tornar explícita a reflexão dos objetivos do grupo no trabalho feito. O professor/Facilitador pode pré-definir o formato dos relatórios e com isso ajudar os Aprendizes a entender o próprio Processo Avaliação e verificar seu progresso.

As informações relativas à observação das interações nos trabalhos em grupo também podem ser utilizadas para análise dos resultados do grupo através de comparações de contribuição no produto final e participação ao longo do processo.
Permita também que os membros do grupo avaliem, através de questionários e entrevistas, o produto do trabalho realizado e cada um dos membros de seu grupo em relação à participação e cooperação.
Para permitir uma análise quantitativa, o sistema deve prever a realização de exames em grupo que devem verificar o progresso de aprendizagem de conceitos/habilidades da equipe.

Usos Conhecidos:
O ambiente Design Discussion Area (DDA) apoia a apresentação dos projetos de dispositivos desenvolvidos por grupos de estudantes. Eles devem mostrar como funcionam seus projetos, explicar as decisões de projeto e discutir os próximos passos do desenvolvimento, através de relatórios pré-definidos. Os relatórios têm como objetivo ensinar aos estudantes como organizar e articular suas experiências. Além disso, o sistema permite que outros estudantes possam fazer perguntas, dar sugestões e apontar problemas utilizando formatos estruturados (Kolodner e Nagel, 1999).

9. Nome: Memória

Contexto:
As atividades cooperativas apoiam-se na interações entre os pares e geram produtos. O ambiente de aprendizagem baseada em projetos deve armazenar as diferentes versões dos produtos gerados, bem como a história de sua construção, através do registro das interações.
Memória Organizacional é uma área de pesquisas relacionada a artefatos de cooperação que podem ser apontados como indicadores explícitos do que tem acontecido em uma organização. Estes artefatos incluem por um lado produtos (documentos e bens fabricados); por outro, registros de colaboração e idéias, particularmente sequências de reuniões, perguntas freqüentemente solicitadas (FAQ), e listas que registram conhecimento comum em um tópico particular. Estes tipos de objetos representam o conhecimento armazenado (a memória) de uma organização, grupo, ou projeto; e eles devem ser armazenados, mantidos e indexados.

Problema:
Como prover suporte à captura, armazenamento e recuperação de informações sobre a memória do processo de trabalho de um grupo em um ambiente de aprendizagem cooperativa baseada em projetos?

Forças:
No caso de ambientes de aprendizagem, a memória do trabalho do grupo pode ser um importante repositório de soluções identificadas e adotadas, servindo como base de estudo para outros grupos e podendo trazer novas idéias e perspectivas sobre um determinado problema. Além disso, a memória do desenvolvimento dos projetos será útil para o Processo Avaliação da aprendizagem pelos professores e estudantes, pois pode-se reconstituir o processo de construção do conhecimento coletivo, bem como a participação de cada membro do grupo.
Informação coletada automaticamente por um sistema computacional é limitada pelo fato de que muitas interações podem ocorrer fora do sistema. Além disso, há o risco de que se resulte em uma quantidade grande de dados registrados não gerenciáveis.

Por outro lado, informação coletada manualmente, ou seja, os participantes precisam fornecer a informação em determinados momentos e estão a par disto, nem sempre é suficiente para entender e avaliar o processo educacional. A observação informal se faz necessária.
Em ambientes de aprendizagem cooperativa apoiados por computadores, a captura de informações de memória das atividades deve ser manual (os estudantes preenchem questionários de avaliação sobre o andamento do processo), e automática (o sistema guarda estruturas de informações sobre as interações dos membros do grupo na execução de suas tarefas em formato estruturado). A estrutura deve ser definida de acordo com o domínio do projeto e com padrões de Representação Conhecimento.
Alguns fatores são importantes para a captura de informações mais seguras e completas: os participantes devem estar preparados para responder questões sobre o seu processo de trabalho, por isso devem conhecê-lo bem; e os participantes devem ter um entendimento sobre a estruturação na troca de informações e construção de bases coletivas de conhecimento.
Segundo Dias (1998), a questão mais importante no suporte à memória de grupo em ambientes cooperativos é a definição de uma estrutura através da qual as informações serão representadas e mantidas.

Solução:
Ao final da realização de um projeto, deve-se ter arquivado um conjunto de elementos que caracterizam e permitem uma discussão sobre este. Os elementos a serem preservados no caso de ambientes CSCL são:

1. Documentos: descrevem os produtos e sub-produtos gerados ao longo e ao final do projeto, podem ter versões que devem ser guardadas como marcos e devem ter explicações associadas a elas que identifiquem os motivos pelos quais foram guardadas.
2. Discussões: são atividades específicas onde os membros do grupo (ou sub-grupos) se reúnem para discutir um determinado assunto de forma síncrona ou assíncrona, devem ser realizadas através de ferramentas apropriadas (do tipo chats, fóruns de discussões ou listas de mensagens), podem ter resultados associados, por exemplo, a decisão a respeito de um problema.
3. Diálogos: são conversas informais ou trocas de informações no contexto da execução de uma atividade no processo de trabalho, devem ser armazenadas como parte da memória de execução de uma atividade.

Para incluir todos os elementos em uma estrutura formal, que permita representação e relacionamentos entre eles, crie uma unidade básica comum de dados no ambiente. A metáfora mais apropriada neste caso é a do hipertexto, pois os indivíduos devem ser responsáveis pela organização de suas perspectivas sobre o conhecimento. Conklin (1996) afirma que as características de uma tecnologia capaz de de capturar informações sobre a memória organizacional devem combinar hipertexto e um método retórico que forneça dados semi-estruturados.
O modelo de estruturação das informações é derivado da Representação Conhecimento, onde deve-se incluir os construtores apropriados para cada caso. Por exemplo, no caso de Discussões, deve-se criar construtores que permitam um modelo de argumentação e Tomada Decisões.

Usos conhecidos:
Um dos sub-sistemas do ambiente ARCOO, Modelagem do Conhecimento, oferece instrumentos para criar e manter mapas de conceitos e bases de informações que compõem o conhecimento coletivo, criando a memória compartilhada em um grande hipertexto (Barros & Borges, 1995).

Em WebGuide (Stahl, 1999), os participantes criam uma rede estruturada de perspectivas onde adicionam livremente links entre suas notas pessoais e as do seu grupo. Esta rede representa e apoia as dinâmicas entre indivíduos e grupos, definindo o processo de colaboração.

10. Nome: Uso Memória

Contexto:
O uso da informação de Memória armazenada e recuperada está relacionado ao aspecto da interpretação. Ter acesso a dados e documentos sobre de acontecimentos passados não é suficiente, é preciso entender seu significado e saber onde aplicá-los, ou seja, transferir o contexto.

Problema:
Que formas podem ser utilizadas para apresentar dados provenientes de atividades já realizadas, de forma que fique explícito o contexto e os participantes desta atividade?

Forças:
A recuperação e apresentação de informações sobre o desenvolvimento de um projeto em ambientes de aprendizagem tem como objetivo a avaliação do projeto e aprendizagem de soluções e conhecimento acumulado. Trazer de volta dados sobre um projeto pode auxiliar outros grupos de estudantes a desenvolverem seus próprios projetos e trabalhar sobre informações já coletadas e discutidas de forma a tentar produzir inovações.
Porém, a interpretação de qualquer coisa depende do contexto dentro do qual é interpretado, ou seja, interpretação é um processo altamente localizado e subjetivo. Da mesma maneira que uma pessoa interpretará algo diferentemente de outro em uma determinada situação, a mesma pessoa em duas colocações diferentes interpretará algo diferentemente. Este é o mesmo problema da hermenêutica: interpretação da escrita por pessoas diferentes do autor original. A resposta básica desta disciplina é que a interpretação sempre é localizada e subjetiva.
Um dos riscos de qualquer mecanismo que mostra o que outras pessoas estão fazendo e estão sabendo (ou fez e soube) é sobrecarga de informação. Se as informações são capturadas em um nível de granularidade alto, estas precisam ser filtradas de alguma maneira. As informações relativas a interações em ambientes CSCL podem conter muitos dados irrelevantes aos propósitos de quem procura, como por exemplo, conversas informais sem significado para o projeto.

Solução:
Crie associações entre as descrições das atividades no modelo gráfico do Fluxo Atividades e as discussões e diálogos estruturados segundo modelos de argumentação provenientes da Memória de grupo. Desta forma, identifica-se o objetivo da atividade, os produtos gerados e as interações que os membros do grupo tiveram para concluí-la.
Cada uma das informações será visualizada através de sua ferramenta específica, se desejado pelo usuário, e uma ferramenta de buscas e filtragem de informações poderá ser utilizada para chegar a um nível de detalhamento maior sobre o estudo.

Usos Conhecidos:
Sistemas de workflow (Marshak, 1995) utilizam representações gráficas para facilitar a visualização dos processos e as informações associadas a eles.

11. Nome: Papéis

Contexto:
Papéis representam coleções de usuários em sistemas cooperativos e estão geralmente associados a funções lógicas assumidas por atores (participantes, indivíduos ou sub-grupos, agentes ou sistemas computacionais), na execução de uma atividade. Papéis podem ser formais ou informais, permanentes ou temporários; podem ser assumidos espontaneamente ou delegados.
Na visão de alguns autores da área de CSCW (Computer-Supported Cooperative Work), papéis também podem representar descrições dinâmicas de usuários, avaliadas na medida em que o sistema cooperativo é utilizado por eles. Além disso, um mesmo participante pode assumir diferentes papéis ao mesmo tempo, na execução da tarefa.
Os sistemas cooperativos podem implementar mecanismos apropriados que criem facilidades para a execução de tarefas específicas relacionadas a determinados papéis. Mesmo os ambientes de aprendizagem que não designam papéis explicitamente aos seus usuários apontam a importância da existência de mecanismos de suporte a diferentes papéis. É o caso de Guzdial et. al. (2000), que afirmam que o ambiente CoWeb pode apoiar papéis específicos, sem contudo torná-los explícitos no ambiente.

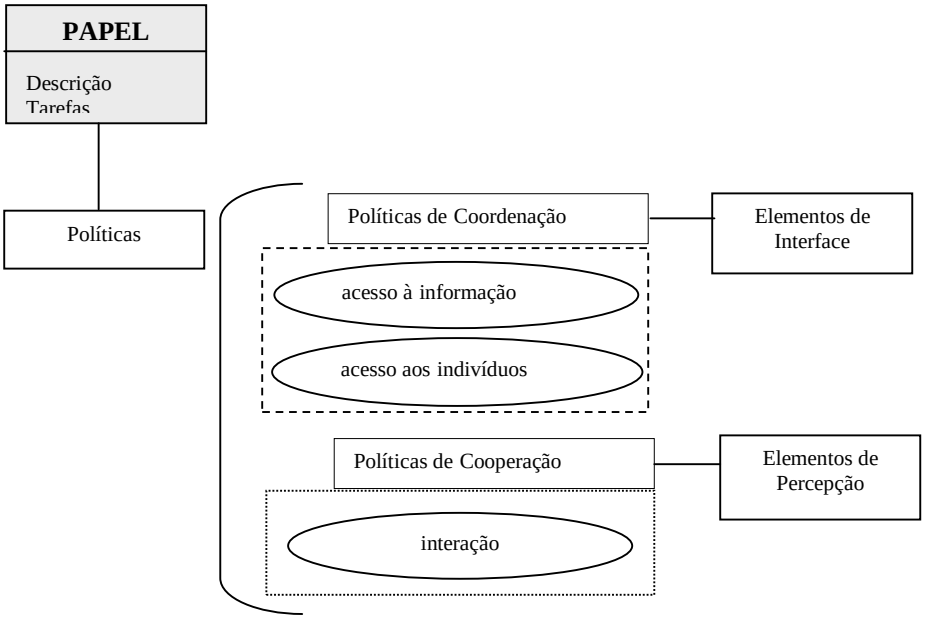
Problema:
Como identificar/definir mecanismos a serem disponibilizados em um ambiente de aprendizagem cooperativa baseada em projetos para apoiar papéis específicos?

Forças:
A importância da definição de papéis em sistemas cooperativos em geral está no fato de que usuários diferentes precisam ter acesso a informações diferentes baseadas nas tarefas associadas aos seus papéis. No caso de ambientes de aprendizagem cooperativa, a importância está no fato de que, ao exercer funções e responsabilidades diversas, cada membro do grupo adquire conhecimento sobre um determinado domínio a partir de diferentes perspectivas, na medida em que o processo de aprendizagem ocorre.
Os papéis definem as relações, as formas de interação entre os participantes e o acesso a objetos compartilhados. Vários papéis podem ser assumidos em sistemas cooperativos, porém em ambientes de aprendizagem, alguns deles são considerados fundamentais dependendo do tipo de tarefa executada, por exemplo, Aprendiz, Facilitador e Coordenador.
Para Guzdial et al. (2000), papéis são produto de um processo social, que ocorre independente do uso do ambiente, somado aos benefícios trazidos pelo ambiente computacional. Portanto, os papéis são explícitos, naturais, estáveis e diretamente associados às características dos indivíduos. A tarefa cooperativa desempenhada pelo grupo deve apontar que tipo de características individuais são requeridas para sua execução.

Solução:
Para que o conhecimento seja adquirido sob diferentes óticas, deve-se prever uma progressão e complementação de papéis, ainda que esta não deva ser uma obrigatoriedade nos ambientes. Portanto é preciso estabelecer um conjunto de papéis necessários a cada uma das tarefas. Estabeleça os papéis, porém torne o ambiente flexível para apoiar dinamicamente as possíveis mudanças nas formas de trabalho e consequentemente nos papéis desempenhados. Ao identificar um papel típico no desenvolvimento de uma tarefa, detalhe esta tarefa e defina as políticas de interação dos participantes.

Políticas descrevem um contingente genérico através do qual eventos específicos são avaliados e manipulados, ou seja, as políticas regem as formas particulares de interação entre usuários e entre usuários e aplicações. Os objetivos das políticas em ambientes de aprendizagem cooperativa são gerenciar imprevistos e ensinar formas de cooperação. Os ambientes cooperativos devem apoiar diferentes políticas de coordenação (controle de acesso a objetos compartilhados) e de cooperação (formas de interação).
Em relação às políticas de coordenação, defina elementos de interface de utilidade específica do papel. Estes elementos dão acesso a ações que podem ser restritas aos usuários que estão assumindo o papel em determinado momento, ou podem ser apresentadas a todos os usuários. Isto deve ser configurável na definição da tarefa.
Em relação às políticas de cooperação, defina elementos de percepção específicos para o papel. Estes elementos têm como função prover informação necessária sobre o desempenho de determinado papel e das possibilidades de acesso ao mesmo por outros usuários.

O esquema de relacionamentos a seguir resume os elementos relacionados aos papéis em um ambiente cooperativo:



Usos Conhecidos:
Para Singley et al. (2000), a questão da melhoria da colaboração através da designação de papéis e configuração de equipes de trabalho ainda não está totalmente explorada. Porém, observa-se que os trabalhos nesta área apontam para o caminho descrito na solução apresentada neste padrão.
A estratégia do ambiente Algebra Jam (Singley et al., 2000) é prover ferramentas no contexto da interface para diferentes tipos de comportamentos colaborativos e associar o uso de ferramentas específicas, de formas específicas, a papéis específicos. Para isto, foi definida uma tipologia de papéis relativos a tarefas particulares envolvidas na solução de problemas, identificada através da disponibilização de determinadas ações na interface do sistema. Segundo a experiência destes autores com o uso do ambiente, há um ganho pedagógico no processo de aprendizagem pelo fato de ter os participantes assumindo cada um dos pontos de vista relacionados aos papéis definidos.

O sistema pode funcionar segundo dois modos. No primeiro caso, permite que os papéis sejam designados explicitamente (modo prescritivo), e somente um sub-conjunto de ações é habilitado na interface. No segundo, os papéis não são designados explicitamente (modo não prescritivo), ficando a interface totalmente liberada para todos os participantes.
Nas pesquisas com o ambiente CoWeb, Guzdial et al. (2000) identificaram uma série de papéis atuando sistematicamente no uso do sistema, em diversos contextos educacionais. Apesar dos papéis não serem designados explicitamente, na medida em que foram identificados, procurou-se prover ferramentas e mecanismos para apoiar as suas atividades e necessidades específicas. Os mecanismos criados são funções disponíveis na interface e elementos de percepção que disponibilizam informações particulares, que são utilizadas por usuários específicos.
No ambiente Kansas (Smith, Hixon e Horan, 1998), não há definição explícita de papéis, porém, os usuários "vêem" e "ouvem" coisas diferentes de acordo com as tarefas designadas a eles. Isto é feito através do dimensionamento das janelas apresentadas a cada usuário, que permitem acesso a elementos de interface e percepção diferentes em cada caso. Além disso, implementa um sistema de "capabilities", que representam funcionalidades do sistema habilitadas ou não nas interfaces de cada usuário. Ao longo das sessões, os papéis poder evoluir através da alteração dinâmica das "capabilities" associadas ao participante.

12. Nome: Coordenador

Contexto:
Vários papéis podem ser assumidos em sistemas cooperativos, porém em ambientes de aprendizagem, alguns deles são considerados fundamentais dependendo do tipo de tarefa executada e encontram-se presentes nos ambientes ainda que de forma não explícita. Um exemplo é um coordenador geral para o projeto e/ou coordenadores das atividades a serem desenvolvidas.

Problema:
Como apoiar as formas de interação do Coordenador em um ambiente de aprendizagem cooperativa baseada em projetos?

Forças:
O coordenador é um papel importante em ambientes de aprendizagem cooperativa baseada em projetos, pois o exercício da liderança também faz parte do aprendizado sobre como trabalhar cooperativamente.
O coordenador estabelece a estratégia para solução dos problemas inerentes às atividades do projeto, determina os recursos disponíveis e necessários, designa tarefas a indivíduos, gerencia as atividades e avalia seu progresso. O coordenador pode participar a princípio de qualquer atividade dentro do processo. O coordenador se relaciona basicamente com os Aprendizes.

Solução:
As políticas de coordenação e de cooperação definem as formas de interação do coordenador no contexto do ambiente de aprendizagem:

- Políticas de coordenação: O coordenador deve ter acesso de leitura a todos os objetos e espaços compartilhados pelo grupo em todos os momentos do desenvolvimento do projeto. Porém, o acesso à edição deve ser restrito a objetos relacionados às suas tarefas específicas.

- Políticas de cooperação: O coordenador identifica os recursos para execução de uma atividade e passa para o grupo (por exemplo, textos para uma discussão). O coordenador tem acesso direto à execução do fluxo de trabalho de cada participante, podendo enviar mensagens de avaliação de progresso para os membros do grupo. O coordenador tem tarefas específicas de interação dependendo da atividade cooperativa que está sendo desenvolvida (por exemplo, coordenar uma sessão de discussão de um texto, passando a vez para cada membro do grupo que solicitá-la).

Para apoiar a implementação destas políticas defina elementos de interface e de percepção apropriados.
Defina elementos de interface que permitam:

- Envio de mensagens sobre a avaliação do processo e os produtos do trabalho realizado (podem se traduzir em mensagens estruturadas aos participantes de acordo com a Representação Conhecimento).
- Ações de intervenção sobre as atividades em desenvolvimento (por exemplo, início, término, interrupção).

Defina mecanismos de percepção que permitam a identificação e conhecimento de:

- Informação sobre o Fluxo Atividades.
- Informação sobre os participantes (Percepção Interação Social): dados pessoais e dados sobre as responsabilidades nas tarefas individuais e de grupo.

Usos conhecidos:
No Ambiente Algebra Jam (Singley et al., 1999), são designados explicitamente cinco papéis: Observador, Aprendiz, Especialista, Líder (Coordenador) e Guia (Facilitador). O Coordenador estabelece estratégias para solução dos problemas, coordena as ações e avalia o progresso do trabalho. O coordenador pode, por exemplo, enviar mensagens estruturadas de sugestões para o uso de recursos ou para demonstrar aprovação ou reprovação quanto a um produto gerado. Além disso, possui elementos de interface específicos para realização de suas tarefas, tais como apresentar metas e designar tarefas através de um “quadro-negro” compartilhado.
Em estudos realizados com o ambiente CoWeb, (Guzdial et al., 2000) identificaram o papel do usuário central ou coordenador. Este usuário normalmente gerencia a estrutura do sistema e guia as interações dos autores de forma a produzir uma melhor definição do espaço compartilhado. Para apoiar este trabalho, é disponibilizada a função de modificação dos títulos das páginas sem a perda de seus links, através da interface do sistema.
Em uma configuração do ambiente Kansas (Smith, Hixon e Horan, 1998) que apoia o paradigma de aprendizagem DTVI (Distributed Tutored Video Instruction), é proposta a definição explícita de quatro papéis: estudante, facilitador, coordenador e desenvolvedor. Para cada um deles é apresentada uma interface diferente conforme suas funções específicas. O coordenador possui acesso liberado a todas as funções e mecanismos de percepção no ambiente.

13. Nome: Facilitador

Contexto:
Vários papéis podem ser assumidos em sistemas cooperativos, porém, em ambientes de aprendizagem, alguns deles são considerados fundamentais dependendo do tipo de tarefa executada, e encontram-se presentes nos ambientes ainda que de forma não explícita. Um exemplo é o facilitador.

Em ambientes de aprendizagem sócio-construtivistas, como no caso dos cooperativos, os aprendizes são considerados sujeitos ativos na busca e construção de seu próprio conhecimento. Neste contexto, não cabe mais a figura do professor transmissor de conhecimento e manipulador de todas as situações, mas sim o facilitador que irá auxiliar os aprendizes em seu processo de aprendizagem.

Problema:
Como apoiar as formas de interação do Facilitador em um ambiente de aprendizagem cooperativa baseada em projetos?

Forças:
O facilitador é alguém que entende os processos desenvolvidos pelo grupo (sociais e de trabalho) e por isso pode ajudar o grupo a entender seus problemas, tanto de ordem social, quanto pedagógica, e encontrar soluções para eles. O facilitador observa e critica as ações dos outros participantes, responde a solicitações de ajuda, aconselha o Coordenador do grupo e guia a atuação dos Aprendizes na solução dos problemas.
O facilitador está presente em todas as tarefas em ambientes de aprendizagem cooperativa baseada em projetos. Porém, com o conhecimento sobre o grupo e o processo de trabalho, ele pode ajudar a aumentar a coesão do grupo e o estabelecimento das regras de funcionamento do grupo. Na medida em que os membros do grupo interagem, a figura do facilitador tende a se tornar menos importante, pois o grupo seguirá os seus caminhos, sofrendo eventuais intervenções do facilitador.
Por sua natureza de sujeito central, o facilitador se relaciona com todos os outros papéis presentes no ambiente.

Solução:
As políticas de coordenação e de cooperação definem as formas de interação do facilitador no contexto do ambiente de aprendizagem:

- Políticas de coordenação: O facilitador deve ter acesso de leitura a todos os objetos e espaços compartilhados pelo grupo em todos os momentos do desenvolvimento do projeto. Porém, o acesso à edição deve ser restrito a objetos relacionados ao Processo Avaliação.
- Políticas de cooperação: O facilitador deve ser capaz de reconhecer quando um problema pedagógico ou de relacionamento está se desenvolvendo dentro do grupo e deve ter habilidade e conhecimento para ajudar o grupo a lidar com isso. Para isto, o facilitador deve gerar intervenções que podem interromper ou não o processo para tratar as questões, provendo sugestões sobre a forma de resolver o problema.

Para apoiar a implementação destas políticas defina elementos de interface e de percepção apropriados.
Defina elementos de interface que permitam:

- Envio de ajuda (que podem se traduzir em mensagens estruturadas aos participantes com sugestões para alguma questão específica, ou apoio a uma ação do participante de acordo com a Representação Conhecimento).
- Criação de objetos para auxílio ao trabalho do grupo dependendo da atividade proposta (por exemplo, listas de referências, templates de relatórios).
- Ações de intervenção sobre as atividades em desenvolvimento (por exemplo, início, término, interrupção).

Defina mecanismos de percepção que permitam a identificação e conhecimento de:

- Informação sobre o Fluxo Atividades.

- Informação sobre os participantes (Percepção Interação Social): dados pessoais e dados sobre as responsabilidades nas tarefas individuais e de grupo.

Usos conhecidos:
Em estudos realizados com o ambiente CoWeb, (Guzdial et al., 2000) identificaram o papel do professor ou facilitador, cuja função é ajudar os estudantes a se engajarem nas atividades propostas. Para isto, ele pode criar páginas de discussão, de revisão e crítica e páginas onde os estudantes depositam seus trabalhos (objetos compartilhados do ambiente). Além disso, foi implementado um mecanismo específico de navegação para o facilitador (interface).
No Ambiente Algebra Jam (Singley et al., 1999), são designados explicitamente cinco papéis: Observador, Aprendiz, Especialista, Líder (Coordenador) e Guia (Facilitador). O facilitador observa e acompanha as ações dos outros participantes, responde a pedidos de ajuda, e guia os aprendizes na solução dos problemas propostos. O facilitador pode enviar mensagens estruturadas respondendo a pedidos de ajuda, fornecendo informações relevantes ao processo ou dando início a uma tarefa. Além disso, possui elementos de interface específicos para realização de suas tarefas.
Em uma configuração do ambiente Kansas (Smith, Hixon e Horan, 1998) que apoia o paradigma de aprendizagem DTVI (Distributed Tutored Video Instruction), é proposta a definição explícita de quatro papéis: estudante, facilitador, coordenador e desenvolvedor. Para cada um deles é apresentada uma interface diferente conforme suas funções específicas. No caso do facilitador, são disponibilizadas ações para dar início e término nas discussões, salvar anotações dos estudantes em páginas Web (objeto compartilhado) e controlar os mecanismos de volume de áudio de todos os participantes.
O ambiente Zebu (Tiessen e Ward, 1999) prevê as seguintes funções para o facilitador (professor): criação de templates para que os estudantes criem suas páginas no contexto das atividades do projeto, criação de listas de recursos a serem utilizados pelos estudantes no projeto (objetos de auxílio), resposta a questões levantadas pelos estudantes e coordenação das atividades através da configuração e acompanhamento do processo de trabalho.

14. Nome: Aprendiz

Contexto:
Em ambientes de aprendizagem, o papel principal é o do aprendiz que é o sujeito que interage com outros sujeitos e com conteúdos, com o objetivo de adquirir algum tipo de conhecimento ou habilidade.

Problema:
Como apoiar as formas de interação do Aprendiz em um ambiente de aprendizagem cooperativa baseada em projetos?

Forças:
O aprendiz é o papel central em ambientes de aprendizagem cooperativa baseada em projetos. O aprendiz tem responsabilidade pela gerência e execução das tarefas para alcance das metas e resolução dos problemas definidos no projeto. Tem como objetivo participar na construção coletiva de conhecimento tendo como benefício pessoal seu aprendizado sobre as áreas de pesquisa do projeto.
O aprendiz está presente em todas as tarefas de projetos em ambientes de aprendizagem cooperativa e é o sujeito principal do desenvolvimento das tarefas e da geração dos produtos. O aprendiz se relaciona com todos os outros papéis presentes no ambiente.

Solução:
As políticas de coordenação e de cooperação definem as formas de interação do aprendiz no contexto do ambiente de aprendizagem:

- Políticas de coordenação: O aprendiz deve ter acesso de leitura e escrita a todos objetos compartilhados nas atividades às quais participa. O ambiente deve estimular o maior número possível de acessos de aporte de informações e contribuições de todos os aprendizes.
- Políticas de cooperação: Os aprendizes devem ter disponíveis vários meios para interagirem entre si: troca de mensagens, fóruns de discussões assíncronos e conversas síncronas. Todas as atividades devem permitir estas interações, mesmo que cada participante tenha uma tarefa específica (por exemplo, escrever uma seção de um documento coletivo). O grupo deve definir em todas as situações uma política de troca de informações antes, durante e depois de cada atividade (por exemplo, na construção coletiva de um texto, (i) participantes discutem o tema; (ii) participantes escrevem partes do texto, consultando os outros membros; (iii) participantes promovem um debate sobre o produto final gerado).

Para apoiar a implementação destas políticas defina elementos de interface e de percepção apropriados.
Defina elementos de interface que permitam:

- Envio de pedido de acesso e ajuda ao Facilitador e a outros Aprendizes (que podem se traduzir em mensagens estruturadas de acordo com a Representação Conhecimento).
- Acesso individual ao Coordenador da atividade na qual está participando (se este papel existir).

Defina mecanismos de percepção que permitam a identificação e conhecimento de:

- Informação sobre o Fluxo Atividades.
- Informação sobre os participantes (Percepção Interação Social): dados pessoais e dados sobre as responsabilidades nas tarefas individuais e de grupo.

Usos conhecidos:
O papel de aprendiz está implícito em qualquer ambiente de aprendizagem cooperativa apoiada por computadores, mesmo que este não seja definido formalmente.
No Ambiente Algebra Jam (Singley et al., 1999), são designados explicitamente cinco papéis: Observador, Aprendiz, Especialista, Líder (Coordenador) e Guia (Facilitador). O Aprendiz assume a responsabilidade pela performance de determinadas tarefas para a solução do problema proposto. O Aprendiz pode enviar mensagens estruturadas de pedidos de ajuda ou fornecendo informações relevantes ao processo. Além disso, possui elementos de interface específicos para realização de suas tarefas, tais como, preenchimento de tabelas ou cálculo de médias.
Em uma configuração do ambiente Kansas (Smith, Hixon e Horan, 1998) que apoia o paradigma de aprendizagem DTVI (Distributed Tutored Video Instruction), é proposta a definição explícita de quatro papéis: estudante, facilitador, coordenador e desenvolvedor. Para cada um deles é apresentada uma interface diferente conforme suas funções específicas. No caso do estudante ou aprendiz, a disponibilização de ações se limita à escrita em um editor cooperativo e controle de seu próprio mecanismo de volume de áudio.

15. Nome: Critérios Nomeação

Contexto:
Ambientes de aprendizagem cooperativa baseada em projetos têm como característica a designação de Papéis, explícita ou não, aos participantes das interações na realização das tarefas. A associação de papéis aos participantes (atores) ao longo do processo pode ser feita de diferentes formas, de acordo com os objetivos do trabalho a ser desenvolvido e com o que se deseja observar no grupo.
Smith, Hixon e Horan (1998) definem duas formas de nomeação de papéis: prescritiva, onde os papéis são designados explicitamente pelo sistema ou por algum outro participante que desempenhe o papel de coordenador; e não prescritiva, onde os papéis não são designados explicitamente.

Problema:
Que critérios devem ser adotados na nomeação de papéis aos membros de um grupo em um ambiente de aprendizagem cooperativa baseada em projetos?

Forças:
A designação ou nomeação de papéis em ambientes cooperativos contribui para o sucesso do desenvolvimento de um projeto, ou seja, os atores certos para papéis certos influenciam no cumprimento dos objetivos de um grupo.
Designar papéis a usuários significa definir quais serão as suas funções no contexto das atividades do projeto cooperativo. Portanto, a designação de papéis a pessoas/grupos/agentes/programas em ambientes de aprendizagem cooperativa deve ser feita segundo critérios que demonstrem a coerência com os objetivos educacionais desejados.
Em muitos casos, encoraja-se a troca ou alternância de papéis para minimizar as críticas e conflitos dentro dos grupos. No caso de ambientes de aprendizagem, esta prática pode trazer benefícios no sentido do exercício de responsabilidades.

Solução:
Os critérios para a nomeação de papéis devem diretamente relacionados aos objetivos educacionais das tarefas propostas no contexto do ambiente.
Portanto, utilize a forma prescritiva de nomeação nos seguintes casos:

- um dos objetivos do ambiente é o aprendizado do trabalho em equipe, ou seja, os participantes não têm experiência de trabalho em equipe e formação de grupos;
- um dos objetivos do ambiente é explorar as habilidades pessoais dos participantes envolvidos já conhecidas anteriormente através da análise de conhecimentos prévios.

Neste caso, defina restrições de interface da ferramenta de acordo com o papel (as interfaces específicas são definidas nos padrões relativos a cada papel).
Utilize a forma não prescritiva de nomeação nos seguintes casos:

- quando se deseja observar o surgimento espontâneo de habilidades no grupo;
- quando as atividades não demandam diferentes papéis obrigatoriamente, por exemplo, uma pesquisa para levantamento de material de apoio para o grupo.

Neste caso, todos os elementos de interface da ferramenta são habilitados para todos, ou seja, a interface é a mesma para todos os usuários, que vão assumir papéis espontaneamente.

Usos conhecidos:
Na experiência de Kynigos (1999) com um ambiente cooperativo voltado para o ensino de matemática, uma vez que desejava-se reproduzir uma situação de trabalho em sociedade, onde cada indivíduo tem sua função, foram adotadas as práticas de nomeação explícita de papéis para todas as atividades previstas no projeto. Ao longo do desenvolvimento do trabalho, esta nomeação foi revista e modificada. A prática de revezamento e acúmulo de papéis ajudou a minimizar os conflitos na execução das tarefas propostas.
No Algebra Jam (Singley et al., 1999), os participantes podem ter os papéis explicitados ou o exercício dos papéis pode surgir espontaneamente sendo reconhecido pelo sistema através de ações associadas a elementos de interface pré-definidos. Além disso, é encorajado o revezamento de papéis para que todos os participantes possam vivenciar as responsabilidades do exercício de cada função.
Em uma configuração do ambiente Kansas (Smith, Hixon e Horan, 1998) que apoia o paradigma de aprendizagem DTVI (Distributed Tutored Video Instruction), são realizadas sessões de videotape de aulas, onde os alunos se vêem através de video-links e fazem anotações colaborativamente utilizando uma ferramenta de edição de texto. De acordo com o objetivo educacional de aprendizado do tema em questão, o sistema propõe a definição explícita de quatro papéis: estudante, facilitador, coordenador e desenvolvedor. Para cada um deles é apresentada uma interface diferente conforme suas funções específicas.

16. Nome: Percepção Interação Social

Contexto:
Em um ambiente de aprendizagem cooperativa apoiado por computadores, a interação entre os membros do grupo é o principal fator para que ocorra cooperação. Sem interação, não se estabelecerá o nível de cooperação desejável para estimular o processo de aprendizagem.
Os participantes envolvidos em uma situação de aprendizagem que requer interação devem estar conscientes sobre com quem estão interagindo, e de como se dará a interação. Para isto, é preciso que estejam disponíveis informações sobre aspectos relacionados à interação social entre os indivíduos.

Problema:
Que elementos devem ser disponibilizados em um ambiente de aprendizagem cooperativa apoiada por computadores para facilitar e estimular a interação social entre os membros do grupo?

Forças:
A percepção sobre as conecções sociais dentro de um grupo induz e facilita as interações entre seus membros.
O conhecimento e expectativa corretos em relação às outras pessoas e seus trabalhos trazem segurança para execução coletiva das tarefas e faz com que fique mais claro o nível de contribuições de cada um.
De acordo com Araujo (2000), os usuários de um groupware devem ser capazes de reconhecer o grupo no qual estão inseridos para interagirem. Isto significa obter informações sobre seus participantes capazes de ajudar o estabelecimento de conexões sociais. Araujo (2000) enumera as seguintes informações relativas à percepção social em um groupware: composição, localização, presença, proximidade (papéis e responsabilidades), disponibilidade e emoção. Em resumo, “as pessoas precisam perceber e ter acesso umas às outras dentro de um mesmo contexto”.

Solução:
Mantenha um componente de percepção social no sistema, que possa ser acessado a qualquer momento, por qualquer participante e que apresente as seguintes informações atualizadas de forma dinâmica:

- Dados pessoais de todos os participantes – localização física, formação, preferências, expectativas, interesses e disponibilidade e recursos para acesso. Estes dados provêm do cadastro de usuários do sistema.
- Alocação de tarefas comuns – sub-grupos dos quais participa (ressaltando aqueles em comum com o usuário que consulta esta informação). Estes dados provêm da definição do Fluxo Atividades.
- Atividade(s) e papéis que está alocado no momento. Estes dados provêm da consulta ao status do Fluxo Atividades.
- Presença no ambiente no momento. Este dado provém do registro de login do ambiente.

A forma mais comum de apresentação deste tipo de informações são as listas de usuários bastante difundidas em sistemas cooperativos. As listas de usuários variam de formas bem simples, contendo apenas informações sobre os participantes "logados" no ambiente em um determinado momento, até as mais sofisticadas que incluem informações em vídeo sobre a disponibilidade de cada membro do grupo (GroupLab, 2000).

Usos conhecidos:
Entre os exemplos de ambientes que implementam componentes de percepção social encontram-se o Algebra Jam (Singley et al., 1999) onde a disponibilidade dos membros do grupo é apresentada sob a forma gráfica (fotos) na interface da tela principal do ambiente. A partir desta representação, o grupo sabe quem está interagindo em determinado momento e como acessar estas pessoas.
O LiNC Virtual School (Isenhour et al., 2000) implementa na tela de abertura do sistema uma lista de usuários pertencentes ao grupo, contendo informações pessoais, projetos dos quais participa e últimas tarefas realizadas.
A ferramenta de discussão Lead Line (Farnham et al., 2000) disponibiliza uma lista de usuários com informações relativas aos papéis desempenhados por cada um deles em uma sessão interativa.
O ambiente Zebu (Ward e Tiessen, 1997) permte importar arquivos com informações sobre um usuário em seu cadastro. Além disso, associa cada usuário a projetos específicos, refletindo as contribuições dos participantes nas páginas de cada projeto, ou seja, consultando um projeto, informações sobre os membros de seu grupo são automaticamente apresentadas.

17. Nome: Percepção Tarefa

Contexto:
Em um ambiente computacional para aprendizagem cooperativa, tarefas são propostas para um grupo, que deverá realizá-las de forma cooperativa. O entendimento por parte dos membros do grupo sobre como os processos deverão ocorrer e o seu acompanhamento, ou seja, informações de diversos tipos sobre as atividades, tornará possível e facilitará seu desenvolvimento.
Araujo (2000) afirma que a qualidade do produto final de uma interação cooperativa depende do grau de consciência de seus participantes sobre os objetivos e a estruturação do trabalho que irão realizar.

Problema:
Que elementos devem ser disponibilizados em um ambiente de aprendizagem cooperativa baseada em projetos para apresentar as informações necessárias sobre uma determinada tarefa a ser desenvolvida pelo grupo?

Forças:
A falta de informações sobre os objetivos e os conhecimentos necessários para realização de uma tarefa pode levar a erros na sua execução. O desenvolvimento coletivo de uma atividade requer integração entre os participantes, e para isso, é preciso que os participantes estejam bastante conscientes dos passos a serem dados para o cumprimento dos objetivos, e do papel de cada um dentro deste processo. Cada membro do grupo deve estar atento ao processo como um todo.
A Coordenação será mais bem realizada se todos estiverem conscientes dos papéis/responsabilidades de cada participante.
Araujo (2000) descreve os dois tipos de informação relevantes ao entendimento das tarefas em um groupware: estrutura (formas e procedimentos) e status (passado, presente e futuro). As pessoas precisam saber o que elas têm que fazer, o que os outros têm fazer e como sua atuação está inserida no contexto do projeto.

Solução:
Disponibilize um componente de percepção de tarefa que apresente as seguintes informações para os participantes de um projeto:

- quais os conhecimentos prévios necessários para realização desta tarefa;
- qual a estrutura da tarefa - passos a serem seguidos para realizá-la;
- qual o tempo necessário ou determinado para sua realização;
- como se dará a participação de cada membro do grupo (papéis e especializações);
- quais os resultados esperados;
- quais as ferramentas disponíveis para utilização como apoio;
- status de sua execução.

Estas informações provêm da definição e do status de execução do Fluxo Atividades.
Este componente pode ser similar aos utilizados em Sistemas de Workflow, que normamente apresentam informações sobre percepção das tarefas a serem desenvolvidas pelos participantes de um projeto através de visões gráficas do processo, onde se representa a estrutura e o status das atividades; e listas de trabalho, que mostram aos participantes as atividades relacionadas a eles e as informações pertinenentes a elas (Araujo, 2000).

Usos conhecidos:
Nos sistemas descritos na literatura não se encontram componentes de percepção de tarefas específicos, porém, na maioria dos casos as informações encontram-se apresentadas nas próprias intefaces dos ambientes. Desta forma, a validação do padrão se refere somente às informações sugeridas na solução.
No ambiente PIE (Gifford e Enyedy, 1999), é apresentado um conjunto de botões que representam a estrutura (fluxo) da atividade. Para cada atividade, existe uma descrição textual dos procedimentos a serem realizados.
No ambiente CLARE (Wan e Johnson, 1994), através de um menu está indicada a sequência de atividades propostas (estrutura do fluxo), que compõem o processo. No contexto de cada elemento do menu, encontram-se interfaces com a definição de cada tarefa a ser realizada.

18. Nome: Percepção Conceitos

Contexto:
Em um ambiente de aprendizagem cooperativa apoiado por computadores, geralmente são propostas aos alunos tarefas que requerem algum tipo de conhecimento prévio e a apropriação de novos conhecimentos para serem cumpridas. Ao rever seus conhecimentos, buscar novos conceitos e compartilhar com outros membros de um grupo, o aluno se encontra em um processo de aprendizagem ativa. Porém, nem sempre fica claro que conceitos devem ser revisados para executar a tarefa.

Problema:
Como facilitar a percepção sobre conhecimentos necessários à execução de uma tarefa em um ambiente de aprendizagem cooperativa baseada em projetos?

Forças:
Segundo Gutwin et al. (1995), existem alguns questionamentos que um Aprendiz pode se fazer ao deparar-se com uma tarefa a ser executada:

- que conhecimento prévio está relacionado a esta tarefa? (Percepção Tarefa)
- o que mais é preciso ser descoberto sobre este tópico?
- é preciso revisar idéias pré-existentes a partir das novas informações?
- é possível criar hipóteses baseadas no conhecimento atual do grupo para prever o resultado da tarefa?

Os sistemas apresentam, em geral, estruturas para Representação Conhecimentos compartilhados, que pode ou não estar relacionada às tarefas desenvolvidas.
Araujo (2000) afirma que "perceber" uma interação com o uso de um groupware envolve compreender o que se passa durante esta interação e, a partir desta compreensão, cada usuário pode estabelecer o contexto e impacto de suas atividades e contribuições individuais em relação à atividade do grupo.

Solução:
Forneça meios para que o Facilitador ou Coordenador possam incluir links de referências relacionados a uma determinada tarefa. Torne explícito os links que apresentam conceitos básicos e avançados sobre o conteúdo.
Além disso, permita que os Aprendizes incluam seus próprios links na medida que tenham amadurecido o conhecimento sobre o conteúdo. Para isto, inclua etapas no Processo Avaliação onde são apresentados Resultados Grupo que demonstrem ao grupo a sua evolução no processo de construção de conhecimento sobre o tema.

Usos conhecidos:
O ambiente Belvedere (Suthers e Weiner, 1995) define uma sequência de fases da pesquisa científica (fluxo de atividades proposto) e provê links para informações diversas a serem utilizadas em cada uma destas etapas, tais como fundamentação do problema, hipóteses já levantadas por cientistas, e sugestões de experimentos. Além disso, permite o relacionamento direto de informações coletadas pelos alunos ao produto gerado por eles na atividade (no caso, diagramas de representação de teorias científicas).
Os ambientes CaMILE (Guzdial, 1997) e CSILE (Oshima, 1997) permitem a inclusão de links de referências na internet para serem consultados pelos usuários no processo de construção de conhecimento através de discussão em fóruns.

O Zebu (Tiessen e Ward, 1999) provê ao professor a possibilidade de fornecer recursos aos alunos para execução de suas tarefas. Isto é feito através da criação de links de referências nos templates e páginas disponibilizados por eles, e desta forma, relacionam a informação à tarefa a ser realizada. A medida que os alunos vão se tornando mais amadurecidos para a busca de informações, o professor pode delegar esta tarefa a eles, que vão usar então os mesmos mecanismos utilizados pelo professor.

19. Nome: Percepção Espaço Trabalho

Contexto:
Ambientes de aprendizagem cooperativa permitem que estudantes trabalhem juntos, compartilhando espaços virtuais. Estes ambientes não conseguem reproduzir todas as nuances que ocorrem em uma situação de interação face a face onde os participantes utilizam canais de comunicação explícita, por exemplo, a fala; e implícita, por exemplo, expressões corporais.
Por isso, os ambientes de aprendizagem cooperativa baseada em projetos devem prover meios de disponibilizar informações que auxiliem os participantes a terem consciência do quê estão compartilhando, como e com quem.

Problema:
Que elementos devem ser disponibilizados em um ambiente de aprendizagem cooperativa baseada em projetos para garantir a percepção do espaço de trabalho entre os membros do grupo?

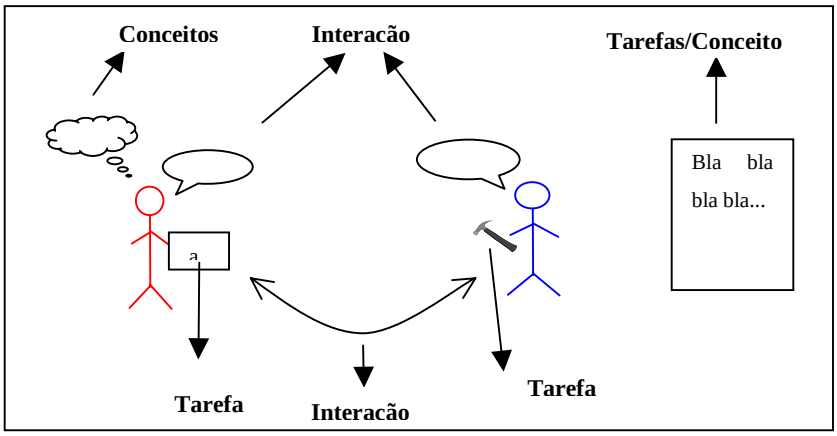
Forças:
A percepção do espaço de trabalho reduz o overhead do trabalho em grupo, permitindo uma interação mais natural e efetiva, e facilita o engajamento dos estudantes em práticas que levam à aprendizagem cooperativa.
A noção do que está acontecendo no grupo como um todo encerra o verdadeiro conceito de aprendizagem cooperativa.
Percepção é um conceito relacionado a mecanismos que garantem que as pessoas podem compreender ou tomar consciência do próprio processo e da interação entre todos os participantes em um ambiente cooperativo. Elementos de percepção são essenciais para que os estudantes possam aprender e trabalhar em equipe.
Em um ambiente computacional para aprendizagem cooperativa, a percepção de cada participante em relação aos outros é um dos elementos chave para propiciar uma interação efetiva (Percepção Interação Social). O fato de um indivíduo saber o que o outro está fazendo em determinado momento pode levá-lo a buscar um contato e então possibilitar trocas.
O acesso a informações sobre contribuições e tarefas já completadas (Percepção Tarefa) também é um fator importante, pois pode aproximar pessoas com interesses comuns dentro do grupo, mesmo que não estejam produzindo alguma coisa juntas.
Prover elementos de percepção envolve pelo menos três aspectos: a própria informação, a tradução da informação, e apresentação da informação. Araujo (2000) aponta as seguintes informações como necessárias para prover percepção do espaço de trabalho: status dos objetos de trabalho (quais são, como encontrá-los e qual o histórico de sua evolução), ações e posição dos participantes.

Solução:
Apresente uma representação dos membros do grupo dentro do espaço de trabalho, de forma que o grupo todo possa ter as seguintes informações:

- onde ele está;
- o que está fazendo neste momento;
- o que já fez anteriomente.

Esta representação pode ser gráfica, icônica, através de janelas, ou de realidade virtual. Muitas vezes é feita simplesmente através da definição de diferentes cores para os usuários, onde estas vão indicar as contribuições nos objetos compartilhados, os objetos que estão sendo trabalhados em um determinado momento, a posição do cursor em uma janela compartilhada, etc. A seguir é apresentado um esquema representativo do Espaço de Trabalho:

Espaço de Trabalho em Ambientes de Aprendizagem Cooperativa



Usos conhecidos:
Os ambientes CSILE (Oshima, 1997), Collaboratory Notebook e CaMILE (Guzdial, 1997) implementam percepção de espaço de trabalho através da noção de mensagem estruturada, onde o autor é representado por suas iniciais, e portanto os outros membros conseguem perceber o que ele está fazendo ou já fez.
No ambiente NICE (Roussos et al., 1997), os participantes de uma seção de trabalho são representados por avatars e todos podem visualizar seus movimentos e suas ações dentro do espaço de trabalho.
No ambiente CLARE (Wan e Johnson, 1994), o modelo de interação é previsto em duas fases do trabalho, onde deve acontecer interação direta entre os participantes: Argumentação e Consolidação. Em ambas as fases, as pessoas visualizam o artefato que está sendo analisado em uma janela (tarefa), e em outra são apresentadas as argumentações das pessoas em relação a este artefato. Através desta janela, as pessoas trocam informações e podem votar por uma decisão final. Os objetos manipulados por cada membro do grupo aparecem relacionados aos seus nomes.

20. Nome: Coordenação

Contexto:
Coordenação é um termo utilizado para descrever um conjunto de mecanismos disponíveis em um ambiente compartilhado, que têm como função gerenciar a Interdependência entre os participantes. Os mecanismos de coordenação garantem que os procedimentos serão praticados no ambiente compartilhado de acordo com regras pré-definidas pelos participantes, ou impostas pelo próprio ambiente.

Em ambientes de aprendizagem cooperativa, que têm como característica um objetivo educacional, a coordenação pode ser traduzida em suporte à definição das formas de trabalho, permitindo que todos tenham acesso ao conhecimento compartilhado e possam desenvolver habilidades cooperativas.

Problema:
Quais são os mecanismos necessários para prover suporte à coordenação em ambientes de aprendizagem cooperativa apoiados por computadores?

Forças:
Coordenação está relacionada ao suporte, à definição e execução das tarefas do grupo, e individuais de cada participante. Na definição das tarefas, estabelecem-se as regras de procedimento. Na execução das tarefas, necessita-se de assistência, tanto em nível de instrumentos, quanto de informações e conceitos. Muitos sistemas cooperativos provêem guias para estruturação das interações sociais no contexto do espaço de trabalho compartilhado (Farnham et al., 2000).
Os ambientes de aprendizagem cooperativa baseada em projetos possuem a particularidade de que o comportamento dos indivíduos é influenciado pelo grupo como um todo, ou seja, o grupo tem sua identidade própria. Os Aprendizes trabalham juntos para solucionar problemas. Desta forma, existem dois aspectos envolvidos: o processo cooperativo de trabalho e os conhecimentos compartilhados na busca da solução.
O ambiente deve permitir o estabelecimento de regras de cooperação e de procedimentos entre os indivíduos.
A Interdependência entre os participantes, necessária para configurar e operacionalizar o ambiente compartilhado não deve prejudicar a autonomia individual, pois esta é condição fundamental no processo de aprendizagem.
O ambiente deve fornecer ajuda aos participantes no sentido de que desenvolver uma tarefa significa também adquirir, compartilhar ou trabalhar na construção de algum tipo de conhecimento.
De acordo com Johnson-Lenz e Johnson-Lenz (1991), outro aspecto da coordenação está relacionado a formas de manter o grupo estimulado, tais como convidar à participação, marcar os eventos do processo de cooperação e definir um ritmo aos trabalhos e encontros. A coordenação, neste sentido, pode ser feita de forma livre, ou seja, não há coordenação, o grupo se auto-regula; centralizada, ou seja, existe um indivíduo que atua como coordenador, tendo responsabilidades pela manutenção do ambiente e ajuda aos participantes; automática, ou seja, o sistema exerce a coordenação; ou semi-automática, o sistema possui algumas funções e um coordenador outras.
Observa-se que na maioria dos ambientes de aprendizagem cooperativos, o professor ou Facilitador exerce o papel de Coordenador geral do projeto, mesmo ele não tenha este papel atribuído de forma explícita. Informalmente, cabe a ele as responsabilidades de manter o grupo no processo de cooperação e gerenciar as negociações e Resolução Conflitos e Tomada Decisões. Percebe-se claramente isto em ambientes como Zebu (Tiessen e Ward, 1999) e WebGuide (Stahl, 1999).
Os sistemas cooperativos oferecem diversos mecanismos para atribuição de direitos de acesso a objetos compartilhados e associação de privilégios aos participantes. Estas ações estão diretamente relacionadas aos procedimentos e responsabilidades estipulados no Fluxo Atividades.

Barros (1995) afirma que um mecanismo de suporte importante neste sentido é o de notificação de eventos. Por exemplo, o sistema avisa aos participantes sobre entrada/saída de um indivíduo no ambiente cooperativo ou pedidos de acesso aos objetos, facilitando o trabalho de coordenação.

Solução:
O ambiente deverá garantir que todos os participantes compartilhem conhecimento e se envolvam no processo cooperativo. Para apoiar o trabalho de coordenação realizado pelo Facilitador, disponibilize Guias aos participantes das interações no ambiente cooperativo.
Os guias são mecanismos de ajuda que devem observar diretamente a atuação dos indivíduos e do grupo no contexto do trabalho que está sendo realizado, relacionando os objetivos educacionais aos conceitos manipulados, de forma a prover ajuda sobre os passos a serem tomados. Os guias podem analisar as interações realizadas, interpretando as mensagens trocadas de acordo com o nível de profundidade da Representação Conhecimento. Esta poderá permitir análise desde a estrutura das mensagens, até seu conteúdo.
Desta forma, os guias deverão atuar com as seguintes responsabilidades:

- Análise: identificação e validação do conhecimento que cada estudante ou grupo de estudantes está apresentando (conteúdo); análise do caminho que está sendo percorrido pelo grupo para busca da solução do problema (estrutura das interações);
- Retorno ou notificação: apresentação de realimentação individual e ao grupo; sugestão das próximas ações.

Estes guias são geralmente implementados nos ambientes como tutores inteligentes ou agentes.

Em relação à padronização de procedimentos, o ambiente deverá apresentar mecanismos de coordenação comuns aos ambientes CSCW: controle de acesso a recursos compartilhados e controle de tarefas.
O controle de acesso se refere à organização do acesso a recursos compartilhados pelos usuários, para resolver conflitos de concorrência.
Pode ser feito segundo várias abordagens diferentes (Dias, 1998): bloqueio simples; mecanismos de transações; protocolos para controle de piso; controle centralizado; detecção de dependência; execução reversível; transformações de operações e mecanismos de check in e check out.
O ambiente deve prover suporte para o controle de tarefas, através da disposição de mecanismos para:

- especificação de como a interação se realizará;
- definição de regras de conduta, procedimentos e limites;
- definição de papéis e responsabilidades;
- acompanhamento da execução das tarefas.

Usos conhecidos:
No ambiente Belvedere (Suthers e Weiner, 1995), um conselheiro ou guia inteligente ajuda os estudantes a focalizarem em aspectos particulares das questões estudadas, sugerindo formas nas quais um diagrama de argumentação pode ser estendido ou melhorado, destacando objetos que possivelmente precisam de atenção, e oferecendo dicas baseadas em princípios tais como consistência, suporte empírico, maximização da cobertura de uma teoria e consideração a teorias.

No ambiente ARCOO, uma sala de estudos metafórica possui entre outros recursos "auxiliares invisíveis", que ajudam os estudantes a desenvolverem seus projetos (Barros e Borges, 1995).
No Algebra Jam (Singley et al., 1999), um tutor inteligente modela uma equipe e as interações entre os seus membros. Isto é feito através da observação de alguns eventos produzidos por papéis pré-definidos. A observação da ocorrência destes eventos atualiza as crenças do tutor sobre a proficiência dos estudantes no domínio estudado, e sobre seus progressos em habilidades cooperativas.

21. Nome: Resolução Conflitos e Tomada Decisões

Contexto:
Durante as sessões de aprendizagem cooperativa, podem surgir conflitos entre os membros do grupo, acarretando problemas na execução das tarefas. Portanto, é necessário que o ambiente de aprendizagem ofereça formas de apoiar a resolução de conflitos.

Problema:
O que representa e como pode ser apoiado o processo de negociação para a resolução de conflitos e tomada de decisões em ambientes de aprendizagem cooperativa baseada em projetos?

Forças:
Segundo a teoria socio-construtivista, o conflito entre aprendizes é uma extensão do conceito piagetiano de conflito entre as crenças do aprendiz e e suas ações no mundo. Quando ocorre um conflito entre pares, fatores sociais previnem os aprendizes de ignorar o conflito e os força a encontrar uma solução. Para Dillenbourg e Schneider (1995), mais do que isso, uma simples desavença ou mal-entendimento pode ter o mesmo efeito que conflitos explícitos, e interações verbais desencadeadas para resolver um conflito estão diretamente relacionadas a resultados de aprendizagem.
No contexto de ambientes de aprendizagem cooperativa, a negociação é um mecanismo auxiliar relacionado à Coordenação, que leva os Aprendizes à tomada de decisões sobre o planejamento e a execução das tarefas, que por sua vez levarão à elaboração da solução dos problemas propostos, promovendo a aprendizagem. Negociar significa argumentar e decidir. Neste tipo de interação, as pessoas possuem opiniões e desejam que os outros as aceitem. Este processo envolve vários mecanismos cognitivos e afetivos, tais como, lógica, inferência, dedução, crença, dúvida, sutileza e envolvimento emocional com o assunto e com os participantes (Barros, 1995).
O processo de tomada de decisão envolve a definição e análise das diferentes alternativas de ação propostas pelos membros do grupo, a identificação de um conjunto de alternativas promissoras para a execução da tarefa cooperativa, a seleção e implementação de uma alternativa e verificação do comportamento da alternativa selecionada na execução da tarefa. Este processo é importante tanto para o desenvolvimento cognitivo dos alunos quanto para o aprimoramento das habilidades sociais.
A cooperação requer um espaço de compartilhamento de idéias e de negociação de cursos de ação. Os conflitos são uma parte inerente do processo de cooperação. Se bem empregados, eles podem aumentar a produtividade do grupo e a aprendizagem. No entanto, os conflitos podem refletir desavenças pessoais e enfraquecer a coesão do grupo. Além disso, armazenar informações sobre o processo de negociação é mais um fator importante para a Memória do grupo.

Solução:
Disponibilize um espaço de negociação síncrono ou assíncrono, onde os participantes possam interagir através de um modelo de argumentação.
Stahl (1999) afirma que colaboração requer divergência (surgimento de idéias) e convergência (negociação, síntese e consenso). Por isso, em um modelo de argumentação voltado para a colaboração deve ser flexível para permitir que uma contribuição ou nota possa ser relacionada a mais de uma outra na hierarquia da estrutura.
No espaço de negociação permita a definição de um agente humano ou computacional com a função de mediador de conflitos. Este agente pode ser responsável pela categorização dos tópicos da discussão, caso o grupo não seja experiente para fazer isso.

Usos conhecidos:
Os Sistemas de Suporte à Decisão em Grupo utilizando modelos estruturados foram umas das primeiras aplicações no domínio de trabalho cooperativo (Barros, 1995).
O ambiente ARCOO (Barros & Borges, 1995) oferece o sub-sistema de Socialização, que visa gerenciar os encontros entre os aprendizes através de Reuniões, Conferências e Conversas, dando suporte, através de ferramentas que utilizam modelos estruturados de argumentação, para a negociação e superação dos conflitos.
Tedesco & Self (2000) desenvolveram um mediador artificial - MArCo, que detecta os conflitos meta-cognitivos e sugere cursos de ação. O mediador opera a partir de um modelo de conflitos, baseado em uma rede de crenças sobre o comportamento dos pares cooperantes.
O framework Habanero oferece The Voting Tool, uma ferramenta de votação que visa ajudar os membros do grupo a tomarem decisões e a superarem conflitos.
No ambiente WebGuide (Stahl, 1999), existe a previsão de implementação de um componente de negociação assíncrona para permitir aos estudantes submeterem suas perpectivas pessoais sobre um determinado tópico em discussão para esta seja votada. Uma vez tendo recebido um número suficiente de votos, esta proposta passa a ser incorporada, passando a representar um conhecimento aceito pelo grupo.
O ambiente PENCACOLAS (González et al., 1997) foi desenvolvido para apoiar a aprendizagem da produção de documentos de forma colaborativa, dando suporte a todas as fases que acontecem durante este processo, entre elas a geração de idéias (brainstorming). Nesta etapa, os participantes têm suporte à colocação de suas idéias para posteriormente optarem por aqueles que serão incorporadas ao produto final do grupo.

Agradecimentos

Os autores agradecem especialmente ao professor Gustavo Rossi por sua contribuição como guia imprescindível à realização deste trabalho com sugestões e comentários enriquecedores.

Referências Bibliográficas

Araujo, R.M. (2000) Ampliando a Cultura de Processos de Software – Um Enfoque Baseado em Groupware e Workflow. Tese de Doutorado. COPPE/UFRJ Programa de Sistemas e Computação.

Aronson, E. (1978). “The Jigsaw Classroom”. Bervely Hills, CA: Sage.

Barros, L.A.; Borges, M.R.S. (1995) ARCOO - Sistema de Apoio à Aprendizagem Cooperativa Distribuída”, Anais do VI Simpósio Brasileiro de Informática e Educação.

Becker, K.; Zanella, A.N. (1998). A Cooperation Model for Teaching/Learning Modeling Disciplines. Anais do International Workshop on Groupware - CRIWG’98, Rio de Janeiro, Brasil.

Conklin, J. E. (1996) Capturing Organizational Memory. Group Decision Support Systems, Inc. http://www.gdss.com

Dias, M.S. (1998) "COPSE - Um Ambientes de Suporte ao Projeto Cooperativo de Software". Tese de Mestrado. COPPE/Sistemas, UFRJ.

Dillenbourg, P., Schneider, D. (1995). Collaborative Learning and the Internet. ICCAI´95.

Enyedy, N., Vahey, P. e Gifford, B.R. (1997). Active and Supportive Computer-Mediated Resources for Student-to-Student Conversations. Anais da Conferência Computer Support for Collaborative Learning'97. Toronto, Canadá.

Farnham, S., Chesley, H.R., McGhee, D.E., Kawal, R. (2000). Structured Online Interactions: Improving the Decision-Making of Small Discussion Groups. Anais da Conferência Computer-Supported Cooperative Work´00. Philadelphia, USA.

Ferraris, C., Martel, C. (2000) 'Regulation in Groupware: The Example of a Collaborative Drawing Tool for Young Children' IEEE Press: Anais do International Workshop on Groupware – CRIWG´00. Madeira, Portugal.

Gerber, L.D. e Becker, K., 2000, "Contributions of Pattern Languages to Framework-Based Development in Layered Architectures". In: *Proceedings of the XXVI Conferencia Latino Americana de Informática - CLEI2000*, Mexico.

González, O.M., Verdú, M.J., Dimitriadis, Y.A., Osuna, C.A., Iglesias, C.A., López, J. (1997) PENCACOLAS: Groupware for Learning. Anais do 3rd.International Workshop on Groupware, Espanha.

Gutwin, C., Stark, G., Greenberg, S. (1995). Support for Workspace Awareness in Educational Groupware. Anais da Conferência Computer Supported Collaborative Learning, USA.

Guzdial, M., Rick, J., Kerimbaev, B. (2000). Recognizing and Supporting Roles in CSCW. Anais da Conferência Computer-Supported Cooperative Work´00. Philadelphia, USA.

Habanero - http://www.ncsa.uiuc.edu/SDG/Software/Habanero/Tools.

Johnson, D.W., Johnson, R.T. & Holubec, E.J. (1990). "Circles of Learning". (3rd. edition). Edina, MN: Interaction Book Company.

Johnson-Lenz, P e Johnson-Lenz, T. (1991). Post-mechanistic Groupware Primitives: Rhythms, Boundaries and Containers. Intelligent Journal of Man-Machine Studies 34.

Kaye A. (1991). Learning Together Apart. In: Collaborative Learning through Computer Conferencing. (Ed.) A. Kaye, Berlin: Springer-Verlag.

Kynigos, C. (1999) Perspectives in Analysing Classroom Interaction Data on Collaborative Computer-Based Mathematical Projects. Anais da Conferência Computer Supported Collaborative Learning. Stanford, USA.

Leite, Aury de Sá e Omar, Nizam (1999). Representação de Conhecimento Pedagógico e Didático em Sistemas Educativos Inteligentes. Anais do X Simpósio Brasileiro de Informática e Educação, Curitiba.

Maher, M.L. (1999). Designing Virtual Campus as a Virtual World. Anais da Conferência Computer Supported Collaborative Learning. Stanford, USA.

Marshak, Ronni T. (1995). "Groupware: Technology and Applications", Capítulo 3: Workflow: Applying Automation to Group Processes. David Coleman & Raman Khanna (editors), Prentice Hall, NJ, USA.

Miao, Y., Haake, J.M., Steinmetz, R. (2000) ' A Rule-based Method to Shift between Learning Protocols' Anais do Educational Mutimedia, Hypermedia and Telecommunications Conference.

Mühlenbrock, M., Hoppe, U. (1999). Computer Supported Interaction Analysis of Group Problem Solving. Anais da Conferência Computer Supported Collaborative Learning. Stanford, USA.

O'Neill, K., Gomez, L.M. (1994) "The Collaboratory Notebook: a Networked Knowledge-Building Environment for Project Learning". Anais da Conferência Educational Mulimedia and Telecommunications-ED-Media'94.

Oshima, J. (1997) "Students´ Construction of Scientific Explanations in a Collaborative Hyper-Media Learning Environment". Anais da Conferência Computer Supported on Collaborative Learning. Toronto, Canadá.

Roussos, M., Johnson, A.E., Leigh, J., Barnes, C.R., Vasilakis, C.A., Moher, T.G. (1997) "The NICE Project: Narrative, Immersive, Constructionist/Collaborative Environments for Learning in Virtual Reality". Anais da Conferência Educational Mulimedia and Telecommunications- ED-Media.

Santoro, F.M., Borges, M.R.S., Santos, N., 2000b, "An Infrastructure to Support the Development of Collaborative Project-Based Learning Enviroments". In: *IEEE Press Proceedings of International Workshop on Groupware – CRIWG´00*, Madeira, Portugal, pp. 78-85.

Singley, M. K., Singh, M., Fairweather, P., Farrell, R., Swerling, S. (2000). Algebra Jam: Supporting Teamwork and Managing Roles in a Collaborative Learning Environment. Anais da Conferência Computer-Supported Cooperative Work´00. Philadelphia, USA.

Singley, M.K., Fairweather, P.G., Swerling, S. (1999). Team Tutoring Systems: Reifying Roles in Problem Solving. Anais da Conferência Computer Support for Collaborative Learning´99. Stanford, EUA.

Slavin, R.E. (1990). "Cooperative Learning: Theory, Research and Practice". Englewood Cliffs, NJ: Prentice-Hall.

Smith, R.B., Hixon, R., Horan, B. (1998). Supporting Flexible Roles in a Shared Space. Anais da Conferência Computer-Supported Cooperative Work´98. Seattle, USA.

Stahl, G. (1999). Reflections on WebGuide: Seven Issues for the Next Generation of Collaborative Knowledge-Building Environments. Anais da Conferência Computer Support for Collaborative Learning´99. Stanford, EUA.

Suthers, D., Weiner, A. (1995). Groupware for Developing Critical Discussion. Anais do Computer Supported for Collaborative Learning'95, EUA.

Tarouco, Liane Margarida R. e Hack, Luciano Emílio. A Avaliação na Educação à Distância: o Modelo de Kirkpatrick. Anais do X Simpósio Brasileiro de Informática e Educação, Curitiba.

Tedesco, P.A. and Self J.A, MArCO: using conflict mediation strategies to support group planning interactions, Technical Report 00/4, Computer Based Learning Unit, University of Leeds, 2000.

Tiessen, E.L., Ward, D.R. (1999). Developing a Technology of Use for Collaborative Project-Based Learning. Anais da Conferência Computer Support for Collaborative Learning´99. Stanford, EUA.

Tyler Ralph Winfred, 1902- Princípios Básicos de Currículo e Ensino; tradução de Leonel Vallandro, Porto Alegre, Globo, 1974.

Wan, D., Johnson, P.M. (1994). Computer Supported Collaborative Learning Using CLARE: the Approach and Experimental Findings. Anais da 1994 ACM Conference on Computer Supported Cooperative Work,Chapel Hill, North Carolina.

Van der Veen, J., Jones, V., Collis, B. (1998) "Workflow applied to Projects in Higher Education". COOPIS´98.

WEB HANDLERS

Gibeon Aquino*e Paulo Borba†

Centro de Estudos e Sistemas Avançados do Recife (CESAR)
Centro de Informática, Universidade Federal de Pernambuco

Abstract

With the widespread use of the Internet, more and more web-based information systems are being developed. Nowadays, many of these are developed in Java because of the several quality and productvity factors offered by the language and, mainly, for the amount of developed solutions using it. Java web technologies are very powerful, but are very new too. For this reason, there isn't a good number of design patterns or idioms for Java web-based systems [1]. In order to contribute to solve this problem, here we describe the Web Handlers pattern, which provides a way to structure and organize web-based systems to prevent replication of code.

Contexto

Em sistemas *Web* baseados no modelo de pedido-resposta é comum existirem diferentes requisições (pedidos) gerando dinamicamente a mesma página HTML como resposta. Esta situação pode ser percebida em um exemplo bem simplificado de sistema bancário que possui apenas as operações de crédito e débito em conta corrente, além de uma operação de *login* no sistema. O funcionamento deste é especificado através do mapa navegacional da Figura 1, que usa a notação de Diagrama de Estados de UML [8], onde as operações são desenhadas como eventos e as páginas HTML (dinâmicas ou estáticas) como estados.

De fato, analisando a figura percebe-se que as operações `Debito`, `Credito` e `Login`, apesar de serem operações distintas, geram como resposta de sua execução a mesma página (`Menu de Movimentações`). Este fato, apesar de exemplificado com um menu principal, ocorre em diversas outras situações.

Outra situação comum em sistemas desta natureza é termos uma mesma requisição gerando diferentes respostas, dependendo da origem da requisição ou do resultado do de seu processamento. Para exemplificar esta situação, considere que o cliente, além de realizar movimentações em conta (débito e crédito), pode fazer atualizações em seu cadastro. A Figura 2 exemplifica bem a situação em que a mesma operação, `Login`, é executada a partir de contextos diferentes (`Página de Login 1` e `Página de Login 2`) e deve gerar uma saída específica (`Menu de Atualização` e `Menu de Movimentações`) dependendo da origem da requisição. Estas duas páginas de *login* possuem o conteúdo bem diferente pois elas estão em contextos distintos e não possuem relação direta.

*Email: gibeon@cesar.org.br
†Parcialmente financiado pelo CNPq, processo 521994/96–9. Email: phmb@cin.ufpe.br

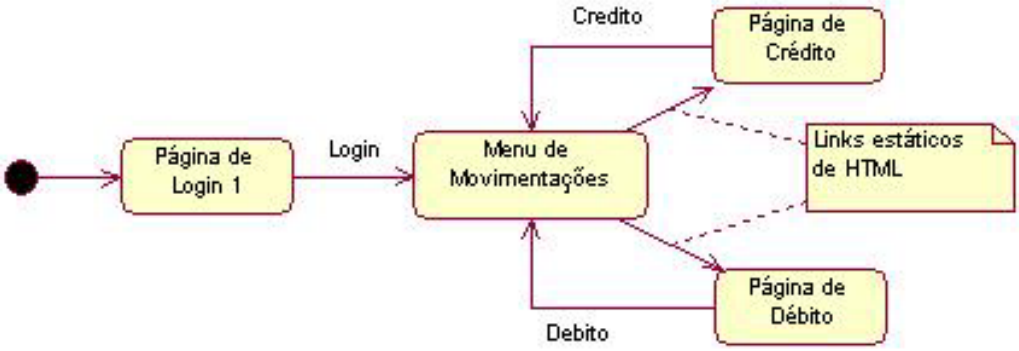


Figura 1: Mapa navegacional simples do sistema bancário

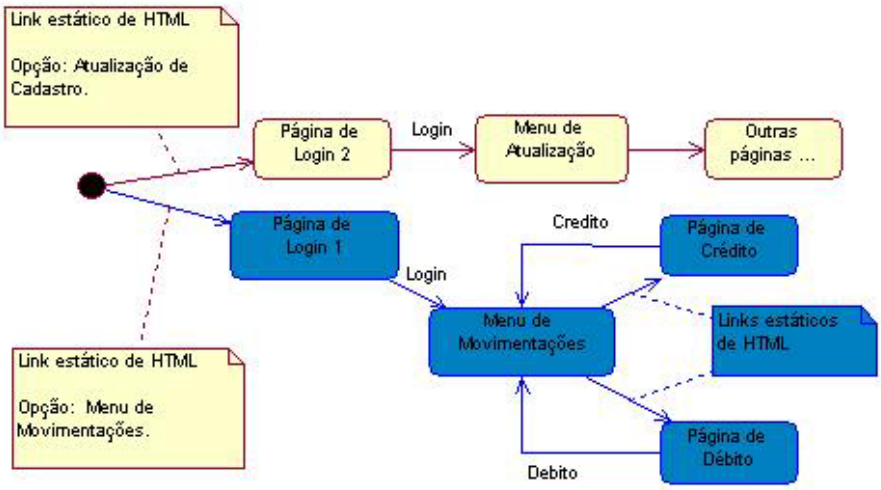


Figura 2: Mapa navegacional completo do sistema bancário

As situações ilustradas pelas Figuras 1 e 2 são bastante comuns em sistemas *Web* não triviais. Estas geralmente trazem problemas de implementação, como duplicação e complexidade do código, quando não é aplicada uma estruturação de componentes *Web* adequada ao problema. Em situações onde a mesma página pode ser gerada como resultado da execução de diferentes requisições (Ilustrado pela Figura 1) é comum ocorrer duplicação de códido relativo à montagem desta página em cada um dos componentes que tratam as requisições. Já em situações onde uma mesma operação pode gerar diferentes páginas como resultado de seu processamento (Ilustrado pela Figura 2) é comum haver um aumento na complexidade de implementação do componente que executa a requisição, já que este precisa decidir que resposta apresentar, além de possuir o código relativo a montagem de cada uma das páginas de resposta gerada por ele.

Problema

Evitar a duplicação de código e complexidade na estruturação de sistemas *Web* com relacionamento M:N entre a apresentação e o processamento.

Forças

Positivas

- A estruturação orientada a página evita a duplicação do código referente à montagem da apresentação.
- A estruturação orientada a operação evita a duplicação do código de processamento da requisição.

Negativas

- A estruturação orientada a página não pode ser aplicada para impedir a repetição do código de processamento da requisição. Além do mais, esta alternativa pode fazer com que o seu componente *Web* fique mais complexo quando a quantidade de operações que geram a mesma página de saída aumenta.
- A estruturação orientada a operação não é capaz de impedir que a lógica de montagem das páginas se repita. Esta também apresenta o problema de poder ficar complexo quando aumenta o número de páginas de saída possíveis para a mesma operação.

Solução

A solução é baseada na construção de entidades denominadas *handlers*. Existem dois tipos destes: *Handlers* de Apresentação e *Handlers* de Processamento. Os primeiros contêm apenas código relativo à montagem das páginas dinâmicas e os outros possuem código (ou chamadas) relativo à execução da lógica de negócio. Com esta estruturação é necessário criar um *handler* de processamento para cada operação do sistema e um de apresentação para cada página dinâmica.

Cada requisição *Web* dispara uma execução do lado do servidor que dinamicamente associa um par de *handlers* (um de apresentação e um de processamento) para responder ao cliente. Esta composição dinâmica é determinada por um parâmetro da requisição do cliente. A entidade responsável pela montagem deste par e delegação da requisição primeiro para o *handler* de processamento, depois para o de apresentação é o **Controlador de Handlers**, que é especificado com mais detalhes nas Seções Estrutura e Implementação.

Os *handlers* de processamento, além de conterem chamadas às operações do sistema, possuem código responsável pela validação dos dados vindos do cliente *Web*, e código responsável pela preparação dos dados para seu par de apresentação.

Os *handlers* de apresentação também possuem validação de dados, além de código de montagem das páginas dinâmicas. Esta validação é necessária porque estes precisam

validar os dados gerados pelo seu pares, já que eles são entidades independentes e podem ser compostos de diferentes formas.

Exemplo da solução

Para o sistema bancário, as seguintes entidades deveriam ser criadas com a utilização do padrão:

- *Handlers* de Apresentação:
  - `HA_MenuAtualizacao`, responsável por montar dinamicamente o `Menu de Atualização`;
  - `HA_MenuMovimentacoes`, responsável por montar dinamicamente o `Menu de Movimentações`.
- *Handlers* de Processamento:
  - `HP_Login`, responsável por executar a operação `Login`;
  - `HP_Credito`, responsável por executar a operação `Crédito`;
  - `HP_Debito`, responsável por executar a operação `Débito`.

Estes *handlers* podem ser compostos de diferentes formas para responder aos diferentes tipos de requisições. A Figura 3 exemplifica algumas composições possíveis para o sistema bancário com a utilização desse tipo de estruturação. De fato, o uso de *handlers* é capaz de resolver os problemas de duplicação e complexidade de código em casos de relacionamentos M:N entre a apresentação e o processamento, podendo assim ser aplicado na maioria das situações comuns a sistemas *Web*. Na Figura 3 é possível ver como os *handlers* podem ser reusados em diferentes requisições, evitando assim a repetição de código.

Aplicabilidade

Use o padrão principalmente em sistemas onde aparecem relacionamentos M:N entre as partes de processamento e apresentação.

Desenvolver o sistema já usando do padrão mesmo quando não ocorrem estas situações é uma boa prática, pois desta forma o desenvolvedor já torna o sistema imune à problemas de repetição de código antes mesmo de identificá-los.

Uma forma simples de identificar se o seu sistema necessita do uso deste padrão é desenhando o mapa navegacional do sistema e verificando a existência de ciclos. O aparecimento de ciclos indica que o seu sistema pode apresentar relacionamento 1:N da apresentação para o processamento (como na Figura 1). Para identificar a ocorrência de relacionamentos 1:N do processamento para a apresentação é só verificar, no mapa navegacional, se a mesma operação aparece em duas transições diferentes (como na Figura 2).

Estrutura

A estrutura do padrão Web Handlers é especificada através do diagrama de classes de UML da Figura 4.

Figura 3: Composições possíveis de *handlers* de apresentação e processamento.

Participantes

- `ControladorHandlers` - Responsável pelo recebimento das requisições e controle da execução dos *handlers* responsáveis pela mesma;
- `HandlerApresentacao` - Responsável pela montagem de uma ou mais páginas semelhantes;
- `HandlerProcessamento` - Responsável pela invocação dos serviços, e também pela geração de dados para os *handlers* de apresentação a serem executados em conjunto com ele;

Os *handlers* possuem algumas operações padrões que devem ser definidas pelo programador para que eles se comportem da maneira esperada, estas são:

- `iniciar` - Inicia o *handler* e é invocado pelo ambiente;
- `finalizar` - Finaliza o *handler* e é invocado pelo ambiente;
- `apresentar` - Contém a lógica referente à montagem das páginas dinâmicas;

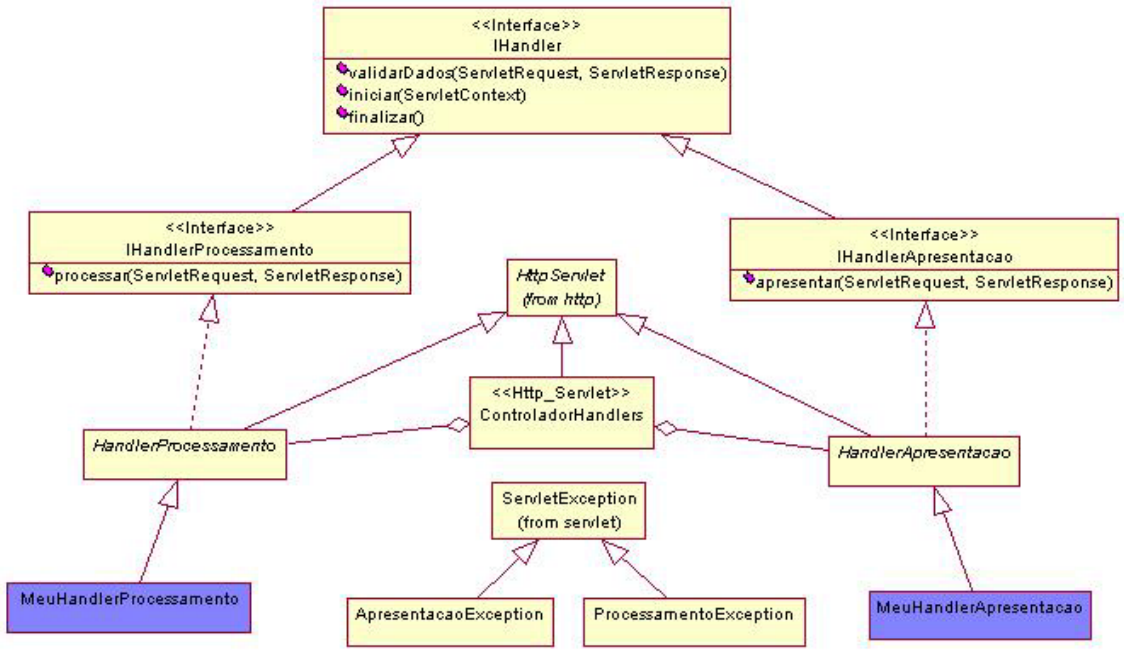


Figura 4: Diagrama de classes do padrão Web Handlers

- `processar` - Contém a lógica referente à chamada dos serviços;
- `validarDados` - Contém regras de validação dos dados de entrada.

Os *handlers* possuem ciclo de vida semelhante aos dos *servlets* [14]. Assim são oferecidos os métodos `iniciar` e `finalizar`, para que o programador possa definir operações que são executadas na sua inicialização e finalização, respectivamente. Todo *handler* pode implementar o método `validarDados`, que é executado antes do `processar` ou do `apresentar`. Para os *handlers* de processamento ele pode ser usado para implementar regras de validação dos dados da requisição *Web*, enquanto que nos de apresentação ele contém a validação dos dados gerados pelo processamento.

Dinâmica

- Toda requisição *Web* é recebida pelo `controlador`, que interpreta seus parâmetros, recupera o `processamento` e `apresentacao` necessários para executar a requisição e dinamicamente faz a composição dos dois, delegando a requisição para o par.
- O `processamento` é o primeiro a ser executado através da chamada ao seu método `processar`. Nele vão estar a chamadas aos serviços do sistemas, implementados por objetos que encapsulam toda a regra de negócio da aplicação (EJBs [13], *Facade* [5] etc). Ele também é responsável por produzir os dados de entrada do *handler* de apresentação que será associado a ele.
- O *handler* `apresentacao` é executado após o término com sucesso do `processamento` através de uma chamada a seu método `apresentar`. Nele não deve haver lógica de

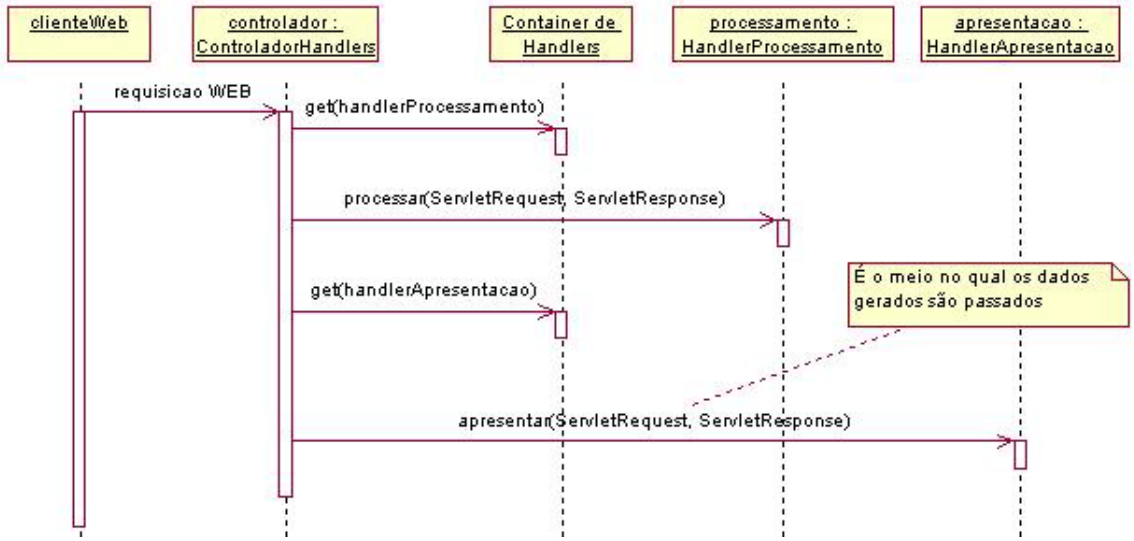


Figura 5: Diagrama de seqüência dos componentes do padrão *Web Handlers*

processamento da requisição, apenas código referente à montagem da página. Este *handler* consulta os dados gerados pelo seu par processamento e constrói a página de resposta baseada nestes.

- O `Container de Handlers` é uma entidade que está neste contexto apenas para que se tenha um entendimento melhor do processo de acesso e recuperação dos *handlers*. Ele nada mais é do que o ambiente onde os *handlers* são executados, em aplicações *Web*, por exemplo, um servidor *Web* ou mais especificamente um *Container Web*.

Conseqüências

- Grande flexibilidade na composição das partes de apresentação e processamento – Os *handlers* de apresentação e processamento são independentes um dos outros e podem ser integrados de forma diferente para responder a diferentes tipos de requisições, desde que eles sejam compatíveis em relação aos dados produzidos e consumidos.
- Maior reuso de código - O padrão evita a repetição desnecessária de código, pois permite o compartilhamento de entidades para responder a diferentes requisições.
- Mudanças nos mecanismos de montagem da apresentação (JSP [7], FreeMarker [17], WebMacro [11] ou Velocity [15]) não causam efeito algum nas entidades de processamento.
- Facilita a implementação de sistemas que requerem diferentes formatos de saída (XML [3], HTML, WML [4], XHTML [4], etc.) para a mesma operação – Com o uso do **Web Handlers** o desenvolvedor pode criar *handlers* de processamento que serão compostos com diferentes *handlers* de apresentação, onde estes últimos possuem implementações para cada formato de saída possível.
- Facilita a implementação de componentes (*handlers* de apresentação e processamento) que podem ser reusados em outros sistemas – Com o uso do padrão é mais

fácil manter uma biblioteca de *handlers* onde o desenvolvedor pode consultar serviços já implementados e compô-los a fim de obter a implementação de um serviço desejado.

- O padrão ajuda a evitar repetição de código, no entanto ele possui a desvantagem de aumentar o número de classes necessárias para implementar um sistema. Por isso é preciso ter um certo cuidado em não tornar o sistema modular demais, pois quanto mais modular ele for, maior será o número de classes que precisam ser criadas;
- A divisão da responsabilidade pelo tratamento da requisição acrescenta uma complexidade à implementação dos componentes, pois é necessário a passagem de parâmetros do *handler* de processamento para o de apresentação a cada requisição, já que eles são entidades distintas.

Implementação

Todos os *handlers* possuem um comportamento bem similar e interfaces bem definidas, por isso é interessante que existam classes e interfaces que dêem apoio ao funcionamento do padrão, provendo comportamento genérico e especificando a interface das operações dos *handlers*. Uma estruturação interessante deste padrão pode ser vista na Figura 6.

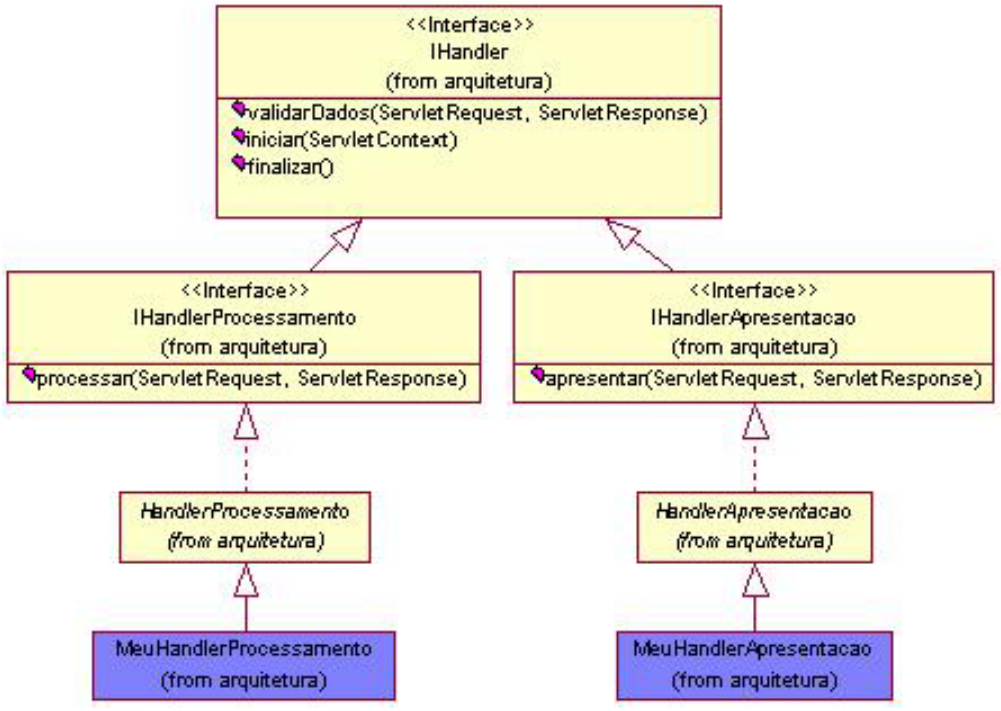


Figura 6: Diagrama de classes refinado dos componentes do Web Handlers

- `IHandler` – Interface genérica que especifica as funcionalidades que todo *handler* deve oferecer.

- `IHandlerProcessamento` – Interface que define todas as operações que devem existir em um *handler* de processamento.
- `IHandlerApresentacao` – Interface que define todas as operações que devem ser oferecidas por um *handler* de apresentação.
- `HandlerProcessamento` – Classe abstrata que implementa os métodos definidos na interface `IHandlerProcessamento`. Ela provê uma implementação padrão para os *handlers* de processamento. Toda entidade de processamento deve estender esta classe para possuir o comportamento de um *handler* de processamento.
- `HandlerApresentacao` – Classe abstrata que implementa as funcionalidades definidas na interface `IHandlerApresentacao`. Toda entidade de apresentação deve estender esta classe para possuir o comportamento de um *handler* de apresentação.

Os tipos `ServletRequest`, `ServletResponse` e `ServletContext` são classes padrões da API de *servlets* [14]. A primeira é o meio pelo qual os parâmetros da requisição são passados. O segundo é usado para enviar resposta para o cliente *Web*. O último é uma referência para o *Servlet Container*, através do qual pode-se recuperar parâmetros de configuração do ambiente, comunicar-se com outras entidades que estejam executando no mesmo contexto, etc. Os componentes aqui apresentados são específicos para uma implementação baseada em *servlets*, no entanto o padrão é genérico o bastante para que possa ser implementado em outras tecnologias.

Para tornar o ambiente de *handlers* ainda mais poderoso é interessante que existam algumas funcionalidades providas pelo contexto onde eles estão sendo executados:

- **Instanciação automática dos *handlers*** - O ambiente é responsável por localizar e carregar os *handlers* necessários para a execução da requisição;
- **Reload automático de *handlers*** - As modificações feitas em *handlers* já carregados serão enxergadas pelo contexto automaticamente;
- **Ciclo de vida bem definido e gerenciado pelo ambiente** - Os *handlers*, assim como os *servlets*, possuem um ciclo de vida bem definido e que é controlado pelo ambiente no qual eles estão executando.

Todas estas características precisam ser implementadas no ambiente dos *handlers*, mas como o ambiente de *servlets* (*Servlet Container*) já provê todos estes serviços, é uma boa idéia usá-los para os *handlers*. Uma forma de usar estes recursos de forma efetiva é fazendo com que os *handlers* sejam executados dentro do *Servlet Container*. Para isso eles precisam ter a interface de um *servlet* e se comportar como tal. Uma pequena modificação na estrutura apresentada anteriormente pode ser feita de forma a atender estes requisitos e preservar a semântica dos *handlers* apresentada até o momento. A Figura 7 dá uma idéia da alteração necessária no modelo.

As implementações genérica `HandlerApresentacao` e `HandlerProcessamento` continuam implementando as mesmas interfaces, mas agora estendem a classe `HttpServlet` (pertencente a API de *servlets*) para herdarem o comportamento de *servlet* e poderem ser gerenciados pelo *Web Container*. Outro detalhe é que estas implementações genéricas declaram todos os métodos herdados da classe `HttpServlet` como "final", desta forma

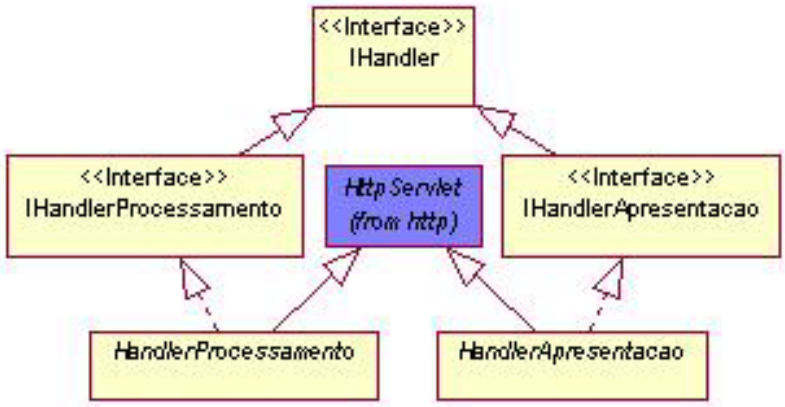


Figura 7: Estrutura do padrão no ambiente de servlets

evitam que suas subclasses redefinam estes métodos. Enfim, para as classes que o programador precisa implementar, esta mudança na herança de `HttpServlet` não causa nenhum efeito direto.

Na Figura 8 pode-se ver o diagrama de classes do padrão completo. Foram acrescentadas as classes de exceção `ApresentacaoException` e `ProcessamentoException`, ambas herdando de `ServletException`, e cada uma destas podem ser lançadas pelos métodos dos *handlers* de Apresentação e Processamento, respectivamente.

A classe `ControladorHandlers` é um *servlet* que faz o papel do Controlador de Handlers mostrado na Seção Estrutura. Este é implementado como um *servlet* porque precisa receber todas as requisições *Web*, e só depois de interpretá-las, repassá-las para os *handlers* responsáveis pelo sua execução.

Uma variação da implementação para resolver o problema do aumento do número de classes do sistema é permitir que o desenvolvedor possa agrupar operações relacionadas ou semelhantes em um único *handler*. Com isso seria necessário implementar um mecanismo de execução de *handlers* mais elaborado, onde além de informações a respeito dos *handlers* a serem executados numa determinada requisição, deveria haver parâmetros indicando que operação executar em cada um deles. Este mecanismo pode ser implementado com a utilização da API *Reflection* [12] de Java.

Código de exemplo

Para exemplificar o uso do padrão são mostradas as implementações de duas classes do sistema bancário, um *handler* de apresentação e outro de processamento. A Figura 8, mostrada anteriormente, dá uma idéia geral da estrutura destas classes e como elas são compostas para atender as requisições do sistema. No trecho abaixo é mostrada a implementação concreta destas classes em Java.

A classe `HP_Login` é um *handler* de processamento que executa a operação de *login* no sistema. Como pode ser visto na linha 1, este estende a classe `HandlerProcessamento`, herdando assim o comportamento de *handler* de processamento. Na linha 3 é declarada uma variável do tipo `Sistema`, que agrupa todos os serviços do sistema e é uma implementação dos padrões *Facade* e *Singleton*. Na inicialização deste *handler* (chamada ao

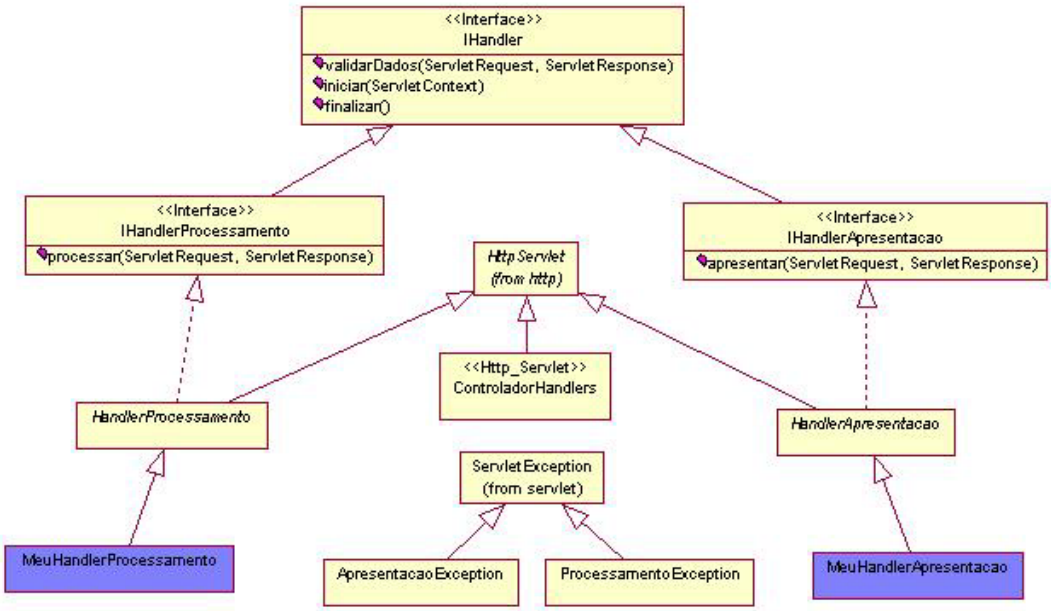


Figura 8: Estrutura do padrão refinada e no ambiente de servlets.

método `iniciar`) é recuperada uma instância do sistema para que seja usado durante o processamento (linha 7).

```
1:   public class HP_Login extends HandlerProcessamento {
2:
3:      private Sistema sistema;
4:
5:      public void iniciar(ServletConfig config)
6:         throws ProcessamentoException{
7:         sistema = Sistema.getInstancia();
8:      }
```

O método `validarDados` verifica se os parâmetros da requisição necessários para execução estão presentes (linhas 13 e 14), caso não estejam é lançada uma exceção indicando que o processamento deve ser interrompido.

```
9:      public void validarDados(HttpServletRequest request,
10:                              HttpServletResponse response)
11:        throws ProcessamentoException{
12:
13:        if(request.getParameter("login") == null ||
14:           request.getParameter("senha") == null){
15:           throw new ProcessamentoException(
                   "Parâmetros imcompatíveis",null);
16:        }
17:     }
```

As linhas 24 e 25 são usadas para recuperar o valor do login e senha, passados como parâmetro da requisição. Estas são usadas para fazer a validação do usuário no sistema através de uma chamada ao método `validarUsuario` do objeto Facade (linha 27). Se o login ocorrer com sucesso, as informações do usuário são recuperadas e armazenadas em um objeto do tipo `Usuario` através de uma chamada ao método `recuperarUsuario` da Facade (linha 28). Após este processo, o objeto é armazenado no `request` para que possa ser recuperado pelo *handler* de apresentação associado a esta requisição.

```
18: public void processar(HttpServletRequest request,
19:                       HttpServletResponse response)
20:    throws ProcessamentoException{
21:    String login,senha;
22:    Usuario usuario;
23:
24:    login = request.getParameter("login");
25:    senha = request.getParameter("senha");
26:
27:    if(sistema.validarUsuario(login,senha)){
28:       usuario = sistema.recuperarUsuario(login);
29:       request.setAttribute("usuario",usuario);
30:    }
31:    else{
32:       throw new ProcessamentoException("Login Inválido",null);
33:    }
34: }
35:}
```

O `HA`$_{\text{M}}$`enuMovimentacao` é um *handler* de apresentação, por isso ele estende a classe `HandlerApresentacao`. Seu método `validarDados` verifica a existência do parâmetro `usuario` (linha 7).

```
1: public class HA_MenuMovimentacao extends HandlerApresentacao {
2:
3:  public void validarDados(HttpServletRequest request,
4:                           HttpServletResponse response)
5:    throws ProcessamentoException{
6:
7:    if(request.getAttribute("usuario") == null){
8:      throw new
9:        ApresentacaoException("Parâmetros imcompatíveis",null);
10:   }
11: }
```

O método apresentar recupera o objeto do tipo `Usuario` através do `request` (linha 23). A lógica principal de montagem da página está na linha 26, onde a página é montada através de uma chamada ao método `Skin.processaPagina()`. Este método recebe com entrada um *array* de chaves, outro de valores e um nome de arquivo. Este arquivo é um *template* de uma página HTML contendo alguns identificadores especiais nos locais

onde serão inseridas informações dinâmicas. A tarefa deste método é varrer o arquivo procurando ocorrências de algumas das palavras do *array* de chaves e substituí-las por palavras do *array* de valores. Na linha 27 a página de resposta é enviada para o cliente *Web*.

```
13: public void apresentar(HttpServletRequest request,
14:                        HttpServletResponse response)
15:    throws ApresentacaoException{
16:
17:    PrintWriter out;
18:    String pagina;
19:    Usuario u;
20:
21:    out = response.getWriter();
22:    try{
23:      u = (Usuario) request.getAttribute("usuario");
24:      String[] chaves = {"$USUARIO"};
25:      String[] valores = {u.getName()};
26:      pagina = Skin.processaPagina(chaves,
                                      valores,
                                      "Menu_Movimentacao.html");
27:      out.println(pagina);
28:    }
29:    catch(Exception e){
30:       throw new ApresentacaoException("Erro de Apresentação",e);
31:    }
32: }
33:}
```

Uso conhecido

- **Portal Encontre & Compre** [10] - Sistema de consultas dos anunciantes Listel, que também permite que o visitante faça transações de negócios on-line com os anunciantes;
- **O Sistema de Fomento Lattes** [9] - Sistema de informação para gestão de programas de fomento ao desenvolvimento científico e tecnológico. O módulo responsável pela emissão de parecer de consultor *Ad hoc* usa o padrão aqui descrito em sua implementação;
- **Prospectar** [16] - Sistema de prospecção tecnológica do Governo Federal;
- **Web2Billing** [18] – É uma solução completa de EBPP (*Eletronic Bill Presentation and Payment*), desenvolvida pela Wiser Tecnologies, e que permite a criação, geração, gerenciamento, apresentação, consulta e pagamento de faturas *online*;
- **FiS (Financial Services)** – O projeto contempla a migração dos Módulos de Contabilidade, Crédito, Lojistas e Serviços da HiperCard, para um novo ambiente tecnológico (J2EE). Estes módulos são integrados ao Sistema de Integrado de Crédito

da HiperCard (SIC), ao R3/SAP e a outros sistemas legados. Também faz parte do projeto o desenvolvimento de um módulo de Controle de Acesso único e centralizado que poderá ser utilizado por qualquer aplicação da HiperCard disponível neste novo ambiente;

- **Fep (Call Center no FEP)** – Desenvolvimento de uma aplicação para a Central de Atendimento HiperCard que autorizará compras no autorizador FEP (*Front End Processor*) e no sistema legado SIC (Sistema de Integrado de Crédito da HiperCard) via browser;

- **Gin (Sistema de Gestão Interna)** [6] – Sistema de apoio a gestão interna do CESAR com cadastros e relatórios gerais, além de englobar os sistemas financeiro e avaliação de colaboradores.

Todos os sistemas descritos anteriormente usam o padrão de Web Handlers na construção de seus serviços *Web*. A implementação do padrão proposta neste documento foi fruto de um trabalho de correção dos problemas identificados nas implementações anteriores, mas a estrutura do padrão continua a mesma.

Padrões relacionados

- Na construção dos *handlers* de apresentação pode ser usado o padrão Web Compiler [2] para implementar a lógica de montagem das páginas dinâmicas. O uso do padrão neste contexto traz uma série de benefícios ao desenvolvimento, pois permite a separação entre o código HTML e Java, facilitando o desenvolvimento e a manutenção do sistema.

- É interessante usar o padrão de projeto Facade [5] para agrupar a lógica de negócio do sistema em um único ponto e fazer com que os *handlers* de processamento chamem estas funcionalidades, ao invés de implementarem-na diretamente em seus corpos.

- O Controlador de Handlers (componente da estrutura do Web Handlers) deve implementar o padrão Web Interceptor [2] já que o Controlador deve ser o único ponto de acesso aos *handlers* do sistema.

- O padrão Super Component [2] pode ser usado na implementação dos *handlers* de apresentação e processamento a fim de evitar a duplicação de código nos métodos `iniciar` e `finalizar`.

Referências

[1] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns – Best Practices and Design Strategies*. Prentice Hall, March 2001.

[2] Gibeon Soares de Aquino Júnior. Desenvolvimento de Sistemas Web em Java, 2002.

[3] W3C Architecture Domain. Extensible Markup Language (XML). Disponível em http://www.w3.org/XML/.

[4] Organization for the Advancement of Structured Informations Standard (OASIS). WAP Wireless Markup Language (WML). Disponível em http://www.oasis-open.org/cover/wap-wml.html.

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

[6] Gin. Sistema de Gestão Interna. http://www.cesar.org.br.

[7] Marty Hall. *Core Servlets and JavaServer Pages.* Prentice Hall, 2000.

[8] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process.* Addison–Wesley, 1999.

[9] Lattes. Sistema de Fomento Lattes. Disponível em http://www.cnpq.br/servicosrestritos/.

[10] Listel. Portal Encontre e Compre. Disponível em http://www.listel.com.br.

[11] Web Macro. Web Macro home page. Disponível em http://www.webmacro.org.

[12] Sun Microsystems. Java Core Reflection Api Documentation. Disponível em http://java.sun.com/j2se/1.3/docs/guide/reflection/spec/java-reflectionTOC.doc.html, 1998.

[13] Sun Microsystems. Enterprise java beans(tm) specification. Version 2.0, Final Release, 22th August 2001.

[14] Sun Microsystems. Java(tm) Servlet Specification version 2.3. Disponível em http://java.sun.com/products/servlets, 17th Setember 2001.

[15] Jakarta Project. Velocity template engine. Disponível em http://jakarta.apache.org/velocity/.

[16] Prospectar. Sistema de Prospecção Tecnológica. Disponível em http://prospectar.cesar.org.br/admin.

[17] SourceForge.net. Freemarker 1.7 - an open-source html template engine for java servlets. Disponível em http://freemarker.sourceforge.net/.

[18] Web-2-Billing. Web 2 Billing Pagamentos online. Disponível em http://www.web2billing.com.br.

A Collection of Patterns for Use Case Maps

Gunter Mussbacher and Daniel Amyot[1]
School of Information Technology and Engineering (SITE), University of Ottawa
161 Louis-Pasteur, PO Box 450, Stn. A, Ottawa (ON), Canada, K1N 6N5
damyot@site.uottawa.ca

Our gratitude extends to Jim Coplien for his efforts as our shepherd.

Abstract: Use Case Maps (UCMs) are a technique used to capture functional requirements and high level designs of complex systems composed of many features. Once you have chosen UCMs as part of your software development process, the question arises as how to most effectively use UCMs. This paper introduces patterns which provide guidance in selecting one of three major UCM styles depending on your software development context. The "Individual Maps" UCM style is most useful for rapidly and independently capturing a few key features of your system, features being optional or incremental units of functionality. The "Standard Root Map" UCM style is most appropriate if a small, evolving system consisting of interacting features needs to be documented. The "Isolation and Integration" UCM style is best applied to large, evolving systems with many interacting features.

1 Introduction

This document consists of two major parts. Section 2 "Background Information" gives a basic overview of a) the application domain of the patterns, i.e. capturing functional requirements and high level designs for complex systems composed of multiple features with a technique called Use Case Maps (UCMs) and b) the framework used to reason about forces, i.e. the NFR (Non-Functional Requirements) Framework. Readers familiar with Use Case Maps may skip the respective sub-section and proceed directly to section 2.2. Readers familiar with the NFR framework, however, are encouraged to read through section 2.2 since slightly different notational elements are used than suggested by the NFR Framework.

Sections 3 to 9 contain the actual collection of patterns organized in pattern form. Section 3 gives an overview of the pattern collection whereas sections 4 to 9 describe the patterns. These patterns provide guidance in selecting one of three UCM styles for various contexts.

2 Background Information

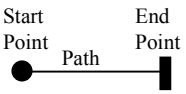
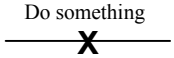
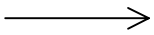
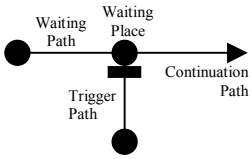
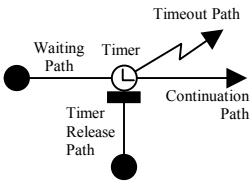
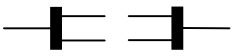
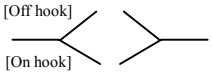
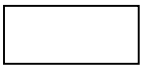
This section introduces the reader to the basic concepts and the notation of Use Case Maps – the application domain of the patterns presented in sections 3 to 9 – as well as the NFR framework used to reason about forces. The material in section 2 covers only those aspects of Use Case Maps and the NFR Framework required for the patterns. For further information on UCMs and the NFR framework see Section 11 "References" at the end of the document.

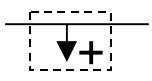
---

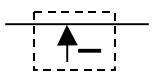
[1] The majority of this work was conducted while the authors were at Mitel Networks, 350 Legget Dr., Kanata (ON), Canada, K2K 2W7.

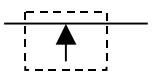
2.1 Use Case Maps Concepts and Notation

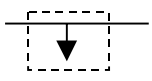
Use Case Maps (UCMs) [8, 9] are a scenario-based software engineering technique most useful at the early stages of software development. UCMs visually represent the causal relationships of responsibilities of one or more use cases combined with structures in one single view. The relationships are said to be causal because they involve concurrency and partial orderings of responsibilities, because they link causes to effects, and because they abstract from component interactions expressed by message exchanges. UCMs show related use cases in a map-like diagram. The map shows the progression of scenarios along use cases.

UCM Notation	Notation Explanation
	**Basic path.** The basic path is the most basic, complete unit. The path represents scenario flow. Paths connect start points, responsibilities, and end points. A path may have any shape as long as it is continuous (can cross itself). The start points represent preconditions or triggering causes. The end points represent post-conditions or resulting effects.
	**Responsibility point.** Represents generic processing (actions, tasks, or functions to be performed). Responsibilities may be bound to a component.
	**Direction** (optional). In general, the positioning of the start and end points of a path indicate direction. In certain cases, it is useful to show the direction on a complicated map.
	**Waiting place.** Represents a waiting place along a path. Propagation along the path stops at the waiting place until the trigger arrives. Waiting places can be triggered by a trigger path as shown or by the environment.
	**Timer.** A special waiting place that expresses the idea that there is a time limit on waiting. When propagation along the waiting path reaches the timer, the timer is set. Propagation along the continuation path continues if the timer release arrives. Propagation along the timeout path continues if the timeout occurs.
	**AND Fork** and **AND Join.** For concurrent paths (two or more).
	**OR Fork** and **OR Join.** An OR fork indicates that the path proceeds in only one out of two or more directions. Labels may identify alternative paths or guarding conditions. An OR Join indicates a common causal segment of two or more paths.
	**Static stub.** Contains only one plug-in (sub UCM), hence enabling hierarchical decomposition of complex maps.
	**Dynamic stub.** May contain several plug-ins, whose selection can be determined at run-time according to a *selection policy* (often described with pre-conditions).
	**Generic component.** Represents an architectural entity.
	**Slot.** Placeholder for dynamic components as operational units. **Dynamic responsibilities** can move dynamic components from a path into a slot or out of a slot onto a path.







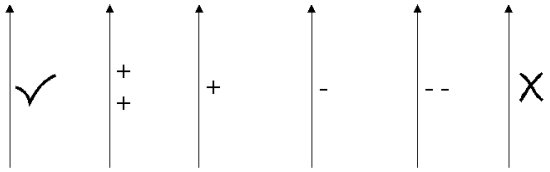


UCMs have a history of applications to the description of reactive systems of different natures (e.g. [2, 3, 4, 10]), to the avoidance and detection of undesirable interactions between scenarios or services (e.g. [4, 10, 18]), to early performance analysis (e.g. [20]), and the generation of more detailed behavioral models (e.g. [17, 19]). The UCM notation is also being considered by ITU-T for describing functional requirements as part of the upcoming User Requirements Notation standard [11]. The table above covers only the notational aspects of UCMs as required for sections 3 to 9. For further information on UCM concepts and the notation, the reader is referred to the virtual library of the *UCM User Group* [21].

The *UCM Navigator* tool [16, 17] supports a superset of the notational elements listed above as well as navigation through UCMs. The tool was used to create the UCMs in this document.

2.2 NFR Framework for Reasoning about Forces

The technique used to illustrate the forces of the context and the impact of solutions on these forces is based on the *NFR framework* [12, 15] and on work on the combination of the NFR framework and patterns [22]. The *OME* tool [14], developed at the Knowledge Management Lab at the University of Toronto, allows the creation of force graphs. The following table covers only the notational aspects of the framework as instantiated for the pattern domain.

<table>
<tr><td>Force</td><td> 
</td><td>Solution</td><td> 
</td></tr>
<tr><td>Contribution Links</td><td colspan="3"> 

Contribution links connect solutions or forces (source) and forces (target). The links indicate the impact a source has on a target.
✓ The source impacts the target so that the target is sufficiently balanced.
++ The source balances the target but not sufficiently.
+ The source balances the target but less than ++.
– The source unbalances the target but less than – –.
– – The source unbalances the target but not completely.
✗ The source impacts the target so that the target is completely unbalanced.
(Note that this notation uses different labels than those in the NFR framework.)</td></tr>
<tr><td>Force/Solution Labels</td><td colspan="3">A label positioned next to a force indicates how balanced the force is.
A label positioned next to a solution indicates how much of the solution has been achieved.
✓ Balanced/Fully achieved
+ Weakly balanced/achieved
0 Less than weakly balanced/achieved but more than weakly unbalanced/not achieved
– Weakly unbalanced/not achieved
✗ Unbalanced/Not achieved
(Note that this notation uses different labels than those in the NFR framework.)</td></tr>
</table>

3 A Collection of Patterns for Use Case Maps

This section gives a general introduction to the collection of patterns including general discussions on the overall context and problem for the pattern collection and the forces relevant to the pattern collection.

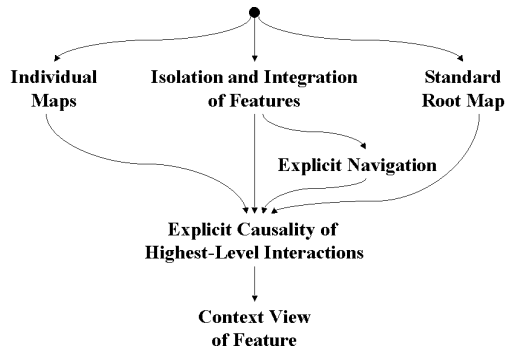


Figure 1: A Collection of Patterns for UCMs

3.1 Introduction

The collection of patterns consists of six patterns as depicted in Figure 1. The confidence in these patterns is very high because the patterns have been observed for numerous features in a large and complex call control software system for a telephone switch. Different development teams have developed these features over years. This paper fully describes the three top-level patterns and gives patlets for the remaining three patterns. Four additional patterns have been identified which extend this collection. These patterns, however, will not be included in this document in consideration of review time and the length of the document.

3.1.1 Overall Context

The overall context of the pattern collection at the root of Figure 1 is defined as follows. You are capturing functional requirements and high-level designs of your system. A scenario-driven software development approach has been chosen for your system. Furthermore, it has been decided that Use Case Maps (UCMs) will be used to capture functional requirements and high level designs. The UCM Navigator provides tool support. Thus, any solution has to consider the constraints of the tool.

UCMs give software engineers a high degree of freedom in how to describe systems. Currently, three UCM styles are known. Literature often suggests the “Individual Maps” [3, 5] approach (see 4) and the “Standard Root Map” [6, 13, 17, 18] approach (see 5). This paper introduces the “Isolation and Integration” approach (see 6). It, however, is not clear whether these approaches are applicable to all sub-contexts of the overall context and, if not, to which sub-contexts they are applicable.

3.1.2 General Problem

Find the best UCM style to be used in the given context.

The problem addressed by the discussed patterns does not relate directly to the use of UCMs for the description of behavioral patterns or design patterns. It relates to how general scenario descriptions (use cases, features) can be best organized in various contexts with the help of the UCM notation. The problem is also not related to whether UCMs are an appropriate technique for capturing requirements and high level designs. These issues have been addressed elsewhere [1, 7, 8, 9].

3.1.3 General Discussion on Forces

Figure 2 shows the hierarchy of forces that have to be considered for the overall context.

a) Evolveability:

An important high-level concern is the evolveability of the system. Several sub-forces contribute to the evolveability of a large system:

a.1) Understandability of a single feature:

A clear understanding of new and existing features is the basis for evolveability. The understandability of a feature depends on:

a.1.1) the simplicity of each UCM required for the feature (the simpler, the better),

a.1.2) the number of hierarchical levels of UCMs per feature (the lower, the better),

a.1.3) the number of distributed UCMs, i.e. the UCMs that belong to the same feature but are disjoint (the lower, the better),

a.1.4) the pollution of the feature description, i.e. whether UCM paths and path elements from different features can be clearly identified (the less polluted, the better), and

a.1.5) the ability to detect undesirable feature interactions [23] and specify desirable feature interactions (the easier, the better). In general, feature interaction is a key force for the understandability of features.

a.1.5.1) Connected and consistent feature descriptions are the principal prerequisite for feature interaction detection and specification.

a.2) Scaleability:

Another important aspect for the evolveability of the system is the scaleability of the chosen UCM style to a large system with hundreds of features. The scaleability depends foremost on connected and consistent feature descriptions (see a.1.5.1)) and to a lesser extent on the distribution and pollution of feature maps (see a.1.3) and a.1.4), respectively).

a.3) Reuseability:

To a lesser extent, the reuseability of UCMs across a number of features contributes positively to the evolveability of the system.

a.4) Ability to specify test cases:

To a lesser extent, the ability to use UCMs for the specification of test cases for single features and feature combinations contributes positively to the evolveability of the system. The ability to specify test cases for feature combinations depends on connected and consistent feature descriptions (see a.1.5.1)).

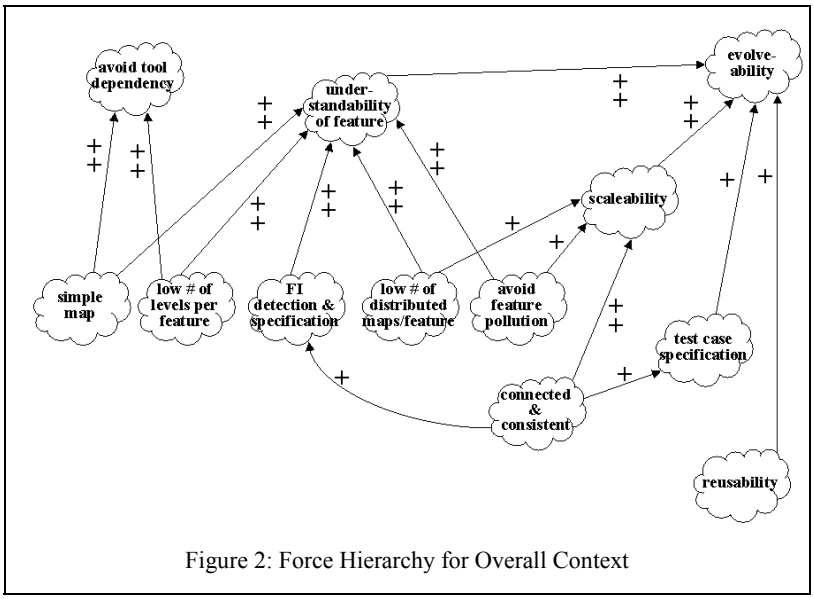


Figure 2: Force Hierarchy for Overall Context

b) Tool dependency:

A second issue to keep in mind is the tool dependency of the chosen UCM

style with respect to the distribution of UCMs to non-authors such as reviewers and new team members. UCMs can be read easily without the tool if it is easy to navigate through a series of UCMs. The ease of navigation relies on the simplicity of UCMs (in terms of the UCM path itself and the number of stubs per feature) (see a.1.1)) and the number of levels of UCMs per feature (see a.1.2)).

In general, most forces are connected with each other in more or less subtle ways. The force hierarchy shows only the important relationships which impact decisions. Some subtle connections, however, exist in addition to the ones shown and are addressed, if necessary, in the discussion of forces for each pattern.

3.1.4 Summary of Known Uses and Examples

All examples used in the following sections are taken from a call control software system but have been simplified and modified to protect confidential material. The changes have been carefully made as not to distort the occurrences of the discussed patterns. Although this paper focuses on examples from the telecommunication domain and cites known uses mainly from the same domain, we know of no reason why the patterns would not apply to other complex domains as well.

4 Individual Maps

Name of Pattern	Individual Maps
Brief Description	Independent feature descriptions are the best UCM style to be used in a context where the understandability of single features and tool independence are very important and the ability to specify test cases also has to be considered.
Confidence in Pattern	*** (out of *, **, or ***)
Discovered By	Ray Buhr
Authors	Gunter Mussbacher, Daniel Amyot
Shepherd	Jim Coplien
Other Reviewers	Rossana Andrade, Tom Gray, Michael Weiss
Date	October 5, 2001

4.1 Context

You are capturing functional requirements and high-level designs of a *few* key features of your system in order to get an initial understanding of the system and early feedback from stakeholders. Since you are creating a prototype rather than a complete specification, you will *not* be concerned if features interact with each other (i.e. one feature may impact the behavior of another feature either in a desirable way or in some unexpected or undesirable way). There is *no* need to reuse parts of features or even evolve a whole system efficiently but it must be possible to change single features *rapidly* and *independently* of other features. Often a pen and paper or a whiteboard rather than a tool are used to quickly create sketches of your prototype.

4.2 Problem

What is the best UCM style to be used in the given context?

4.3 Forces

All forces of the pattern collection have been described in 3.1.3. In the context of the "Individual Maps" pattern only some of these forces are relevant as indicated in Figure 4 to Figure 6. Note that the feature distribution, pollution, interaction, reusability, and scalability forces are not considered in this context because these five forces are not an issue if only a few features are captured.

If the "Standard Root Map" UCM style is applied in this context, the forces will not be sufficiently balanced (see Figure 4) even though the understandability force is reasonably well balanced.

- The understandability of a single feature is not as good as it could be because a higher number of stubs causes the maps to be a little bit more complex and at least one more level (the root level) is introduced as an additional level for all features.
- More importantly, tool dependency has increased because the stub-rich root map requires significantly more jumps between maps.
- It is possible but cumbersome to specify test cases for single features as definitions of features are buried in the map hierarchy.

If the "Isolation and Integration" UCM style is applied in this context, the forces are even less sufficiently balanced than with the "Standard Root Map" UCM style (see Figure 5). Both, understandability of single features and tool independence, are affected by the higher complexity required for the "Isolation and Integration" UCM style.

4.4 Solution

The "Individual Maps" UCM style describes each feature individually (see Figure 3). A hierarchy of one or more UCMs may be used to describe a feature but each UCM belongs solely to the described feature and is not reused for any other feature.

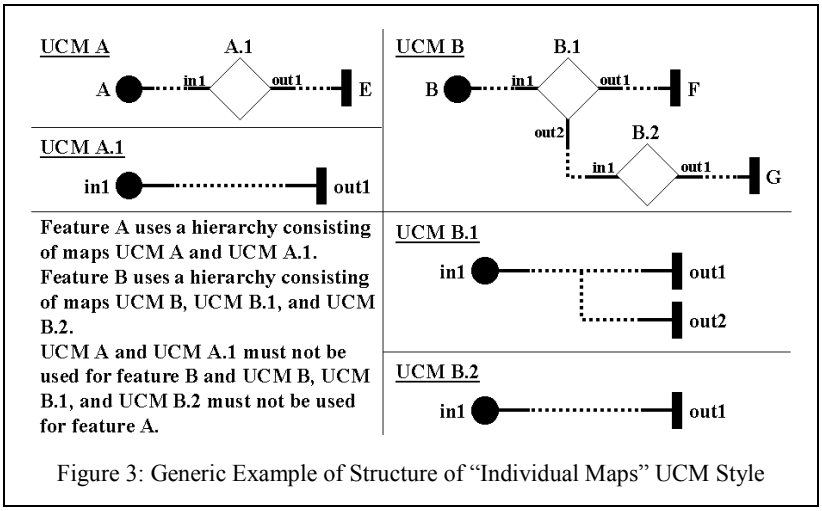


Figure 3: Generic Example of Structure of "Individual Maps" UCM Style

4.5 Resulting Context

Figure 6 shows how the solution balances the forces identified in 3.1.3. The "Individual Maps" UCM style balances well all relevant forces in the context because:

- The simplicity of UCMs and the number of levels of UCMs per feature are well balanced. This is as good as any approach can get since UCMs always have to trade-off simple

UCMs with additional levels of UCMs (i.e. number of stubs and plug-ins). Thus, the understandability of single features is perfectly balanced.

- Tool dependency is kept low because reading UCMs without the tool requires only the minimum amount of jumps between different levels of UCMs due to the optimal number of UCM levels.
- Test cases for single features can be specified quite easily due to the clear, unpolluted, and complete description of each feature.

Although the "Individual Maps" UCM style balances well tool dependency and understandability forces, this approach does not show the causal relationship between user actions (e.g. Figure 7 does not define whether *directory number* or *end call* can occur before *start call*). The pattern "Explicit Causality of Highest-Level Interaction" in section 8 addresses this problem.

As a consequence of using the "Individual Maps" UCM style, one cannot expect other concerns relevant in the overall context of the pattern language but not relevant in the sub-context of this pattern to be addressed.

- The style cannot detect

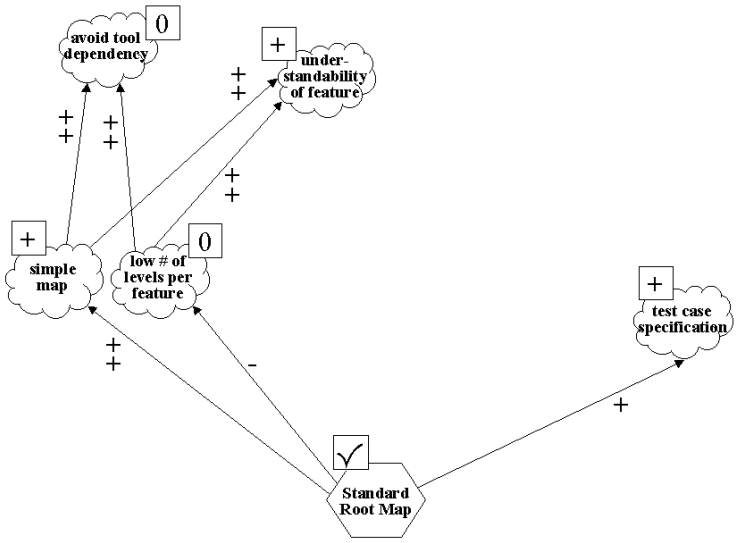


Figure 4: Impact on Individual Maps Forces by Standard Root Map

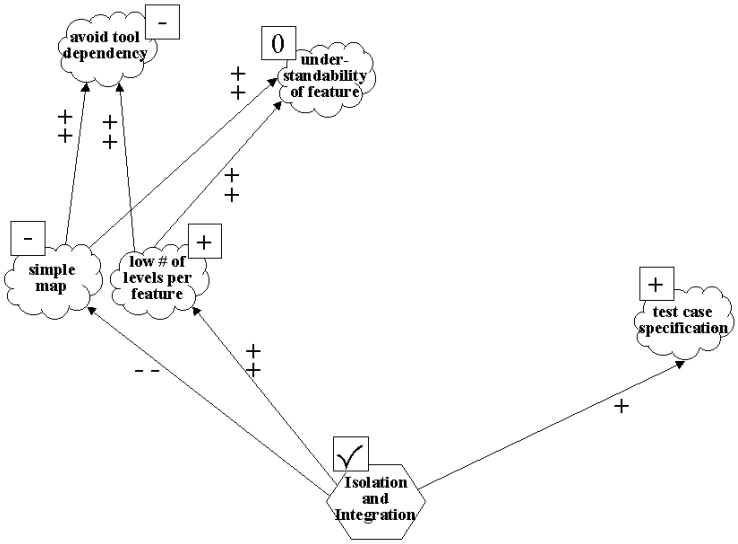


Figure 5: Impact on Individual Maps Forces by Isolation & Integration

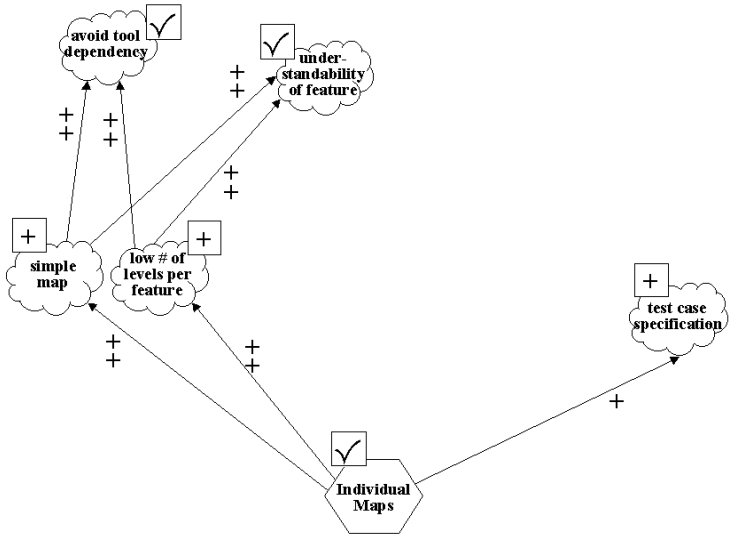


Figure 6: Impact on Individual Maps Forces by Individual Maps

interactions among features because consistency across features is not enforced. The style also cannot specify feature interactions because the definitions of different features are not linked.

- Test cases for feature combinations cannot be specified for the same reasons.
- Per definition of the style, reuse cannot be achieved.
- The approach does not scale well. Feature definitions are not linked thus more and more inconsistencies will potentially be introduced with each new feature. Although each new feature by itself requires only a similar effort to document than previous features, the amount of time eventually spent to work out the inconsistencies greatly increases.

4.6 Known Uses

For further examples of the "Individual Maps" UCM style in addition to the ones shown in Section 4.7 see [3, 5].

4.7 Examples of "Individual Maps" UCM Style

Examples of two features documented using the "Individual Maps" UCM style are shown in Figure 7 (Basic Call) and Figure 8 to Figure 10 (Automatic Call Distribution). Both examples use an underlying call model based on originating and terminating call halves.

4.7.1 Basic Call

The Basic Call feature (Figure 7) starts at the *start call* start point. The path then waits at the *dialing* waiting place until a directory number is entered (*directory number* start point). If a wrong number is entered the path takes a left turn and ends at *fail*. If a correct number is entered, the terminating call half is created (*create_TCH*).

The rest of the scenario depends on the device's availability. If the device is busy, the terminating call half follows the path labeled *[busy]*, ends at *idle.t*, and in parallel informs the originating call half which applies *busy* tone and ends at *fail*. If the device is not busy, the device rings, the terminating call half follows the path labeled *[not_busy]*, then waits at the *wfa.t* waiting place, and in parallel informs the originating call half which applies *ringback* tone and sets the *wait for answer* timer.

What comes next depends on whether the callee answers (*answer* start point) before the *wait for answer* timer expires. If the timer expires, the originating call half ends at *fail* and in parallel informs the terminating call half which follows the *[ring_timeout]*

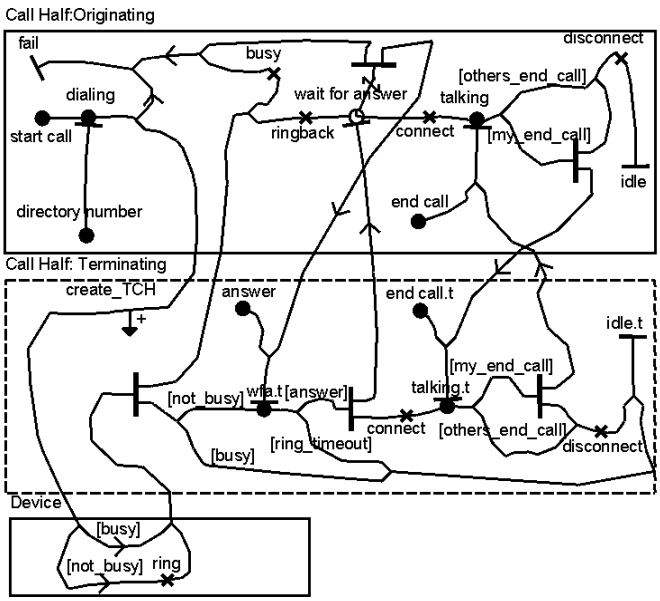


Figure 7: Basic Call (Individual Maps)

path and ends at *idle.t*. If the callee answers, the terminating call half follows the *[answer]* path, *connects*, and then waits at the *talking.t* waiting place. In parallel, the terminating call half informs the originating call half which *connects* and waits at the *talking* waiting place.

The rest of the scenario depends on whether the caller or the callee ends the call. If the caller ends the call (*end call* start point), the originating call half follows the *[my_end_call]* path, *disconnects*, ends at *idle*, and in parallel informs the terminating call half which follows the *[others_end_call]* path, *disconnects*, and ends at *idle.t*. If the callee ends the call (*end call.t* start point), the terminating call half follows the *[my_end_call]* path, *disconnects*, ends at *idle.t*, and in parallel informs the originating call half which follows the *[others_end_call]* path, *disconnects*, and ends at *idle*.

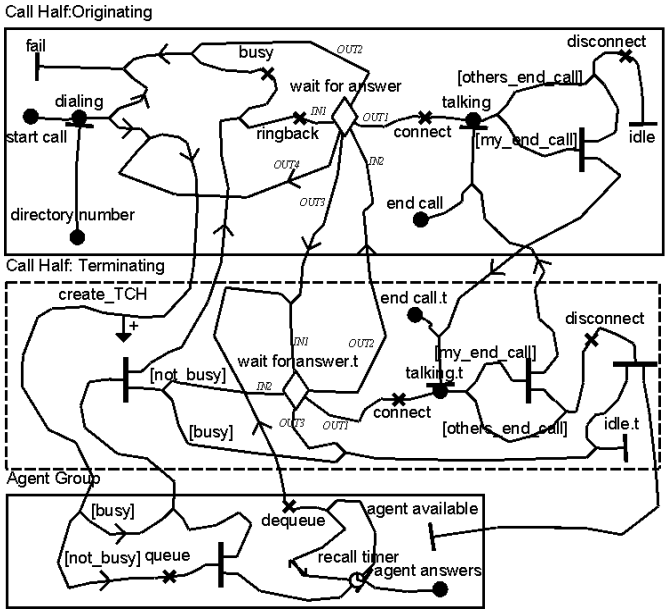


Figure 8: ACD (Individual Maps)

4.7.2 Automatic Call Distribution (ACD)

ACD distributes incoming calls to a group of agents according to their availabilities. The ACD feature (Figure 8, Figure 9, and Figure 10) duplicates a lot of the Basic Call feature. Therefore, only the changes to Basic Call behavior will be described. The biggest difference is that an agent group has replaced the device. In the case of a busy agent group, the agent group just informs the terminating call half. If the agent group is not busy, the agent group additionally *queues* the call and sets the *recall timer*.

The behavior of the originating and terminating call half is exactly the same as Basic Call behavior until the originating and terminating call halves are waiting at the *wait for answer* and *wfa.t* waiting places, respectively. The waiting places have been moved to two plug-in maps due to space constraints and readability concerns. The in and out labels on the UCMs show the binding between the parent map (ACD) and the plug-in maps (Wait For Answer, Wait For Answer.t).

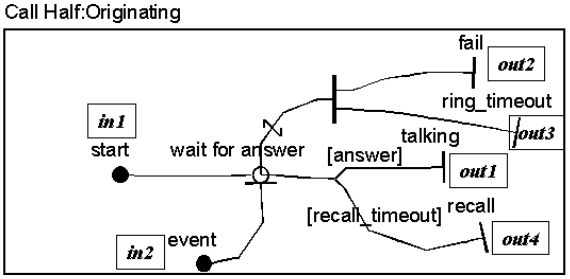


Figure 9: Wait For Answer (Individual Maps – ACD)

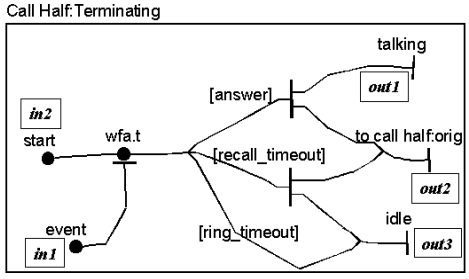


Figure 10: Wait For Answer.t (Individual Maps–ACD)

What comes next depends on whether the agent answers *(agent answers* start point) before the *recall* timer or the *wait for answer* timer expires. If the recall timer expires before the agent answers, the agent group *dequeues* the call and informs the terminating call half. This terminating call half follows the *[recall_timeout]* path (see Figure 10), ends at *idle.t*, and in parallel informs the originating call half which follows the *[recall_timeout]* path (see Figure 9) and tries to *recall* (i.e. the path loops back from the *wait for answer* stub and a new terminating call half is created (*create_TCH*)). If the agent answers, the agent group *dequeues* the call and informs the terminating call half. The terminating and originating call halves then follow normal Basic Call behavior.

Please note that the *wait for answer* timer is always set to a longer duration than the *recall* timer is. In case the wait for answer timer expires before the agent answers, a fatal error of the agent group may be assumed and normal Basic Call behavior occurs.

The last variation of Basic Call behavior occurs when either the caller or the agent *ends the call*. In addition to Basic Call behavior, the terminating call half informs the agent group after *disconnecting* that the *agent* is now *available* again.

5 Standard Root Map

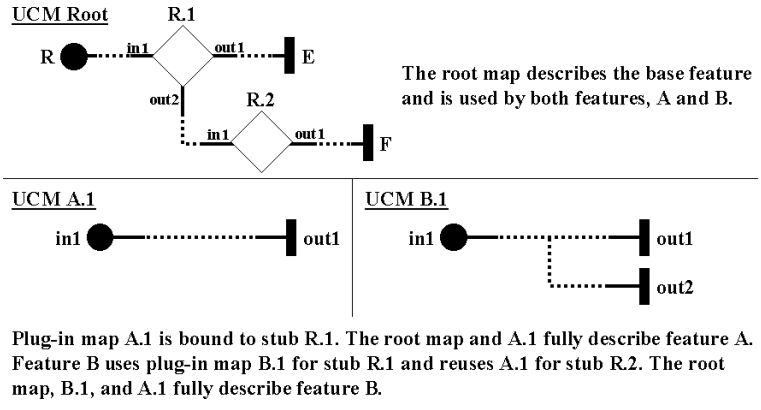
Name of Pattern	Standard Root Map
Brief Description	A combination of a root map, stubs, and plug-in maps is the best UCM style to be used in a context where evolveability, understandability of single features, and the ability to detect, specify, and understand feature interactions are very important, and reuseability across various features, the ability to specify test cases, and tool dependency also have to be considered.
Confidence in Pattern	*** (out of *, **, or ***)
Discovered By	Ray Buhr
Authors	Gunter Mussbacher, Daniel Amyot
Shepherd	Jim Coplien
Other Reviewers	Rossana Andrade, Tom Gray, Michael Weiss
Date	October 5, 2001

5.1 Context

You are capturing functional requirements and high-level designs of a *small* system that will be *evolving* over a long time or has been evolving for a long time. The features of the system may be *interacting* thereby increasing the system's complexity. Feature interaction occurs if one feature impacts the behavior of another feature either in a desirable way or in some unexpected or undesirable way. A base feature defining the framework for all other features usually but not exclusively indicates feature interactions. Most of the time new features are variations of existing features and build on the base feature and other features.

5.2 Problem

What is the best UCM style to be used in the given context?



Plug-in map A.1 is bound to stub R.1. The root map and A.1 fully describe feature A. Feature B uses plug-in map B.1 for stub R.1 and reuses A.1 for stub R.2. The root map, B.1, and A.1 fully describe feature B.

Figure 11: Generic Example of Structure of "Standard Root Map" UCM Style

5.3 Forces

All forces of the pattern collection have been described in 3.1.3. In the context of the "Standard Root Map" pattern only some of these forces are relevant as indicated in Figure 12 to Figure 14. Note that the feature distribution, pollution, and scaleability forces are not considered in this context because these three forces are not an issue if only a small system is captured.

If the "Individual Maps" UCM style is applied in this context, the forces will not be sufficiently balanced (see Figure 12) even though the tool dependency force is perfectly balanced.

- The style cannot detect interactions among features because consistency across features is not enforced. The style also cannot specify feature interactions because the definitions of different features are not linked. Therefore, the understandability force is not balanced sufficiently although feature maps are simple and the number of levels is low.
- Per definition of the style, reuse cannot be achieved.
- With mediocre understandability of features, no reuse, and test cases only for single features, evolveability is not balanced at all.

If the "Isolation and Integration" UCM style is applied in this context, the forces will not be sufficiently balanced (see Figure 13) even though the evolveability force is reasonably well balanced.

- The style is more dependent on tool support because of the complexity of the maps and the frequent use of stubs.
- The increased complexity of feature maps makes it more difficult to understand features although feature interaction detection and specification is possible and the number of levels per feature is kept low.
- Although consistency is enforced by the integration of features, the reuseability force is not as well balanced as it could be since more than one reusable unit may exist on a single UCM.

To summarize, a perfect solution should balance the tool dependency force and sub-forces a.1.1) and a.1.2) of the understandability force as the "Individual" UCM style does. Furthermore, a perfect solution should balance the feature interaction and specification force as the "Isolation and Integration" UCM style does but further improve the reusability and map complexity forces.

5.4 Solution

The "Standard Root Map" UCM style describes the base feature on a UCM called the root

map which contains several stubs (see Figure 11). The root map and the default set of plug-in maps for the stubs define the base feature. Other features may use one or more different plug-in maps and thus vary the behavior of the base feature. Since stubs may be used at any level of the map hierarchy, new features may introduce variations also at any level.

5.5 Resulting Context

Figure 14 shows how the solution balances the forces identified in 3.1.3. All in all, the "Standard Root Map" UCM style balances well the evolveability force at the expense of a slight increase in the complexity of UCMs and thus tool independence. The evolveability force is sufficiently balanced because:

- Feature interaction detection and specification is possible since all features are connected to each other even though it is cumbersome because feature definitions are buried in the map hierarchy.
- The understandability of a single feature is sufficiently balanced because feature interaction detection and specification is possible and feature maps are only slightly more complex.

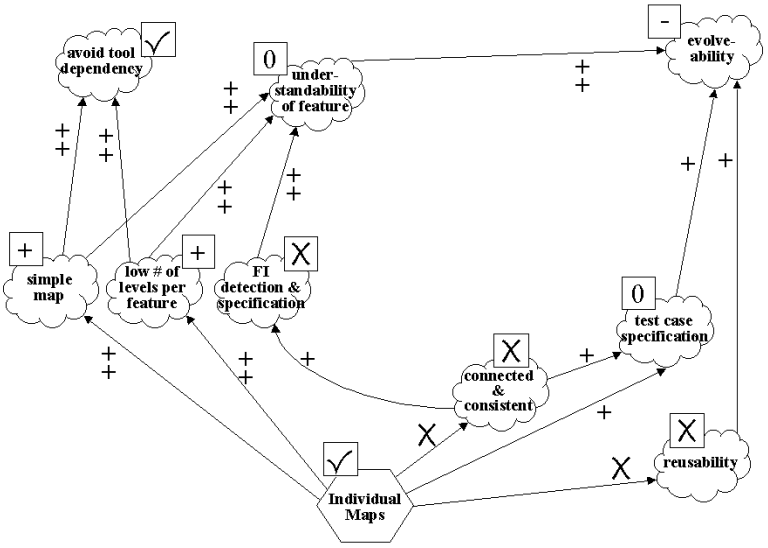


Figure 12: Impact on Standard Root Map Forces by Individual Maps

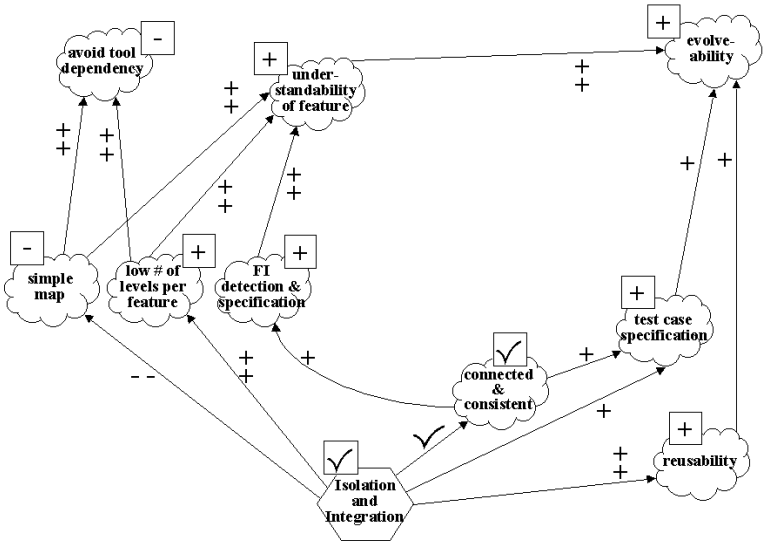


Figure 13: Impact on Standard Root Map Forces by Isolation & Integration

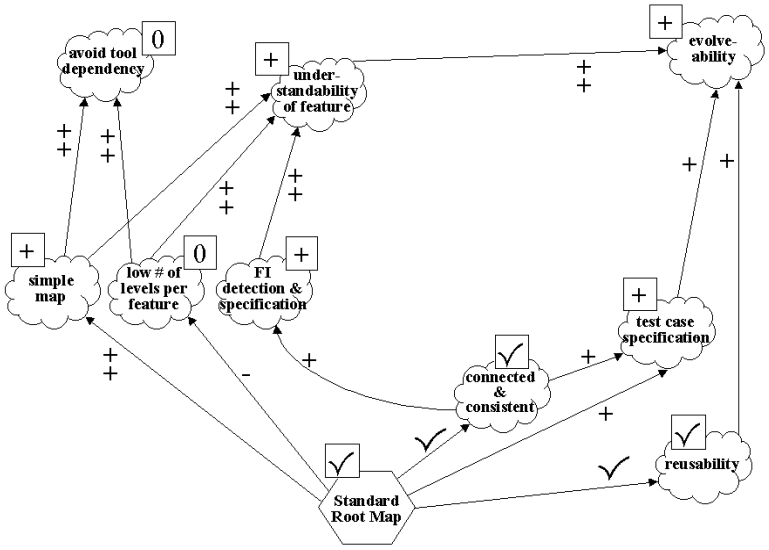


Figure 14: Impact on Standard Root Map Forces by Standard Root Map

- Reuseability can be achieved through the use of stubs with each plug-in map being one potential reusable unit.

Although the "Standard Root Map" UCM style balances well the evolveability force, this approach is more dependent on tool support than the "Individual Maps" UCM style because the stub-rich root map requires significantly more jumps between maps. The simplicity of UCMs is a little worse compared to the "Individual Maps" UCM style because the number of stubs has increased and the number of levels per feature has also increased by at least one level (the root level). Tool support and understandability, however, are still balanced better than with the "Isolation and Integration" UCM style.

This approach also does not show the causal relationship between user actions (e.g. Figure 15 does not define whether *directory number* or *end call* can occur before *start call*). The pattern "Explicit Causality of Highest-Level Interaction" in section 8 addresses this problem.

As a consequence of using the "Standard Root Map" UCM style, one cannot expect other concerns relevant in the overall context of the pattern language but not relevant in the sub-context of this pattern to be addressed.

- Feature maps do get polluted and are distributed which will cause problems if the system increases in size. Therefore, scaleability is not optimal yet although it has improved compared to the "Individual Maps" UCM style due to a greater degree of consistency.

5.6 Known Uses

For further examples of the "Standard Root Map " UCM style in addition to the ones shown in Section 5.7 see [6, 13, 17, 18].

5.7 Examples of "Standard Root Map" UCM Style

Examples of two features documented using the "Standard Root Map" UCM style are shown

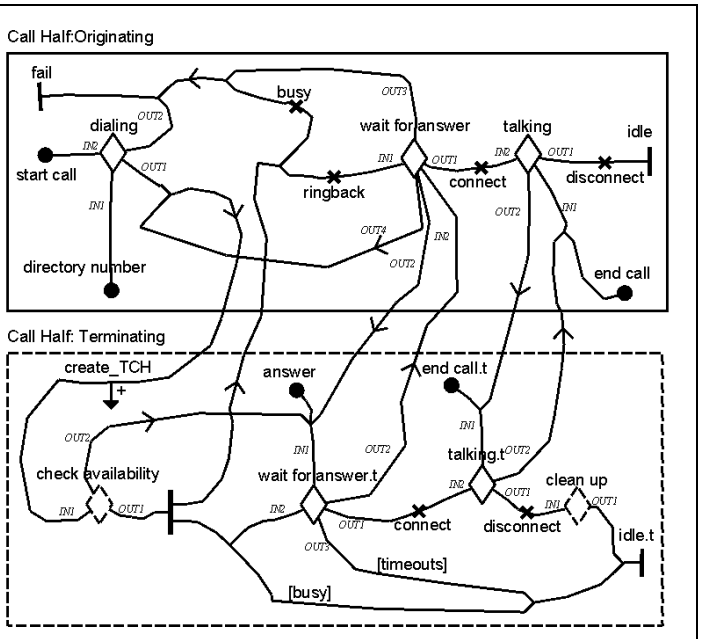


Figure 15: Basic Call root map (Standard Root Map)

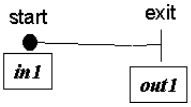


Figure 16: Nothing
(Standard Root Map – Basic Call)

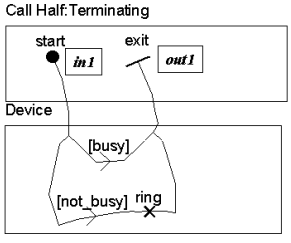


Figure 17: Check Availability
(Standard Root Map – Basic Call)

in Figure 15 to Figure 21 (Basic Call) and Figure 22 to Figure 25 (ACD). Both features start from the same root map (Figure 15).

5.7.1 Basic Call

The root map (Basic Call – Figure 15) and its six default plug-in maps (Dialing, Check Availability, Wait For Answer, Wait For Answer.t, Talking, Nothing) specify exactly the same behavior as explained for the "Individual Maps" UCM style (see 4.7.1).

Once again, in and out labels specify the bindings between stubs on the Basic Call map and their plug-in maps. The following list defines which stubs in Figure 15 contain which plug-in maps:

- the *check availability* stub contains the *Check Availability* plug-in (see Figure 17),
- the *dialing* stub contains the *Dialing* plug-in (see Figure 18),
- the *talking* and *talking.t* stubs contain the *Talking* plug-in (see Figure 19), and
- the *wait for answer* stub contains the *Wait For Answer* plug-in (see Figure 20),
- the *wait for answer.t* stub contains the *Wait For Answer.t* plug-in (see Figure 21),
- finally the *wait for answer.2* stub in Figure 20, *wait for answer.t.2* stub in Figure 21, and *clean up* stub in Figure 15 all contain the *Nothing* plug-in (see Figure 16).

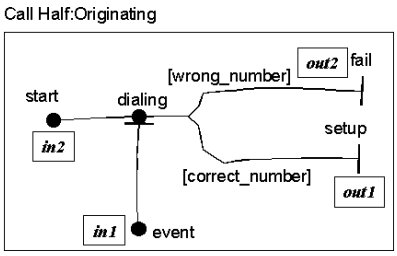


Figure 18: Dialing (Standard Root Map – Basic Call)

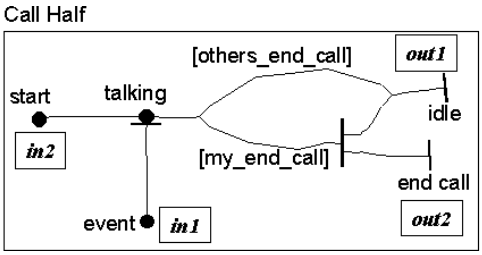


Figure 19: Talking (Standard Root Map – Basic Call)

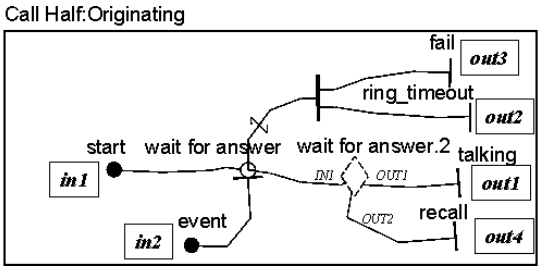


Figure 20: Wait For Answer
(Standard Root Map – Basic Call)

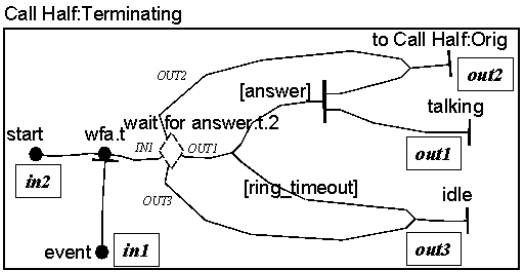


Figure 21: Wait For Answer.t
(Standard Root Map – Basic Call)

5.7.2 Automatic Call Distribution (ACD)

The new ACD plug-ins in conjunction with the Basic Call root map and plug-ins specify the exact same behavior as explained in the "Individual Maps" UCM style (see 4.7.2). The ACD feature overrides the Basic Call behavior by using the following four new plug-in maps:

- the *clean up* stub in Figure 15 contains the *Clean Up.ACD* plug-in (see Figure 22),
- the *check availability* stub in Figure 15 contains the *Check Availability.ACD* plug-in (see Figure 23),
- the *wait for answer.2* stub in Figure 20 contains the *Wait For Answer.ACD* plug-in (see Figure 25), and

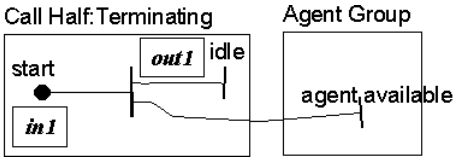


Figure 22: Clean Up.ACD
(Standard Root Map – ACD)

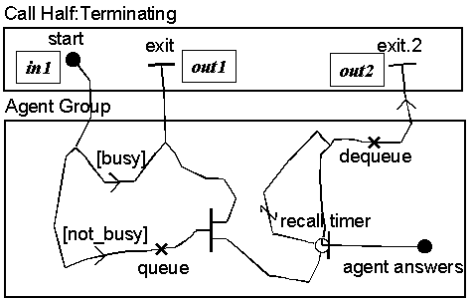


Figure 23: Check Availability.ACD
(Standard Root Map – ACD)

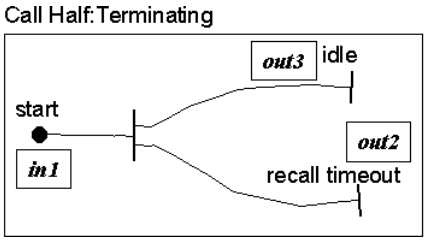


Figure 24: Wait For Answer.t.ACD
(Standard Root Map – ACD)

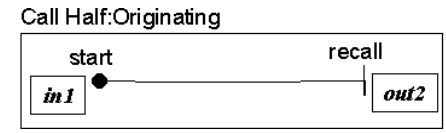


Figure 25: Wait For Answer.ACD
(Standard Root Map – ACD)

- the *wait for answer.t.2* stub in Figure 21 contains the *Wait For Answer.t.ACD* plug-in (see Figure 24).

6 Isolation and Integration of Features

Name of Pattern	Isolation and Integration of Features
Brief Description	Isolating features from each other but also linking them in a well-defined way is the best UCM style to be used in a context where evolveability, scaleability, understandability of single features, and the ability to detect, specify, and understand feature interactions are very important, and reuseability across various features, the ability to specify test cases, and tool dependency also have to be considered.
Confidence in Pattern	*** (out of *, **, or ***)
Discovered By	Gunter Mussbacher
Authors	Gunter Mussbacher, Daniel Amyot
Shepherd	Jim Coplien
Other Reviewers	Rossana Andrade, Tom Gray, Michael Weiss
Date	October 5, 2001

6.1 Context

You are capturing functional requirements and high-level designs of a *large* system that will be *evolving* over a long time or has been evolving for a long time. The system consists of *many interacting* features increasing its complexity. Feature interaction occurs if one feature impacts the behavior of another feature either in a desirable way or in some unexpected or

undesirable way. A base feature defining the framework for all other features usually but not exclusively indicates feature interactions. Most of the time new features are variations of existing features and build on the base feature and other features.

6.2 Problem

What is the best UCM style to be used in the given context?

6.3 Forces

All forces of the pattern collection have been described in 3.1.3. In the context of the "Isolation and Integration of Features" pattern all of these forces are relevant as indicated in Figure 26 to Figure 28.

If the "Individual Maps" UCM style is applied in this context, the forces will not be sufficiently balanced (see Figure 26) even though the tool dependency force and sub-forces a.1.1), a.1.2), a.1.3), and a.1.4) of the understandability force are well balanced.

- The style cannot support the detection of interactions among features because consistency across features is not enforced. The style also cannot specify feature interactions because the definitions of different features

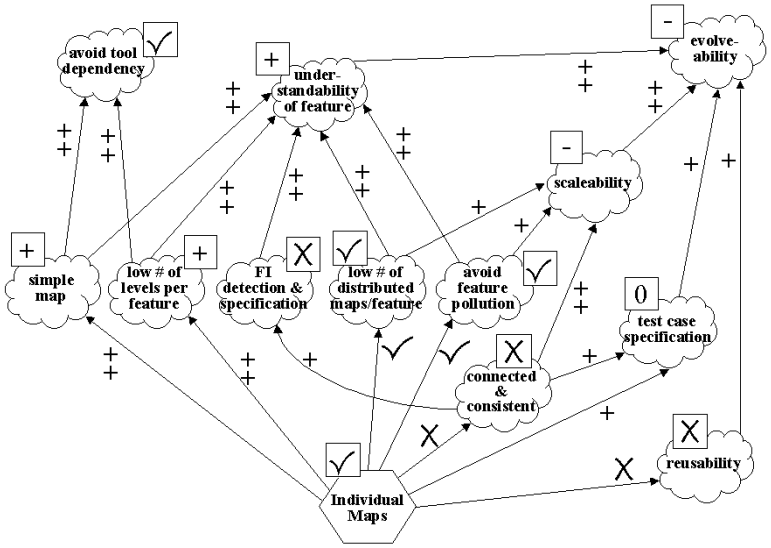


Figure 26: Impact on Isolation & Integration Forces by Individual Maps

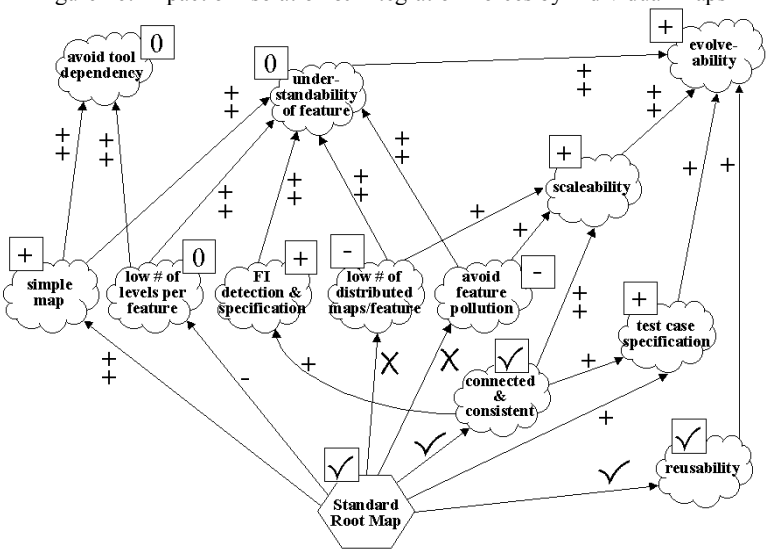


Figure 27: Impact on Isolation & Integration Forces by Standard Root Map

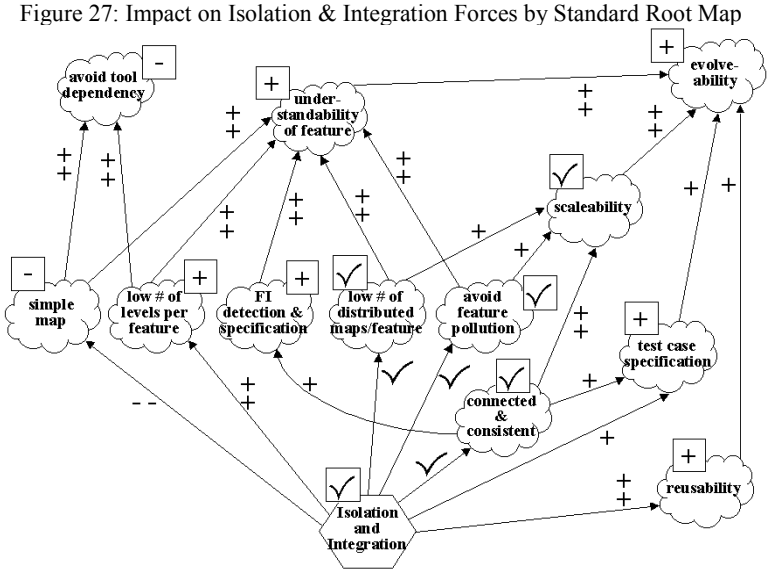


Figure 28: Impact on Isolation & Integration Forces by Isolation &Integration

are not linked. Therefore, the understandability force is only weakly balanced.
- Per definition of the style, reuse cannot be achieved.
- Test cases for feature interactions cannot be specified.
- The approach also does not scale well. Feature definitions are not linked thus more and more inconsistencies will potentially be introduced with each new feature. Although each new feature by itself requires only a similar effort to document than previous features, the amount of time eventually spent to work out inconsistencies greatly increases.

If the "Standard Root Map" UCM style is applied in this context, the forces will not be sufficiently balanced (see Figure 27) even though the reuseability force is perfectly balanced and the balance of the scaleability and feature interaction and detection forces is improved.

- Feature maps do get polluted now since all new features are plug-ins of at least the base root map. Therefore, variations of the base behavior caused by these new features do show up in the root map which should only show the base feature. E.g. the loop back from the *wait for answer* stub in Figure 15, out-path 2 of the *check availability* stub in Figure 15, and some out-paths of the *wait for answer.2* and *wait for answer.t.2* stubs in Figure 20 and Figure 21, respectively, are not required for Basic Call but exist because of ACD. Considering the large number of features in the system, feature maps get so polluted that the original description of the feature is effectively lost.
- Feature maps are now distributed. E.g. the set of UCMs specific to the ACD feature includes the UCMs in Figure 22, Figure 23, Figure 24, and Figure 25. These four UCMs are completely disjoint (one cannot move directly from any UCM to another). In contrast, the three ACD feature maps from the "Individual Maps" style (Figure 8, Figure 9, and Figure 10) are directly connected to each other. The reason for the distributed feature maps is that everything has to go through the root map which belongs only to the base feature. Considering a large number of features, it becomes difficult to find all UCMs that belong to a given feature. A naming scheme could help but breaks down when UCMs are being reused for various features. The UCM Navigator's functionality to group maps into sets could be used but navigation through the set still remains difficult. The main starting point for a feature is also not as apparent as it is in the "Individual Maps" UCM style.
- Scaleability improves because consistency is enforced to a greater degree. Scaleability, however, is not optimal yet because pollution and distribution of features remain a problem in large systems.
- Feature interaction detection and specification is now possible since all features are connected to each other but is cumbersome since complete feature definitions are buried at various levels in the map hierarchy and the specification of feature interactions further pollutes the description of features.
- The understandability of a single feature suffers considerably since feature maps get polluted, feature maps are distributed, and at least one more level (the root level) has been introduced as an additional level for all features.
- Tool dependency has also increased because the stub-rich root map requires significantly more jumps between maps.

Overall, this UCM style contributes positively but not sufficiently to evolveability because of a trade-off of reuseability and moderate amounts of feature interaction detection and specification, scaleability, and test case specification against the overall understandability of a single feature.

To summarize, a perfect solution should balance the tool dependency force and sub-forces a.1.1), a.1.2), a.1.3), and a.1.4) of the understandability force as the "Individual" UCM style does. Furthermore, a perfect solution should balance the reuseability force as the "Standard Root Map" UCM style does but further improve the scaleability and feature interaction detection and specification forces.

6.4 Solution

The "Isolation and Integration" UCM style isolates features from each other but does not keep these features completely separate. The features are linked to each other in a well-defined way. First, a root map for each feature provides initial isolation. Second, specific UCM structures identify locations where a link between two features may exist, thus further isolating features. The *event stub* structure is used for locations where the system is ready to deal with an event that may not be related to the feature. The *feature interaction (FI) fork* structure is used for locations where a variation of the feature may occur and this variation is caused by another feature. Once these two kinds of locations have been isolated, it is possible to integrate features by referencing the isolated locations.

The remainder of this section explains in more detail the event stub structure, the FI fork structure, and the referencing mechanism.

6.4.1 Event Stub Structure

The event stub structure is inserted at the first kind of location (see Figure 29 and Figure 30). Note that the FI fork structure is part of the event stub structure. The binding between the stub and the plug-in is defined as follows: (IN1, start), (IN2, event), (OUT1, success), (OUT2, fail), and (OUT3, feature).

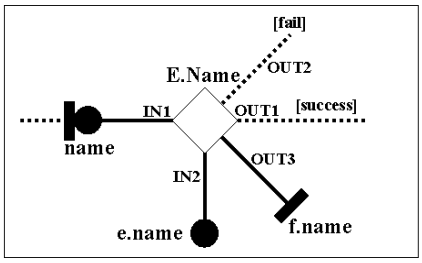


Figure 29: Structure of Event Stub

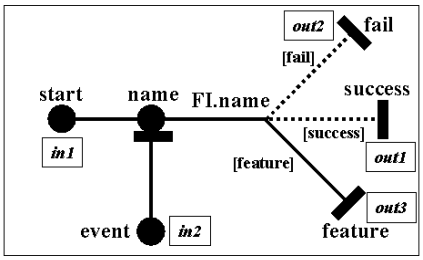


Figure 30: Structure of Plug-in for Event Stub

The event stub has two uses. First, if a UCM author wants to express that the current scenario is at the location represented by the event stub, path IN1 is taken. This positions the scenario at the *name* waiting place in the plug-in and the system is now waiting for an event. Second, if the UCM author wants to express that the scenario is continued because an event occurred, the source of the event is connected to the *e.name* start point and path IN2 is taken. This allows the scenario to continue to the FI fork labeled *FI.name*. At this point, the event that occurred is examined. If the event is known to the feature the event stub belongs to, one of the success or fail exits will be taken (one or more of these may be specified as required by the feature). If the event is unknown to the feature (i.e. another feature is changing the behavior of this feature – feature interaction!), then the feature exit will be taken. The FI fork is therefore defined as: "IF ((NOT success) AND (NOT

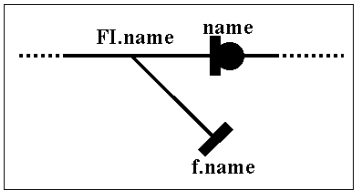


Figure 31: Structure of FI fork

fail)) THEN feature". Note that this definition is only dependent on the feature the event stub belongs to.

6.4.2 FI Fork Structure

At the second kind of location, only a subset of the event stub structure is required. Just the FI fork is added possibly followed by an end point connected to a labeled start point (see Figure 31). In this case, the definition of the FI fork explicitly states the reason for taking the feature exit (*f.name*) (e.g. "IF (ACD) THEN feature"). The FI fork indicates that the behavior of the feature is altered by another feature which will cause the scenario to exit at the feature exit. A guard (e.g. [feature]) may be used to label the branch to the feature exit (*f.name*) but is often omitted due to space constraints.

6.4.3 Referencing Mechanism

Features are integrated by referencing the isolated locations. The reference mechanism works as follows.

Each end point representing a feature exit implicitly sets a postcondition (see *f.name* in Figure 29 and Figure 31). The referencing feature then defines a start point on its UCM with an implicit precondition that matches the postcondition from the referenced feature exit. Thus, the scenario will continue from the referencing start point when the referenced feature exit is reached. This is achieved by labeling the end and start points the same (preferably with the feature exit name) and placing a guard right after the start point. The guard shows the actual precondition in square brackets (see *[recall_timeout]* and *[ACD]* in Figure 37) and allows multiplexing multiple scenarios caused by different events/preconditions onto one path.

Similarly, all *name* start points (see Figure 29 and Figure 31) define implicit preconditions. The referencing feature then defines an end point on its UCM with an implicit postcondition that matches the precondition of the *name* start point. Thus, the scenario will continue from the referenced start point if the referencing end point is reached. Once again this is achieved by labeling the end and start points the same (preferably with the name of the start point).

Sometimes, a referencing feature necessitates the insertion of an *end point/start point pair* into the map of the referenced feature. This pair consists of an end point connected to a labeled start point (see *recall* in Figure 32). The pair allows the definition of an entry point into an existing feature. This entry point can be used by another feature to continue a scenario with an existing feature. The same referencing mechanism is used for start points in end point/start point pairs as is used for start points in event stub or FI fork structures.

6.5 Resulting Context

Figure 28 shows how the solution balances the forces identified in 3.1.3. All in all, the "Isolation and Integration" UCM style balances well the evolveability force at the expense of the simplicity of the UCMs and thus tool independence. The evolveability force is sufficiently balanced because:

- Features are not polluted since the hooks required by other features can be clearly identified. A reader of the feature maps interested only in the feature itself can simply ignore all feature exits of event stubs, all FI forks, and all end points connected to start points. Since all features interacting with the current feature are multiplexed onto the same feature exit

paths, variations of behavior introduced by the interacting features do not require additional paths to be shown on the current feature's map (as it was the case with the "Standard Root Map" UCM style).

- Feature maps are not distributed since all feature maps are connected to and can be accessed from the feature's own root map.
- The scaleability force is well balanced because the pollution and distribution problems have been addressed.
- Feature interaction detection is as possible as with the "Standard Root Map" UCM style since features are integrated with each other. The specification of feature interactions, however, does not lead to feature pollution thus slightly improving the balance of the feature interaction force. The improvement, however, is not enough to affect the force hierarchy.
- Because consistency is enforced by the integration of features, the reuseability force remains relatively well balanced. The force, however, is not as well balanced as compared to the "Standard Root Map" UCM style since more than one reusable unit may exist on a single UCM.
- The overall understandability of the feature has been improved significantly from the "Standard Root Map" UCM style since features are now not polluted, feature maps are now not distributed, the total number of maps decreases compared to the "Standard Root Map" UCM style, and the number of levels per feature is kept low since a general root map is not introduced.

Although the "Isolation and Integration" UCM style balances well the evolveability force, this approach is more dependent on tool support than others because of the complexity of the maps and the frequent use of stubs. Furthermore, some problems regarding the navigability of UCMs and the degree of completeness of feature descriptions have not been addressed.

The complexity of the maps has two reasons. First, the event stub, FI fork, and end point/start point pair structures introduce a certain amount of complexity due to the number of required stubs. Second, the integration of features results in a somewhat less intuitive definition of a feature than the "Individual Maps" UCM style. Paths may be disjoint making it difficult to follow the causal flow (e.g. see Figure 37). Furthermore, no visual distinction between referencing start and end points and non-referencing start and end points is given, thus adding further complexity. In general, a navigation problem exists since the references between UCMs of different features are implicit and cannot be traversed by the tool and since the main start point of a feature is not apparent. The pattern "Explicit Navigation" in section 7 addresses these issues.

Moreover, the definitions of features without the referenced maps may not be as complete as one would want. For instance, the ACD feature in Figure 37 does not give you a clue of what happens after the *s2.t* end point. The reader of the UCMs has to look at the referenced map (the Basic Call root map) to find out. Therefore, a feature root map by itself does not quite define complete scenarios, rather only the extensions and variations. The patterns "Explicit Navigation" in section 7 and "Explicit Causality of Highest-Level Interaction" in section 8 address this problem.

Another issue is that user actions identified in Figure 7 and Figure 8 of the "Individual Maps" UCM style and in Figure 15 of the "Standard Root Map" UCM style (such as *start call*, *directory number*, *answer*, and *end call*) have been lost by the "Isolation and Integra-

tion" UCM style. In addition to the lost naming, the causal relationship between user actions is still not shown. This leads to a more complicated map because it is more difficult to understand the feature. The patterns "Explicit Causality of Highest-Level Interaction" in section 8 and "Context View of Feature" in section 9 tackle these issues.

6.6 Known Uses

Further examples of call control features that make use of the event stub structure in addition to the ones shown in Section 6.7 are Hold and Conference.

6.7 Examples of "Isolation and Integration" UCM Style

Examples of two features documented using the "Isolation and Integration" UCM style are shown in Figure 32 to Figure 36 (Basic Call) and Figure 37 (Automatic Call Distribution).

6.7.1 Basic Call

The Basic Call root map (Figure 32) and its four plug-in maps (Dialing, Wait For Answer, Wait For Answer.t, Talking) specify exactly the same behavior as explained for the "Individual Maps" style (see 4.7.1). To understand Basic Call, one can simply ignore all feature exits, all FI forks, and all end points connected to labeled start points. These structures provide only the hooks for other features to alter Basic Call behavior.

Once again, in and out labels specify the bindings between stubs on the Basic Call map and their plug-in maps. The naming convention for

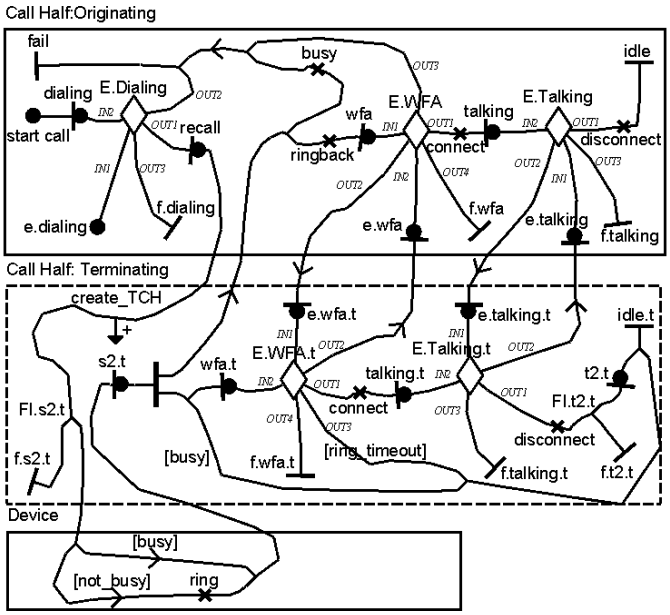


Figure 32: Basic Call root map (Isolation and Integration)

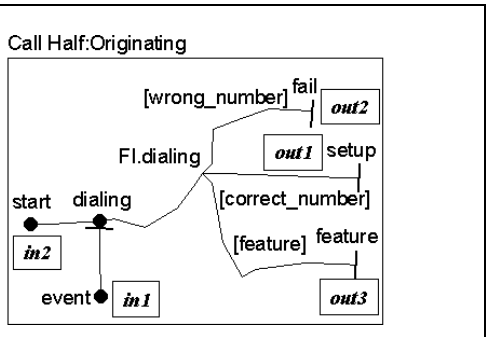


Figure 33: E.Dialing
(Isolation and Integration – Basic Call)

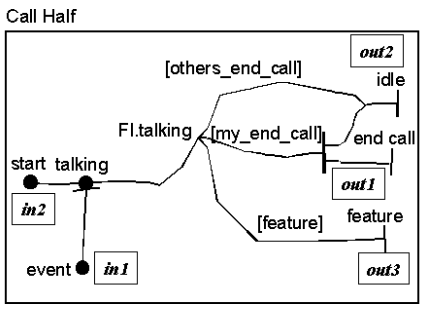


Figure 34: E.Talking
(Isolation and Integration – Basic Call)

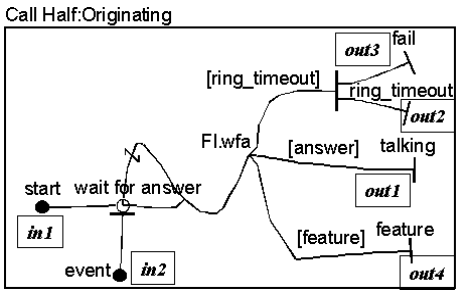


Figure 35: E.WFA
(Isolation and Integration – Basic Call)

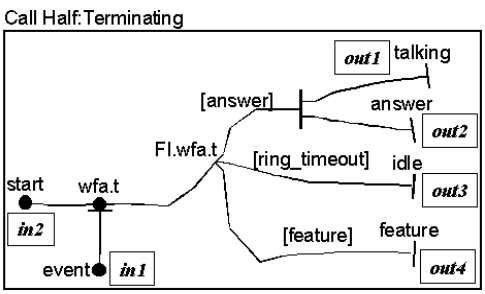


Figure 36: E.WFA.t
(Isolation and Integration – Basic Call)

event stubs is illustrated in Figure 29. The following list defines which stubs in Figure 32 contain which plug-in maps:

- the *E.Dialing* stub contains the *E.Dialing* plug-in (see Figure 33),
- the *E.Talking* and *E.Talking.t* stubs contain the *E.Talking* plug-in (see Figure 34),
- the *E.WFA* stub contains the *E.WFA* plug-in (see Figure 35), and
- the *E.WFA.t* stub contains the *E.WFA.t* plug-in (see Figure 36).

The event stub structure was applied five times to the Basic Call feature (*E.Dialing*, *E.WFA*, *E.WFA.t*, *E.Talking*, and *E.Talking.t*). The FI fork structure was applied twice (*FI.s2.t* and *FI.t2.t*) and the end point connected to a labeled start point was applied once (*recall*) as required by ACD. Note that the dynamic stubs from the "Standard Root Map" UCM style correspond to the FI forks of the "Isolation and Integration" UCM style. The following list shows the definitions of all FI forks of the Basic Call feature:

- *FI.s2.t* and *FI.t2.t* (both see Figure 32):
  IF (ACD) THEN feature
- *FI.dialing* (see Figure 33):
  IF ((NOT wrong_number) AND (NOT correct_number)) THEN feature
- *FI.talking* (see Figure 34):
  IF ((NOT my_end_call) AND (NOT others_end_call)) THEN feature
- *FI.wfa* (see Figure 35) and *FI.wfa.t* (see Figure 36):
  IF ((NOT ring_timeout) AND (NOT answer)) THEN feature

6.7.2 Automatic Call Distribution (ACD)

The new ACD root map (Figure 37) references the Basic Call root map to specify the same behavior as explained in the "Individual Maps" style (see 4.7.2). The following list defines which start and end points on the ACD root map align with which

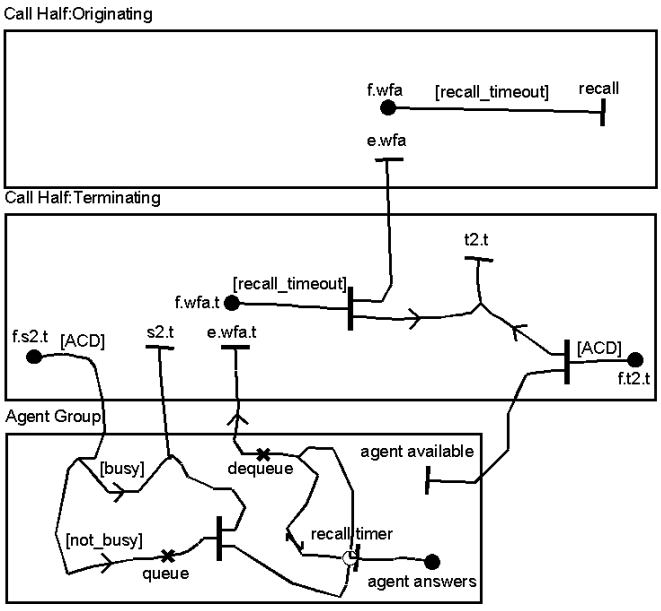


Figure 37: ACD root map (Isolation and Integration)

end and start points on the Basic Call root map:

- the *f.s2.t*, *f.wfa*, *f.wfa.t*, and *f.t2.t* start points on the ACD root map references the end points with the same name on the Basic Call root map,
- the *e.wfa*, *e.wfa.t*, *recall*, *s2.t*, and *t2.t* end points references the start points with the same name.

The following guards have been added to the feature description as required by the integration step:

- the *[ACD]* guards after the *f.s2.t* and *f.t2.t* start points, and
- the *[recall_timeout]* guards after the *f.wfa.t* and *f.wfa* start points.

Therefore, the ACD feature starts at the *f.s2.t* start point once the *f.s2.t* end point on the Basic Call root map is reached. After performing ACD specific actions, the scenario continues in Basic Call at start point *s2.t*. If the agent *answers* or the *recall timer* expires, the scenario reaches the *e.wfa.t* end point and therefore continues from the *e.wfa.t* start point in Basic Call. From the naming conventions, one can deduce that the *answer* scenario requires normal Basic Call behavior whereas the *recall timeout* scenario requires new behavior as shown on the ACD root map. Note how the path continues from *e.wfa.t* via the Basic Call root map to *f.wfa.t* if the *recall timeout* scenario occurs. The recall timeout scenario ends at *t2.t* and *e.wfa*. on the ACD root map. Therefore, it continues at the *t2.t* start point on the Basic Call root map and at the *f.wfa* start point (similarly to *e.wfa.t* and *f.wfa.t*) on the ACD root map. Finally, it ends at the *recall* end point which continues at the *recall* start point on the Basic Call root map.

7 Explicit Navigation

Disjoint paths and implicit references make it difficult to follow causal flow.
Therefore:
Use pre/postcondition stubs to join paths, navigate explicitly from map to map, and show complete scenarios.

8 Explicit Causality of Highest-Level Interaction

Ambiguous feature descriptions are often caused by implicit causal relationships of highest-level user interactions with the system.
Therefore:
Use paths with “at-location” markers to explicitly show highest-level causal relationships.

9 Context View of Feature

Explicit highest-level causal relationships on a single map and lost naming of user actions make UCMs unnecessarily complex.
Therefore:
Use higher level UCMs for clear “Context View” of features.

10 Conclusion

The characteristics of a system determine the most effective UCM style. If a few key features need to be captured rapidly and independently, the "Individual Maps" UCM style is most useful. If a small but evolving system consisting of interacting features needs to be specified, the "Standard Root Map" UCM style is most appropriate. If a large, evolving system with many interacting features needs to be documented, the "Isolation and Integration" UCM style is best applied. This paper introduces patterns for each UCM style, each pattern providing guidelines on how and when to use the style.

Future work includes the extension of this pattern collection with a more detailed description of the three patlets and four additional patterns. Furthermore, the existing patterns need to be reevaluated regularly since changes to the UCM Navigator tool may impact these patterns. Finally, the patterns themselves may impact or spur the development of new features for the UCM Navigator tool and other such tools.

11 References

1. Amyot, D., *Use Case Maps as a Feature Description Language*, in S. Gilmore and M. Ryan (Eds), *Language Constructs for Designing Features*, pp. 27-44, Springer-Verlag, 2000.
2. Amyot, D. and Andrade, R., *Description of Wireless Intelligent Network Services with Use Case Maps*, in *SBRC'99, 17th Brazilian Symposium on Computer Networks*, Salvador, Brazil, May 1999.
3. Amyot, D. and Logrippo, L., *Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System*, in *Computer Communication*, 23(8), April 2000.
4. Amyot, D., Buhr, R.J.A., Gray, T., and Logrippo, L., *Use Case Maps for the Capture and Validation of Distributed Systems Requirements*, in *RE'99, Fourth IEEE International Symposium on Requirements Engineering*, pp. 44-53, Limerick, Ireland, June 1999.
5. Amyot, D., Logrippo, L., and Buhr, R.J.A., *Spécification et conception de systèmes communicants : une approche rigoureuse basée sur des scénarios d'usage*, in *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'97)*, pp. 159-174, Hermes, Paris, 1997.
6. Andrade, R., *Applying Use Case Maps and Formal Methods to the Development of Wireless Mobile ATM Networks*, in *Lfm2000: The Fifth NASA Langley Formal Methods Workshop*, Williamsburg, Virginia, USA, June 2000.
7. Andrade, R. and Logrippo, L., *Reusability at the Early Development Stages of the Mobile Wireless Communication Systems*, in *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2000), Vol. VII, Computer Science and Engineering: Part I*, pp. 11-16, Orlando, Florida, July 2000.
8. Buhr, R.J.A., *Use Case Maps as Architectural Entities for Complex Systems*, in *Transactions on Software Engineering*, pp. 1131-1155, IEEE, December 1998.
9. Buhr, R.J.A. and Casselman, R.S., *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA, 1995.
10. Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S., *High Level, Multi-agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Ex-*

ample, in *PAAM'98, 3rd Conference on Practical Application of Intelligent Agents and Multi-Agents*, London, UK, March 1998.
11. Cameron, D. et al., *Draft Specification of the User Requirements Notation*, Canadian contribution CAN COM 10-12 to ITU-T, November 2000.
12. Chung, L., Nixon, B.A., Yu, E., and Mylopoulos, J., *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, 2000.
13. Elammari, M. and Lalonde, W., *An Agent-Oriented Methodology: High-Level and Intermediate Models*, in *Proceedings of the 1st International Workshop on Agent-Oriented Information Systems (AOIS'99)*, Heidelberg, Germany, June 1999.
14. *Goal-oriented Requirement Language (GRL) Web Site*, 2001, http://www.cs.toronto.edu/km/GRL/.
15. Gross, D. and Yu, E., *From Non-Functional Requirements to Design through Patterns*, in *Requirements Engineering*, 6:18-36, Springer-Verlag, 2001.
16. Miga, A., *Application of Use Case Maps to System Design with Tool Support*, M.Eng. thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, October 1998, http://www.UseCaseMaps.org/tools/ucmnav/.
17. Miga, A., Amyot, D., Bordeleau, F., Cameron, D., and Woodside, M., *Deriving Message Sequence Charts from Use Case Maps Scenario Specifications*, 10th SDL Forum, Copenhagen, Denmark, June 2001.
18. Nakamura, M., Kikuno, T., Hassine, J., and Logrippo L., *Feature Interaction Filtering with Use Case Maps at Requirements Stage*, in *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00)*, Glasgow, Scotland, UK, May 2000.
19. Sales, I. and Probert, R., *From High-Level Behaviour to High-Level Design: Use Case Maps to Specification and Description Language*, in *SBRC 2000, 18th Brazilian Symposium on Computer Networks*, Belo Horizonte, Brazil, May 2000.
20. Scratchley, W.C. and Woodside, C.M., *Evaluating Concurrency Options in Software Specifications*, in *MASCOTS'99, Seventh International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 330-338, College Park, MD, USA, October 1999.
21. *Use Case Maps Web Site* and *UCM User Group*, 1999, http://www.UseCaseMaps.org.
22. Weiss, M., *Patterns and Non-Functional Requirements*, Presentation at CITO Software Research Review, March 2001, http://fusion.scs.carleton.ca/~weiss/research/nfr/cito.pdf.
23. Zave, P., *Requirements for evolving systems: A telecommunications perspective*, in *RE'01, Fifth IEEE International Symposium on Requirements Engineering*, pp. 2-9, Toronto, Canada, August 2001, http://www.research.att.com/~pamela/re1.pdf.

A Pattern Language for Providing Client-Server Confidential Communication

Jerffeson Teixeira de Souza *
School of Information Technology and Engineering,
University of Ottawa
K1N 6N5, Canada
jsouza@site.uottawa.ca

Stan Matwin
School of Information Technology and Engineering,
University of Ottawa
K1N 6N5, Canada
stan@site.uottawa.ca

Abstract

This paper extracts and documents patterns that identify problems and solutions concerning confidentiality in a client-server environment. These patterns are then organized as a pattern language. The idea is to include a new layer that is responsible for providing the security framework. This layer is composed by a Client Secure Socket and a Server Secure Socket. In order to obtain confidentiality, a combination of symmetric and asymmetric (public/private) cryptography techniques is proposed. For data encryption is proposed the use of the symmetric system with a Session Key. And for exchanging the Session Key, the public/private key pair model is used. This combination provides a fast and reliable cryptosystem.

Keywords: Confidentiality, Client-Server Communication, Cryptography, Pattern Language.

1 Introduction

The confidentiality of the data exchanged in a business environment has become more and more important. The design and implementation of a secure communication channel is a hard and expensive task. The idea of this work is to guide the development of a client-server application

*Sponsored by CAPES (Brazilian Federal Agency for Graduate Studies).

that requires confidential communication. In order to do that, the combination of two cryptography approaches: Symmetric and Asymmetric is suggested. The goal is to use the advantages of each approach and keep the performance of the system high. This paper presents a collection of patterns that help developers create client-server applications. The focus of this paper is on the confidentiality of the data exchanged in the client-server communication. It was designed to be used by developers that need to include security features in their client-sever applications.

The patterns presented in this paper were conceived when I was required to develop a client-server application that should implement some security features. During my research, I realized that most of the existing client-server applications used a very similar variation of key exchange. From this observation, I started extracting the common solutions from this applications and creating this pattern language to document such solutions so that new developments could be facilitated.

2 Background

Cryptography can be defined as the art or science encompassing the principles and methods of transforming an intelligible message into one that is unintelligible, and then retransforming that message back to its original form [4].

2.1 Symmetric Cryptography

In symmetric cryptography, the encryption algorithm requires the same secret key (called Session Key) to be used for both parts of the communication [4]. Because of the type of the key, this process is called secret key encryption. With this approach, an original message is encrypted with a Session Key and sent to the receiver. The receiver decrypts this message with the same Session Key to retrieve the original message.

The advantage of these algorithms is that they are fast and efficient. However, the problem is the key exchange, i.e., the mechanism for safely ensuring both parties, the sender and the receiver, have the secret key. This is one of the weakest areas of symmetric cryptography. How do you send the key to your partners? You cannot just send it in an e-mail message, because it could be intercepted and compromise your security. Furthermore, how can you be sure that your partners will keep your key secure?

A variant of this approach works with two Session Keys, one to each part of the communication. In each part, one key is used for encryption and the other for decryption. This variant is used in this work.

2.2 Asymmetric (Public/Private Key) Cryptography

One solution to the problem of symmetric key security is asymmetric cryptography. This uses two keys that are mathematically related. One key is called the private key and is never revealed, and the other is called the public key and is freely given out to all potential correspondents [4]. The complexity of the relationship between the public key and the private key means that, provided the keys are long enough, it is practically impossible to determine one from the other. The one

problem with asymmetric cryptography is that it is CPU usage is very high and this can cause potential performance problems when many simultaneous sessions take place.

The almost universal public/private key algorithm is named RSA [5]. A sender uses the receiver's public key to encrypt the message. Only the particular receiver has the respective private key to decrypt the message. In addition, it is important to inform that the patent for the RSA algorithm (used is this work), that was issued on September 20, 1983, exclusively licensed to RSA Security Inc. by the Massachusetts Institute of Technology, expired on September 20, 2000. Since then, RSA Security is making the RSA algorithm publicly available and waiving its rights to enforce the RSA patent for any development activities including this algorithm occurring since September 2000 [5].

3 Guidelines for the Readers

The patterns used here have a form consisting of eleven sections. The *Context section* shows the situations in which the pattern may be applied. The *Problem* section presents a question that expresses the problem this pattern solves. The *Forces* section describes the driving forces behind possible solutions. The *Solution* section presents an answer to the question from the problem section that resolves the forces as well as possible. The *Structure* section is a detailed specification of the structural aspects of the pattern. The *Dynamics* section describes the run-time behavior of the pattern. The *Rationale* section explains the rationale behind the solution. The *Sample Code* section presents code fragments that illustrate how the pattern can be implemented in Java. The *Resulting Context* section describes the context that we find ourselves in after the pattern has been applied. The *Known Uses* section describe places where the pattern is used. Finally, The *Related Patterns* section describes any related patterns. Examples of these sections can be found in [3] and [2]. Some sections are not presented in some patterns.

The problem and the solution sections are sufficient to get an overview of the pattern. The other sections explain the rationale behind the pattern and allow the readers to gain a deeper insight of it.

4 The Pattern Language

4.1 Overview

The pattern language presented in this paper has five patterns. The relationship among these patters is showed in Figure 1. As we can see, the Client-Server Secure Communication Pattern is the main one. All the others are proposed to specialize the solution presented in the first pattern.

The following table shows all patterns summarizing their problems and the respective solutions.

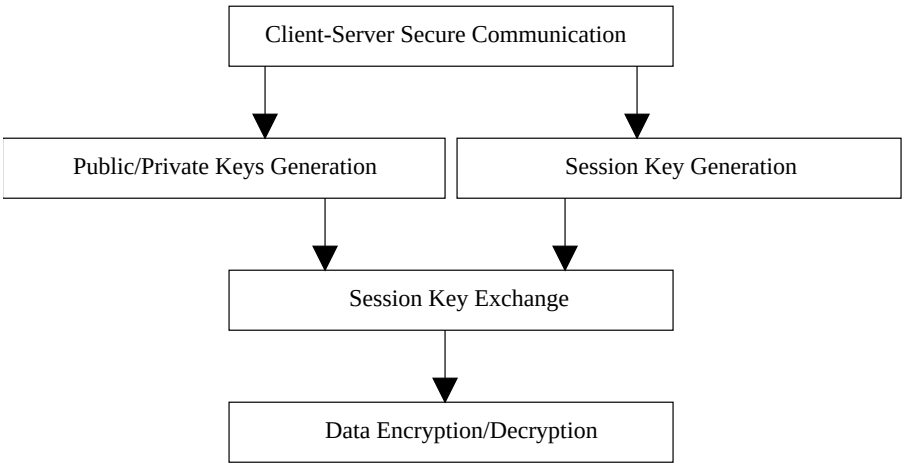


Figure 1: The Pattern Language Structure

Pattern	**Problem**	**Solution**
Client-Server Secure Communication	• How to provide confidentiality in a client-server communication?	• Provide an extra layer with two components: Secure Client Socket and Secure Server Socket that encrypts and decrypts the data. • Use a combination of symmetric and asymmetric cryptography techniques.
Public/Private Key Generation	• How to generate a public/private key pair?	• Get the key pair by using the RSA algorithm.
Session Key Generation	• How to generate a Session Key?	• Set the session key to a random number.
Session Key Exchange	• How to exchange the session keys?	• Use an asymmetric approach to send the Servers Session Key to the Client and vice-versa.
Data Encryption/ Decryption	• How to encrypt and decrypt data with the session key?	• Produce encrypted data as an exclusive-or of the original data and the key.

4.2 The Patterns

4.2.1 Client-Server Secure Communication

Context

- You are developing a Client-Server application where the confidentiality of the exchanged data in the communication is important.

Problem

- How do you provide confidentiality in a client-server communication?

Forces

- efficiency - Data encryption must be as fast as possible.
- independence - Users do not have to worry about security.
- security - The more secure the system is, the better.

Solution

- Create a security layer to be included in the communication environment so that this layer is responsible for guaranteeing the confidentiality of the data exchanged.
- Create two components to this new layer: a Client Secure Socket and a Server Secure Socket. These components implement all security features. These are responsible for establishing the communication parameters and for encrypting and decrypting the data, so that both client and server can work independently of this new layer.
- Then, generate a public/private key pair (*Public/Private Key Generation*) and a Session Key (*Session Key Generation*) to the Client and the Server.
- Use a combination of symmetric and asymmetric (public/private key) cryptography techniques. To encrypt and decrypt data (*Data Encryption/Decryption*) use a symmetric key (Session Keys). To exchange the symmetric key (*Session Key Exchange*) use an asymmetric algorithm.

Structure

The Client Secure Socket must be placed in the client side. All data received and sent by the client must pass through by this component. The same happens in the server side. When the client sends a piece of information to the server, the Client Secure Socket gets it, encrypts it and sends it to the server.

On the other side, the Server Secure Socket intercepts the message, decrypts it and deliveries it to the Server so that both client and server can work independently of this new layer. The structure is showed below:

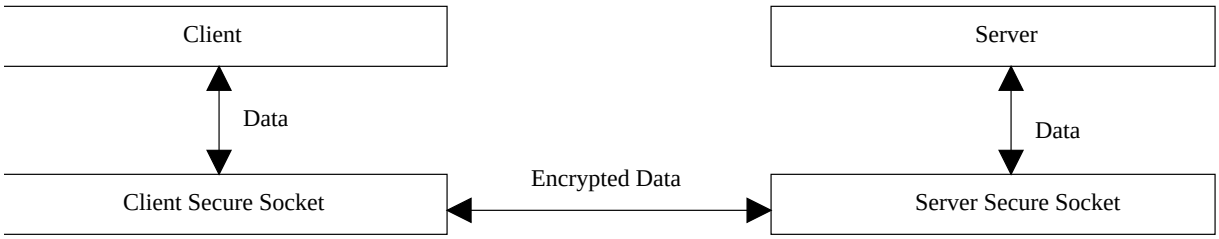


Figure 2: New Layer Structure

Dynamics

Before the data exchanging starts, it is necessary that the Client Secure Socket and Server Secure Socket have some parameters such as Session Keys and Public Keys. These parameters come from the initialization process that takes place in the beginning of the communication. This initialization includes the following steps:

1. The Client Secure Socket generates its Session Key;
2. The Server Secure Socket generates its Session Key;
3. The Client Secure Socket generates the Client's Public/Private key pair;
4. The Server Secure Socket generates the Server's Public/Private key pair;
5. The Client Secure Socket sends the Client's Public Key to the Server side;
6. The Server Secure Socket sends the Server's Public Key to the Client side;
7. The Client Secure Socket sends to the Server Secure Socket its Session Key encrypted with Server's Public Key;
8. The Server Secure Socket receives and decrypts the Client's Session Key with Server's Private Key;
9. The Server Secure Socket sends to the Client Secure Socket its Session Key encrypted with Client's Public Key;
10. The Client Secure Socket receives and decrypts the Server's Session Key with Client's Private Key;

In order to exchange data after the initialization, the following steps must be done:
Data from the Client to the Server:

1. The Client delivers the data to the Client Secure Socket;
2. The Client Secure Socket receives and encrypts the data with the Server's Session Key;
3. The Client Secure Socket sends the data to the Server side;
4. The Server Secure Socket intercepts and decrypts the data with the its Session Key;
5. The Server Secure Socket delivers the data to the Server.

Data from the Server to the Client :

1. The Server delivers the data to the Server Secure Socket;
2. The Server Secure Socket receives and encrypts the data with the Clients's Session Key;

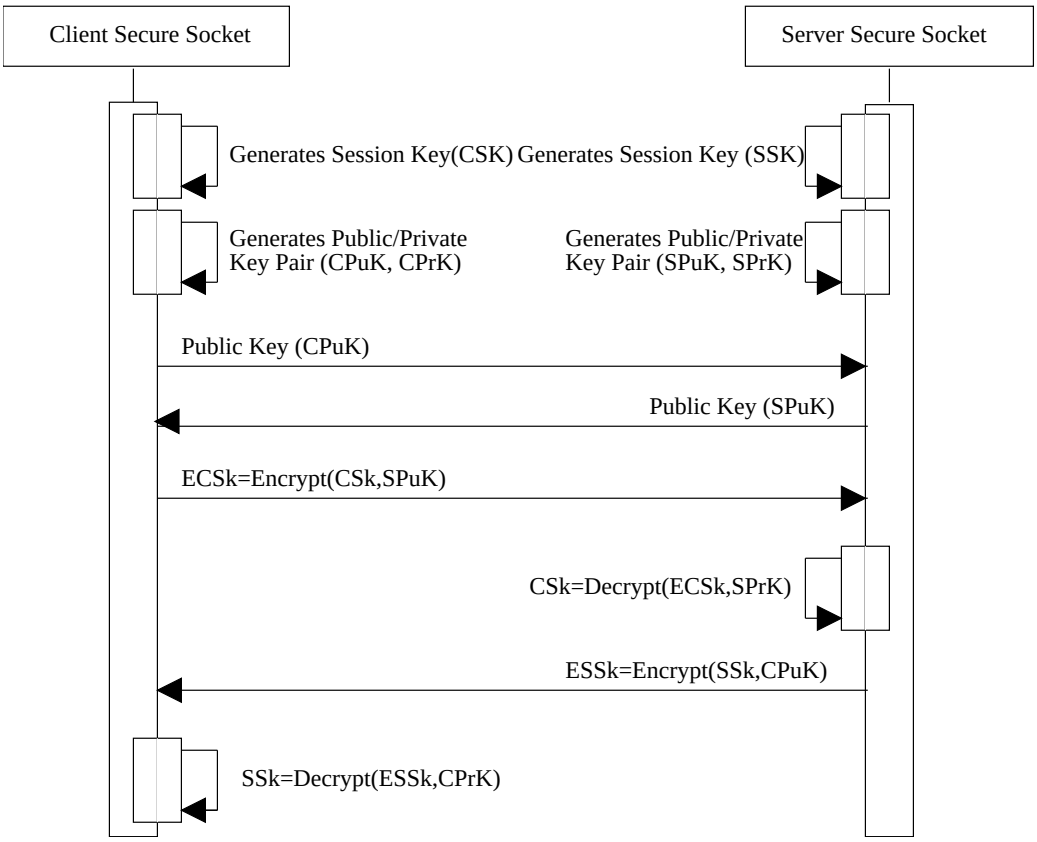


Figure 3: Initialization Process

3. The Server Secure Socket sends the data to the Client side;
4. The Client Secure Socket intercepts and decrypts the data with the its Session Key;
5. The Client Secure Socket delivers the data to the Client.

Rationale

The symmetric approach is able to process the encryption of large messages in a very fast way. However, its security relies on the key exchange. The asymmetric approach is a much more secure encryption technique, but the problem with such technique is that it requires an intensive CPU slice and this can cause potential performance problems when many simultaneous sessions take place.

The combination of symmetric and asymmetric cryptography techniques provides us with a reliable and fast security framework. The data is exchanged by using a pair of symmetric keys, one for the client and one for the server, proportioning a really fast solution. The use of two session keys makes the system more secure. The symmetric keys (session keys), that are small pieces of information, are exchanged by using the asymmetric approach. As this approach is much secure, it guarantees the security of the symmetric keys and consequently of the whole system.

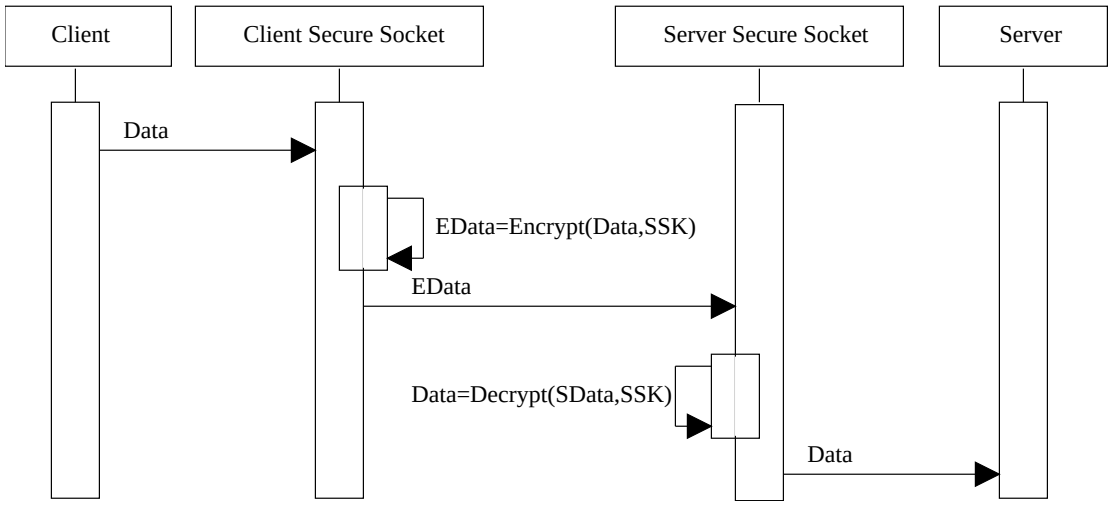


Figure 4: Data Exchange (from Client to Server)

Note that the security of the session keys exchanged in the initialization process is in fact that, as the session key is encrypted with the receiver's public key, only the receiver can decrypt this message and get the session key, because only the receiver knows the correspondent private key.

Resulting Context

The application of this pattern creates a security framework that enables data encryption in a reliable and fast basis. However, there are some questions that must be answered before a real system can be built. These questions concern the generation of the symmetric (*Session Key Generation*) and asymmetric (*Public/Private Key Generation*) keys, the exchange of the symmetric keys (*Session Key Exchange*) and the data exchange (*Data Encryption/Decryption*).

Know Uses

The combination of symmetric and asymmetric cryptographic approaches is used by the Secure Socket Layer (SSL) Protocol [6] that runs over TCP/IP to provide security to communication between clients and servers.

Related Patterns

The *Information Secrecy* and *Cryptographic Metapattern* are presented in [1] within a pattern language for cryptographic software.

4.2.2 Public/Private Keys Generation

Context

- You are developing a Client-Server application and you are applying the Client-Server Secure Communication Pattern.

Problem

- How do you generate a public/private key pair?

Forces

- efficiency - The generation of the key pair must consume the least possible amount of time.
- security - It must be as hard as possible to discover the Private Key from the Public Key.

Solution

- Follow the algorithm below to generate the key pair :

  1. Choose two large prime numbers p and q, with 512 bits of length or longer.
  2. Calculate $n = p * q$.
  3. Calculate $f(n) = (p - 1) * (q - 1)$.
  4. Find an integer e which is a relative prime to $f(n)$, i.e., a number e that has no common divisors with $f(n)$.
  5. Calculate d, the inverse of e modulo $f(n)$, i.e., $d = e^{-1}\ mod\ f(n)$.
  6. Set the Public Key to $\{e, n\}$.
  7. Set the Private key to $\{d, n\}$.

Rationale

The efficiency of the algorithm above is based on hardness of factorization, in the sense that it is easy to multiply two big prime numbers, but for most very large primes, it is extremely time-consuming to factor them. This way, the algorithm provides a fast way to generate the key pair and makes it infeasible to discover p and q from $n = p * q$. These characteristics guarantee that it is unfeasible to get the Private Key from the Public Key.

Resulting Context

With theirs public/private key pair, client and server are now able to apply the asymmetric cryptographic approach for data exchanging in a safe basis (*Session Key Exchange*).

Know Uses

The RSA Algorithm [5] uses the procedure described in this pattern to generate the public/private key pair.

Sample Code

```
// Primes Generation
public static BigInteger[] generatePrimes() {
    BigInteger PrimesTemp[] = new BigInteger[2];
    Random Rdn = new Random();

    // The constructor BigInteger(int, int, Random) returns a randomly
    // selected BigInteger with the specified bitLength that is probably prime.
    PrimesTemp[0] = new BigInteger(NumBitsKey,10,Rdn);
    Rdn.setSeed(Ln.longValue());
    PrimesTemp[1] = new BigInteger(NumBitsKey,10,Rdn);
    return PrimesTemp;
}

// Key Pair Generation
public static KeyPair generateKeys(BigInteger p, BigInteger q) {
    BigInteger BIntAux;

    //Represents the public/private key pair
    KeyPair KPairAux = new KeyPair();
    N = p.multiply(q);
    FN = (p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
    Random RdnAux = new Random();
    BIntAux = new BigInteger(NumBitsKey,RdnAux);

    // Find e
    while ((FN.gcd(BIntAux)).compareTo(BigInteger.ONE) != 0) {
        RdnAux = new Random();
        BIntAux = new BigInteger(NumBitsKey,RdnAux);
    }

    E = BIntAux;
    D = E.modInverse(FN);
    KPairAux.PrKey = new PrivateKey(D,N);
    KPairAux.PuKey = new PublicKey(E,N);
    return KPairAux;
}
```

4.2.3 Session Key Generation

Context

- You are developing a Client-Server application and you are applying the Client-Server Secure Communication Pattern.

Problem

- How do you generate a Session Key?

Forces

- efficiency - The generation of the session key must consume the best possible amount of time.
- security - The larger the session key is, the more secure the system is.

Solution

- Set the session key to a random generated number, with 128 bits or longer.

Rationale

By setting the session key to a random number, we have a fast method of generation. The size of the key is an important factor to be considered. A 128-bit key or longer provides a very good security.

Resulting Context

Now, both client and server have their own Session Keys. However, in order to start the secure communication, it is necessary that both client and server have the Session Key of each other (*Session Key Exchange*).

Know Uses

Most of the symmetric crytosystems, such as DES and IDEA [4], use random generated numbers as keys.

Sample Code

This Java method generates a random Session Key of Key_Size bytes.

```
static byte[] generateSessionKey (int Key_Size){
    byte[] Skey= new byte[Key_Size];
    Random Ran= new Random();
    Ran.nextBytes(Skey);
```

```
    return Skey;
}
```

4.2.4 Session Key Exchange

Context

- You are developing a Client-Server application. You are applying the Client-Server Secure Communication Pattern and both Client and Server Secure Sockets have its public/private key pair and the session key. Besides, both client and server know the public key of each other.

Problem

- How do you exchange the session keys?

Forces

- efficiency - Data encryption must be as fast as possible.
- security - The more secure the session keys are, the more secure the whole system is.

Solution

- Use an asymmetric approach to send the Server's Session Key to the Client and vice-versa.
- Let Sk be the sender's Session Key and $PuKey = \{e, n\}$ the receiver's Public Key. Compute the encrypted session key ESk as $ESk = Ck^e \ mod \ n$.
- Send this encrypted key to the receiver.
- To decrypt the session key compute $Sk = Esk^d \ mod \ n$. Where d and n are parts of the receiver's private key $\{d, n\}$.

Rationale

As asymmetric cryptography is very secure, its use in the session key exchange guarantees the security of the whole system. In addition, as the session keys are small, it makes this a quite fast approach.

Resulting Context

After this pattern both client and server know the Session Key of each other. At this moment, they can start the secure communication (*Data Encryption/Decryption*).

Know Uses

The RSA Algorithm [5] uses the procedure described in this pattern not to exchange keys in particular but to perform data encryption and decryption in general.

Sample Code

```
protected byte[] cryptSessionKey(byte[] stream, BigInteger e, BigInteger n) {
    BigInteger M = new BigInteger(stream);
    BigInteger C;
    C = M.modPow(e,n);
    return C.toByteArray();
}

protected byte[] decryptSessionKey(byte[] stream, BigInteger d, BigInteger n) {
    BigInteger M = new BigInteger(stream);
    BigInteger C;
    C = M.modPow(d,n);
    return C.toByteArray();
}
```

4.2.5 Data Encryption/Decryption

Context

- You are developing a Client-Server application. You are applying the Client-Server Secure Communication Pattern and both client and server have the session key of each other.

Problem

- How do you encrypt and decrypt data?

Forces

- efficiency - Data encryption must be as fast as possible.
- security - The information exchanged in the communication must be as secure as possible.

Solution

- Let T be the data to be encrypted and Sk the receiver's session key. Compute the encrypted data C as $C = T\ xor\ Sk$.
- In order to decrypt an encrypted data C, compute the original data T as $T = C\ xor\ Sk$. Where Sk is the receiver's session key.

Rationale

The use of the *xor* ($exclusive - or$) operator provide an efficient encryption method, mainly because of its speed. As the key used here was exchanged in a secure basis, the confidentiality of the data is guaranteed.

Other more used encryption techniques such as DES, IDEA, RC5, etc. could be used as solutions for the problem of encryption/decryption of data based on the exchanged keys. This could bring to the application an increased level of security really necessary in some situations where security is the first priority (such as financial applications, etc.). However, the use of such algorithms would increase the complexity of the system in such a way that the cost-benefit analysis would be unfavorable. By considering that, it seems that the use of the *xor* allows a very good level of security without increasing the complexity of the system.

Sample Code

This Java method crypts an array of bytes M by using a key K. M and K must have the same length.

```
static byte[] encrypt/decrypt(byte[] M, byte[] K) {
    byte[] D = new byte[M.length];
    for (i=0;i<M.length;i++){
        D[i] = (byte)(M[i]^K[i]);
    }
    return D;
}
```

5 Conclusions

The pattern language presented in this paper provides guidelines for secure client-server application developers. The five patterns presented here discuss all features that are necessary to build an efficient security environment. In addition, the composition of symmetric and asymmetric cryptosystems allows us to build systems in a reliable and fast basis.

The most important advantage in the application of this Pattern Language is in fact that developers do not need to have a deep knowledge about cryptography to create a very efficient encryption environment.

6 Acknowledgments

I would like to thank my classmates, my friends Rossana Castro and Igor Sales and my brother Cidcley Souza, who provided me with many valuable suggestions for improvement. Also, I would like to thank my shepherd Alexandre Braga for his relevant comments on this paper. Finally, we acknowledge CAPES for its financial support.

References

[1] Braga, A. M.,Rubira, C. M. F. and Dahab, R., Tropic: A Pattern Language for Cryptographic Software, *Proceedings of the Pattern Language of Program (PLoP 1998)*, USA, 1998.

[2] Buschmann, F., et al., *Pattern-Oriented Software Architecture*, John Wiley and Sons, New York, NY., 1996.

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

[4] Menezes, A.J., Orschot, P.C. and Vanstone, S.A. *Handbook of Applied Cryptography*, CRC Press, 1996.

[5] *RSA Security Inc.* www.rsa.com.

[6] Treese, G. W. and Stewart L. C., *Designing Systems for Internet Commerce*, Addison-Wesley, 1998.

Padrões de Análise para Aplicações de Gestão Urbana em Sistemas de Informação Geográfica[1]

Jugurta Lisboa Filho
Universidade Federal de Viçosa
Departamento de Informática,
36571-000, Viçosa/MG, Brasil
jugurta@dpi.ufv.br

Cirano Iochpe
Universidade Federal do Rio Grande do Sul
Instituto de Informática,
91501-970, Porto Alegre/RS, Brasil
ciochpe@inf.ufrgs.br

Karla A.V. Borges
PRODABEL
31230-000, Belo Horizonte/MG, Brasil
karla@pbh.gov.br

ABSTRACT

An analysis pattern is any part of a requirement analysis specification that can be reused in the design of other information systems as well. Urban management systems (e.g. Tax Control Systems, Urban Transportation System) are implemented in a similar way for many Municipalities. This paper proposes three analysis patterns that make possible the reuse of geographic database design for urban area planning and management applications, developed in Geographic Information System (GIS).

Keywords: Analysis Patterns, GIS, Conceptual Model, Reuse.

RESUMO

Um padrão de análise é qualquer parte de uma especificação de requisitos que se origina em um projeto e pode ser reutilizada em outros projetos de sistema de informação. Sistemas de gestão urbana (ex.: Sistemas de informação urbana, parcelamento do solo, Controle de Tributos Municipais, Sistema de Transporte, Sistema de saúde) possuem a característica de serem implementados de forma muito semelhante em diferentes municípios. O artigo propõe três padrões de análise que ilustram a possibilidade de reutilização de esquemas de banco de dados em aplicações da área de gestão urbana, desenvolvidos com o uso de Sistemas de Informação Geográfica (SIG).

Palavras-clave: Padrões de análise, SIG, Modelo conceitual, Reutilização.

---

[1] Trabalho financiado parcialmente pela FAPEMIG e pelo CNPq.

1 INTRODUÇÃO

O sucesso do desenvolvimento de grandes sistemas de informação tem como um de seus pontos chave a representação, de forma não ambígua, dos resultados da análise de requisitos e do projeto através do uso de formalismos bem conhecidos. Experiências têm demonstrado que as etapas de análise de requisitos e projeto conceitual do banco de dados são atividades complexas e que demandam muito tempo. Segundo Johannesson [13], uma das razões para isso é que o conhecimento do domínio da aplicação e o levantamento dos requisitos do sistema são feitos, quase sempre desde o início, para cada novo sistema sendo desenvolvido.

Aplicações baseadas em Sistemas de Informação Geográfica (SIG), embora apresentem alguns requisitos especiais (ex.: manipulação de dados referenciados geograficamente), devem ser desenvolvidas utilizando-se técnicas que são empregadas com sucesso no desenvolvimento de qualquer sistema de informação. Uma das técnicas que vem recebendo atenção especial, principalmente pela comunidade de projetistas de sistemas orientados a objetos, é o emprego de instrumentos que possibilitem a reutilização de componentes de software através da definição de padrões.

Um padrão é uma combinação recorrente de elementos de modelagem que ocorrem em algum contexto [6]. Padrões podem ser aplicados nas diversas etapas do desenvolvimento de software, recebendo, consequentemente, diferentes denominações como padrões de análise, padrões de projeto, padrões de arquitetura, idiomas (padrões de implementação), etc.

A reutilização de componentes de software vem sendo feita, embora informalmente, desde a implementação dos primeiros programas de computador. O uso do conceito de padrões de projeto na área da Ciência da Computação é, contudo, bem mais recente [10]. O emprego de padrões na área de desenvolvimento de software contribui para aumentar a reusabilidade e a qualidade de componentes de software. Padrões de projeto possibilitam a disseminação do conhecimento e a troca de experiências entre projetistas, bem como facilitam a comunicação entre diferentes membros de um projeto [20].

Padrões de análise, tem sido propostos como instrumentos para a reutilização de soluções durante as fases de análise de requisitos e modelagem conceitual do banco de dados [6], [9], [12], [13], [19] e [20].

Um padrão de análise é qualquer parte de uma especificação de requisitos que se origina em um projeto e pode ser reutilizada em diversos projetos [20]. Padrões de análise possibilitam a reutilização de soluções de análise em diferentes sistemas. Alguns padrões são menos genéricos e podem ser reutilizados em diferentes aplicações dentro de um mesmo domínio, enquanto outros padrões podem ser aplicados em diferentes domínios. Como exemplos de padrões de análise, específicos para um domínio, pode-se citar os padrões para aplicações na área de seguradoras [20] e os padrões para aplicações na área bancária [17]. Dentre os exemplos de padrões mais genéricos estão o padrão para reservas e locação de entidades reutilizáveis (ex.: reserva de quarto de hotel, reserva para aluguel de automóvel) [7], os padrões *Observações* e *Medidas* [9] e os padrões *Contratos* e *Documentos* [12].

Uma comparação dos diversos tipos de padrões existentes pode ser encontrada em [3]. Segundo Fernandez [8], dentre os motivos que diferenciam os padrões de análise dos padrões de projeto, pode-se citar:

- padrões de análise são dependentes da aplicação, pois sua semântica descreve aspectos específicos de algum domínio ou aplicação;

- padrões de projeto estão mais próximos da implementação por focar, principalmente, os aspectos típicos de projeto como, por exemplo, interfaces homem-máquina, criação de objetos, propriedades estruturais básicas;
- padrões de projeto podem ser aplicados a um número maior de aplicações. Por exemplo, a maioria das aplicações possui interface homem-máquina.

Uma particularidade das aplicações de SIG é que, normalmente, os dados manipulados por essas aplicações possuem um forte relacionamento entre si, devido a esses dados retratarem fenômenos geográficos que ocorrem sobre uma mesma região geográfica. Por exemplo, muitos dados espaciais (ex.: dados temáticos) são derivados ou utilizam dados básicos (ex.: dados topográficos) para serem representados. O conjunto de tipos de dados que compõe, normalmente, a base cartográfica para uma determinada aplicação de SIG possui uma estrutura conceitual muito parecida para a maioria das aplicações. Essa particularidade torna as aplicações de SIG fortes candidatas a se beneficiarem da reutilização de projetos de bancos de dados já existentes [14], como já vem ocorrendo com o compartilhamento de dados geo-espaciais em meio digital [21].

O artigo propõe três padrões de análise aplicáveis à etapa de análise de modelagem conceitual de banco de dados geográficos na área de gestão urbana. O restante do artigo está organizado como segue. A Seção 2 descreve a abordagem UML-GeoFrame, usada na modelagem conceitual de aplicações de SIG. A Seção 3 descreve os padrões de análise identificados em aplicações na área de gestão urbana. A Seção 4 descreve as conclusões finais e as perspectivas de trabalhos futuros.

2 MODELO CONCEITUAL PARA BANCO DE DADOS GEOGRÁFICOS

A solução apresentada por padrões de análise para projeto de banco de dados geográficos deve incluir um esquema conceitual de dados. Aplicações de SIG impõem uma série de requisitos de modelagem (ex.: fenômenos geográficos x objetos convencionais, visão de campo x visão de objetos, aspectos espaciais, múltiplas representações, aspectos temáticos, etc.), os quais são representados através de modelos conceituais próprios para estas aplicações [16].

Em [15], mostrou-se a adequação da abordagem UML-GeoFrame para especificação de padrões de análise para aplicações de SIG. Esta abordagem tem como base o modelo de classes da Linguagem UML-*Unified Modeling Language* [1], sendo que a modelagem dos requisitos da informação geográfica é feita através de estereótipos definidos no *framework* GeoFrame [15].

O GeoFrame é um *framework* conceitual que serve de base para a modelagem de aplicações de SIG, fornecendo um diagrama de classes a partir das quais as classes do domínio da aplicação são modeladas (especializadas). Para possibilitar a obtenção de esquemas de dados de fácil entendimento por parte de usuários leigos, o GeoFrame fornece um conjunto de estereótipos, ilustrado na Figura 1, cuja semântica é a de substituição de relacionamentos entre as classes da aplicação e as classes do GeoFrame.

Na Figura 1, o primeiro conjunto de estereótipos é usado para diferenciar os dois principais tipos de objetos pertencentes a um banco de dados geográficos. *Fenômeno geográfico* é especializado em *Objeto geográfico* [◬] e *Campo geográfico* [◭], segundo as duas formas de percepção dos fenômenos geográficos descritas por Goodchild [11]. Objetos não geográficos, ou convencionais, são modelados de forma tradicional, sendo identificados pelo estereótipo [△].

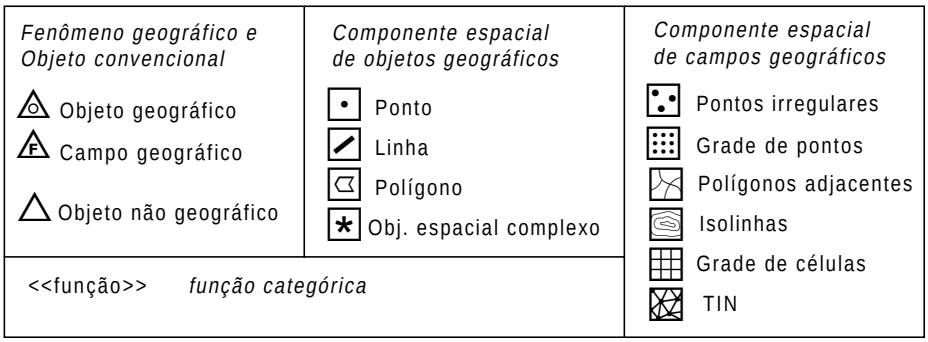


FIGURA 1 - Estereótipos do *framework* GeoFrame

O segundo e o terceiro conjunto de estereótipos são usados para a modelagem do componente espacial de fenômenos segundo as visões de objeto e de campo, respectivamente. A ocorrência de múltiplas representações é especificada combinando-se dois ou mais estereótipos. Por exemplo, uma classe *Município* pode ter duas formas de abstração de seu componente espacial, pontual e/ou poligonal, sendo especificado como [⊡◻].

Por último, o estereótipo <<função>> é usado para caracterizar um tipo especial de associação que ocorre quando da modelagem de funções categóricas. Segundo Chrisman [4], numa estrutura de cobertura categórica o espaço é classificado em categorias mutuamente exclusivas, ou seja, uma variável possui um valor do tipo categoria em todos os pontos dentro de uma região (ex.: tipos de solos).

A Figura 2 ilustra um extrato de esquema conceitual usando a notação UML-GeoFrame, onde percebe-se diversos temas (ex.: Ativ_Carvão, Uso_Solo), modelados como pacotes. Cada tema reúne classes coesas, que podem ser subclasses de objetos comvencionais [△] (ex.: EmpresaCarbonífera e TipoUsoSolo), de fenômenos geográficos percebidos na visão de objetos [◬] (ex.: Município, Jazida) e na visão de campo [◭] (ex.: UsoCobSolo, Topografia). Exemplos de múltiplas representações aparecem nas classes MinaCarvão [⊡◻], RecursoHidrico [／◻] e Topografia [⁘▦▩].

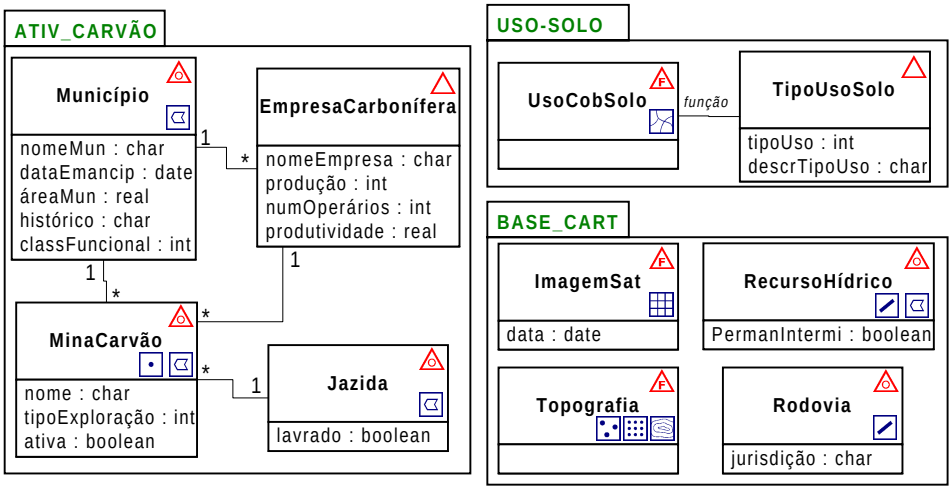


FIGURA 2 – Exemplo de esquema de dados UML-GeoFrame

3 PADRÕES DE ANÁLISE EM APLICAÇÕES URBANAS

Sistemas de geoprocessamento, mais especificamente os SIG, são usados em diversas áreas como Meio Ambiente, Telecomunicações, Negócios e Marketing, Monitoramento de Frotas, Administração Pública, entre outras. Em cada uma dessas áreas de aplicação é necessário criar um modelo de análise específico para o universo a ser trabalhado, de tal forma que os objetos observados possam estar relacionados com uma determinada região geográfica.

Na área de gestão urbana, a região de interesse corresponde a uma cidade, a qual é formada pelo ambiente natural e construído, possui traçado viário, construções, áreas livres, vegetação, clima, sua população, etc.

A cidade é um organismo vivo, mutante, dinâmico onde existem contrastes profundos que necessitam ser administrados em prol da qualidade de vida de sua população [2]. Sistemas tradicionais de representação, como os mapas, são estáticos mesmo que produzidos por meio de computador (sistemas de CAD), pois representam situações existentes no momento em que foram produzidos. Um SIG possibilita dinamizar os mapas, mantendo o registro da evolução da realidade, com base em dados coletados a partir de tarefas administrativas. Para tanto, a gestão necessita ver a cidade como um todo. Independentemente das diferentes visões e atuações sobre a cidade, ela é única e sensível à condição temporal [2].

A necessidade de gerenciar o município de forma integrada e a preocupação com a qualidade de vida urbana têm levado as prefeituras a se interessarem cada vez mais pelo uso de SIG [2]. No entanto, o primeiro desafio é obter recursos humanos com capacidade para projetar, implantar e manter os sistemas de gestão utilizando a tecnologia de SIG. A dificuldade é ainda maior quando o problema é transferido para prefeituras de porte médio ou pequeno.

Através da experiência adquirida pelos autores no desenvolvimento de aplicações de gestão urbana, a primeira característica que se observa é o grande potencial de reusabilidade das soluções adotadas, seja por diferentes órgãos de uma mesma administração, seja por diferentes prefeituras. Padrões de análise provêem um mecanismo altamente considerável na redução destas dificuldades, uma vez que: (1) possibilitam que um projetista menos experiente reutilize conhecimentos já testados e validados anteriormente; (2) na gestão urbana, o ambiente básico que compõe a base cartográfica digital (ex.: ruas, quadras, lotes e bairros) pode ser reutilizado por diversas aplicações.

A seguir, são apresentados três padrões de análise, identificados a partir da análise de esquemas conceituais de diversos bancos de dados em aplicações de gestão urbana. Por questões de simplificação são apresentados apenas os atributos e operações essenciais em cada exemplo.

Para a especificação dos padrões optou-se pela estrutura definida por Meszaros [18], na qual a descrição de um padrão deve conter, no mínimo, os seguintes itens: *Problema-Contexto-Forças-Solução*.

3.1 Padrão: Malha Viária Urbana

Problema:

Quais os elementos pertencentes à malha viária de uma cidade?

Contexto:

No Brasil, praticamente todas as cidades apresentam um mesmo padrão de organização, no qual são estruturadas com base em suas vias de locomoção (ex.: ruas, avenidas, travessas). O conjunto de trechos de vias e seus cruzamentos formam uma rede viária urbana.

Forças:

- Cada via de locomoção, considerada uma instância de logradouro, deve possuir um código de identificação e um nome, além de estar, normalmente, dividida em diversos trechos.
- Um trecho de logradouro corresponde ao segmento de via compreendido entre duas conexões, em seqüência, deste com outros logradouros que o cruzam ou interceptam.
- O conjunto formado pelas conexões (ou pontos terminais) e pelos trechos de logradouros constituem a malha viária urbana.

Solução:

A Figura 3 mostra o diagrama de classes pertencente ao padrão.

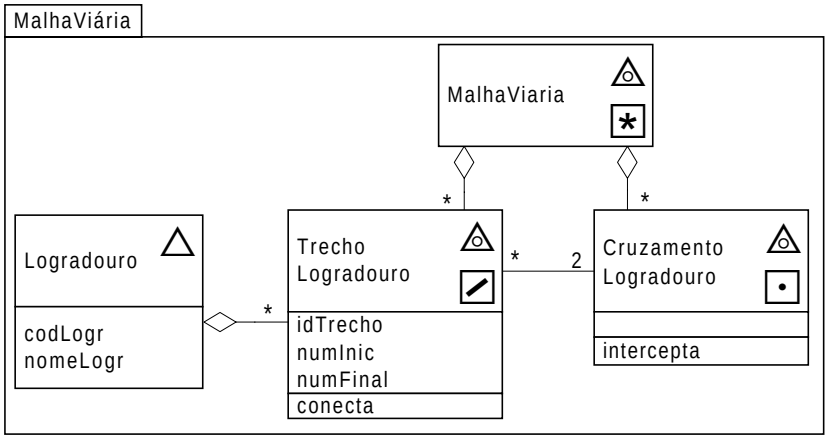


FIGURA 3 – Diagrama de classes do padrão “Malha Viária Urbana”

Para cada fenômeno geográfico, o padrão especifica apenas as propriedades (atributos e operações) mais genéricas, as quais devem ser estendidas e especializadas para cada aplicação específica. Consequentemente, são especificadas as possíveis abstrações de seus componentes espaciais. Por exemplo, o componente espacial da classe *TrechoLogradouro* é especificado como sendo linear [/].

Participantes:

A classe MalhaViária é um fenômeno geográfico representado por um objeto espacial complexo, o que é simbolizado por [*]. Nesta classe podem ser definidos atributos relativos à rede como um todo. Logradouro é uma classe convencional implementada, normalmente, como uma tabela em um SGBD relacional. Cada logradouro é composto de diversos trechos de logradouros, que correspondem às arestas da rede. Um trecho de logradouro pode estar conectado a outros trechos de logradouros, mas esta conexão é representada por instâncias da classe CruzamentoLogradouro, que são os nós da rede. As operações de manipulação dos elementos da rede podem ser implementadas como métodos das classes MalhaViária, TrechoLogradouro e CruzamentoLogradouro, dependendo de sua funcionalidade.

Padrões relacionados:

O padrão de análise *Malha Viária Urbana* utiliza o padrão "State Across a Collection" [5] na modelagem dos fenômenos Logradouro e Trecho de Logradouro. Além disso, pode-se abstrair um novo padrão de projeto que modele uma estrutura de uma rede qualquer, composta de arcos e nós, cuja topologia entre seus elementos seja mantida a fim de possibilitar a realização de operações comuns a estruturas de redes como cálculo do caminho ótimo (necessita de pesos para cada arco), navegação através da rede, distância entre dois nós, etc.

3.2 Padrão: Rede de Circulação Viária Urbana

Problema:

Como modelar os elementos de uma rede de circulação viária urbana?

Contexto:

A circulação de veículos em uma cidade é realizada sobre a malha viária urbana. A rede de circulação viária fornece o sentido do tráfego, enquanto a malha viária fornece a estrutura de vias. Algumas vias de locomoção possuem sentido único e outras sentido duplo. Cada trecho é classificado de acordo com sua importância para o sistema viário como, por exemplo, se é uma via coletora, de ligação regional ou uma via local (diversas aplicações fazem uso dessas informações).

Forças:

- Cada trecho de circulação, que pode compreender vários trechos de logradouro, possui informações sobre o sentido permitido para tráfego de veículos.
- Alguns trechos não permitem circulação de veículos (ex.: ruas de pedestres).

Solução:

A Figura 4 ilustra o diagrama de classes do padrão. Uma rede de circulação viária sobrepõe a rede da malha viária, desta forma, o padrão *Rede de Circulação Viária Urbana* estende o padrão *Malha Viária Urbana*, descrito na seção anterior.

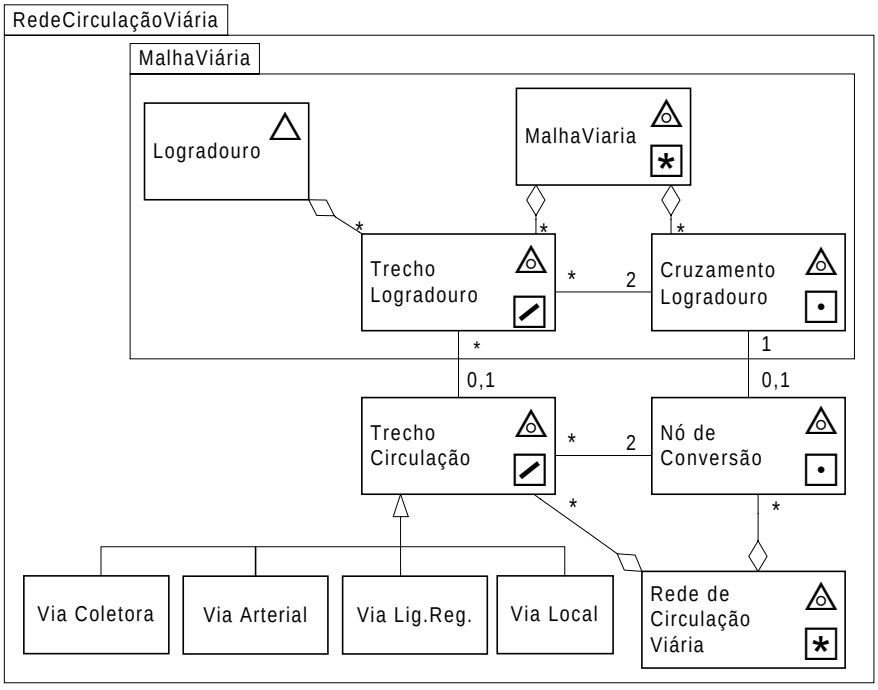


FIGURA 4 – Diagrama de classes do padrão "Rede de Circulação Viária Urbana"

Participantes:

Da mesma forma que no padrão *Malha Viária Urbana*, a classe RedeDe-CirculaçãoViária possui representação espacial complexa, formada pela representação de trechos e nós de conversão. Operações envolvendo toda a rede são definidas como métodos desta classe. Cada trecho de circulação pode estar associado a vários trechos de logradouros, significando a sobreposição de uma rede mais compacta sobre uma rede mais completa. Por outro lado, nem todo trecho de logradouro faz parte de um trecho de circulação, como ocorre com as vias de pedestre. Da mesma forma, há cruzamentos de logradouros que podem não ser, necessariamente, um nó de conversão. De acordo com a aplicação, um trecho de circulação pode ser especializado de diferentes formas. No contexto dos sistemas viários é comum aparecer a classificação dos trechos de circulação quanto ao tráfego e tamanho da via (ex.: via arterial, coletora, ligação regional e local).

Padrões relacionados:

O padrão *Rede de Circulação Viária Urbana* tem como base o padrão *Malha Viária Urbana*.

Exemplo:

Um sistema de itinerário de ônibus necessita da existência da circulação viária que, por sua vez, está sobre a malha viária, embora a malha viária também possa ser usada para outros fins. A Figura 5, extraída do esquema conceitual do banco de dados do sistema de transporte urbano da cidade de Belo Horizonte, ilustra o uso do padrão *Rede de Circulação Viária Urbana.*

Observa-se a reutilização de toda a modelagem referente aos temas *Malha Viária* e *Rede de Circulação Viária*. O projetista necessita modelar somente a parte do sistema relativa à sua aplicação, ou seja, o sistema de transporte de ônibus urbano. Outros exemplos de uso deste padrão de análise incluem os sistemas de roteamento de veículos para atendimento de emergência (ex.: ambulâncias, policiamento, corpo de bombeiros) e roteamento para entrega de mercadorias.

3.3 Padrão: Loteamento Urbano

Problema:

Como estruturar os dados de uma base cadastral urbana?

Contexto:

O ponto de partida para qualquer aplicação de SIG, que tenha uma cidade como área geográfica de interesse, é a elaboração da base cartográfica digital, integrada a um cadastro multifinalitário. Estas duas bases de dados são utilizadas por aplicações diversas como atendimentos de urgência (ex.: ambulância, bombeiros, segurança pública), controle de matrícula em escolas públicas, distribuição de postos de saúde, arrecadação de tributos, redes de infra-estrutura (ex.: água, esgoto, luz, telefonia), etc. Estas aplicações necessitam de informações como traçado viário, localização de bairros, quadras, lotes e, em alguns casos, até mesmo informações precisas sobre os limites das construções dentro de cada lote.

Forças:

- O nível de detalhe da base cadastral depende da existência de dados digitais espaciais para a cidade sendo modelada. Nem sempre é viável financeiramente obter a base cadastral na escala pretendida. Quanto maior a escala original, maior os custos de obtenção e maior os problemas de manutenção dos dados.

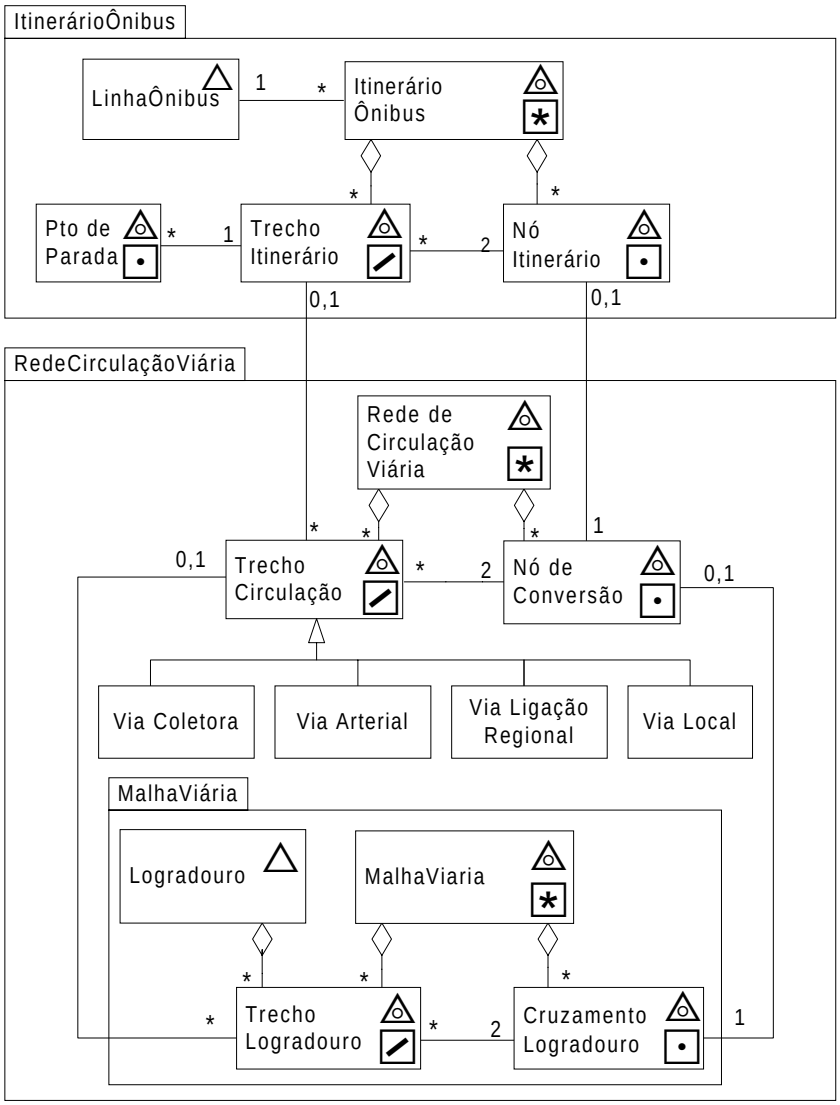


FIGURA 5 – Exemplo de uso dos padrões “Malha Viária Urbana” e “Rede de Circulação Viária Urbana”

- Dependendo do porte do município, diferentes tipos de divisões são empregados. Os tipos mais comuns incluem divisões administrativas e bairros.
- O conceito de bairro não é único para todas as cidades. Por exemplo, uma quadra pode não pertencer, necessariamente, a um único bairro. Em algumas cidades os limites de um bairro podem cortar até mesmo um lote.
- Um lote deve possuir dois tipos de representação espacial: a representação de seus limites e a representação correspondente à frente do lote, também conhecida por “testada do lote”. O mesmo pode ocorrer com as quadras na representação de “faces de quadra”.

Solução:

A Figura 6 mostra o diagrama de classes que compõe o padrão.

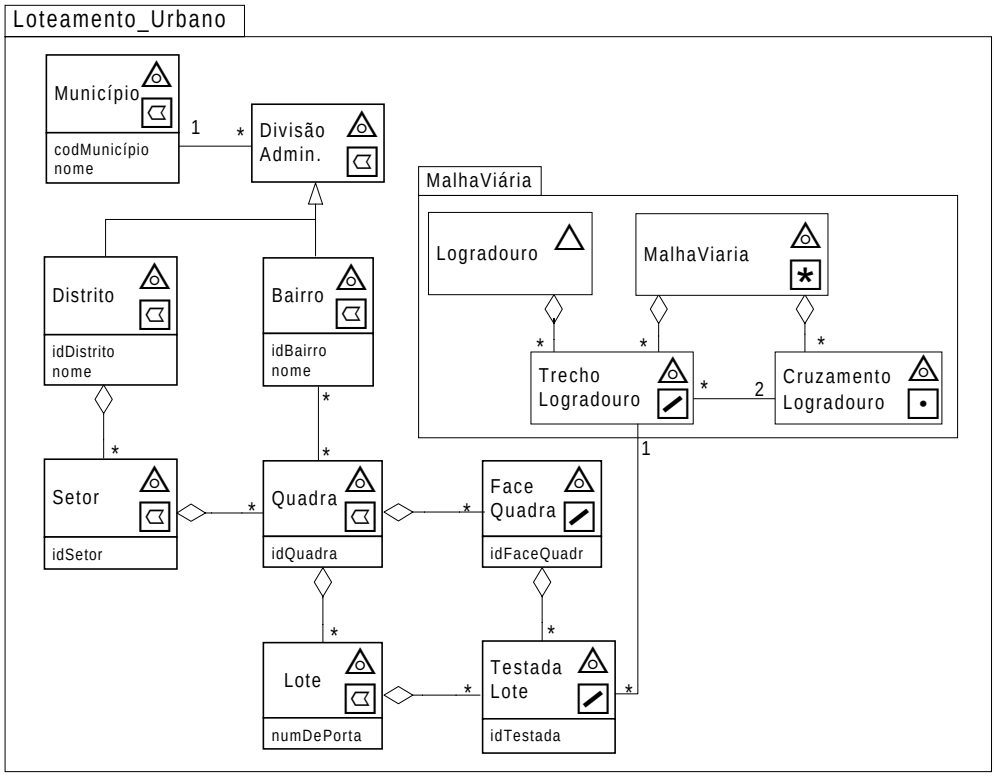


FIGURA 6 – Diagrama de classes do padrão “Loteamento Urbano”

Participantes:

A classe DivisãoAdministrativa pode ser especializada em outras subdivisões municipais (ex.: setores censitários, zonas de coleta de lixo, zonas de policiamento). A cidade, ou sede municipal, corresponde a um distrito.

A classe Bairro está associada à classe Quadra, através de uma multiplicidade “um-para-muitos”, mas esta associação deve ser adaptada a cada situação específica. Em alguns municípios o limite de um bairro pode não respeitar os limites das quadras, neste caso, a multiplicidade seria “muitos-para-muitos”, o que implica em uma situação não desejada.

Outra variação que pode ocorrer diz respeito à forma de associar o lote ou a quadra com o trecho de logradouro. Na solução apresentada, por exemplo, o trecho de logradouro está associado à testada do lote. No entanto, o lote poderia estar associado diretamente ao trecho de logradouro. Em situações nas quais o maior nível de detalhe são as quadras, o trecho de logradouro estaria associado à face de quadra.

Padrões relacionados:

Malha Viária Urbana.

Exemplo:

O uso do padrão *Loteamento Urbano* pode ser visto na Figura 7, a qual ilustra um sistema de cadastro urbano para fins de tributação do Imposto Predial e Territorial Urbano (IPTU).

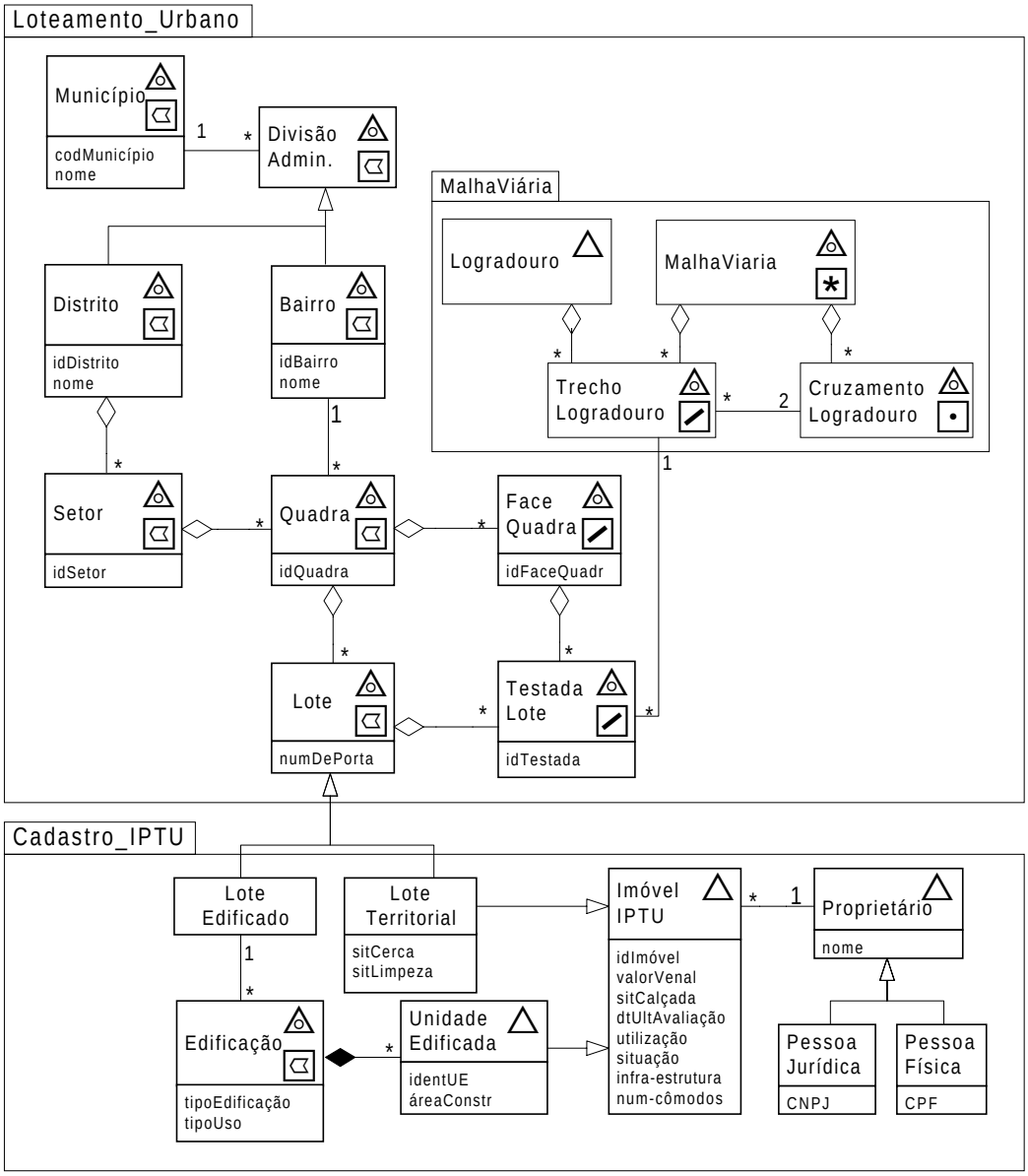


FIGURA 7 – Exemplos de uso do padrão “Loteamento Urbano”

No exemplo, um lote edificado pode possuir diversas edificações (ex.: um condomínio com vários edifícios, um Shopping Center). Cada edificação pode ser composta de diversas unidades edificadas (ex.: apartamentos, lojas). Tanto as unidades edificadas como os lotes territoriais (não edificados) constituem unidades para fins de tributação, modelado pela classe Imóvel IPTU, o qual pode estar associado, normalmente, a um proprietário principal. Foram incluídos apenas os atributos mais comuns, uma vez que a necessidade de atributos depende dos objetivos do sistema. O padrão “Party” [9] é empregado na modelagem dos diferentes tipos de lotes e nos diferentes tipos de proprietários.

O exemplo também serve para ilustrar a situação em que, embora não seja possível para uma prefeitura reutilizar dados georreferenciados de outro município, com exceção de dados regionais, é muito provável que o projeto do banco de dados desenvolvido por uma prefeitura possa ser reutilizado em grande parte por outras prefeituras. Isso ocorre devido à semelhança de legislação entre os municípios brasileiros.

4 CONCLUSÕES

Como se pode observar, a partir dos padrões de análise apresentados neste artigo, padrões de análise não são soluções completas. Padrões descrevem orientações, projetos iniciais e propostas de solução para problemas recorrentes. Padrões de análise necessitam ser adaptados em cada caso específico de reutilização.

A abordagem de padrões de análise apresenta grande potencial para melhorar a qualidade das aplicações de gestão municipal usando SIG, bem como para reduzir o tempo e, consequentemente, os custos das etapas de análise de requisitos e modelagem conceitual do banco de dados.

Entretanto, para o sucesso desta abordagem é necessário criar a cultura da cooperação entre os desenvolvedores de sistema. Por exemplo, muitos usuários reutilizam dados georreferenciados obtidos de terceiros, mas não disponibilizam seus próprios dados [21]. Reutilizar uma boa solução documentada por outro projetista é uma idéia muito atraente, mas é necessário que todos contribuam com a abordagem de reutilização, documentando suas soluções, por exemplo, na forma de padrões de análise.

Um padrão de análise não necessita apresentar uma solução original. Pelo contrário, padrões devem documentar soluções já testadas e validadas, pois são soluções para problemas recorrentes. Soluções para problemas únicos não necessitam ser documentadas na forma de padrões, pois provavelmente não necessitarão ser reutilizadas. Assim, a evolução da documentação de um padrão, a partir da contribuição de outros projetistas que o tenham reutilizado, deveria ser decorrência natural de seu uso. Os padrões de análise descritos neste artigo podem e devem ser melhorados. Contribuições, comentários e críticas são bem vindos.

Como continuidade deste trabalho, está prevista a investigação de alternativas para disponibilização dos padrões de análise existentes (para banco de dados geográficos) e, também, o estudo de aplicações de SIG em diferentes domínios (ex.: redes de infra-estrutura, aplicações ambientais), com o objetivo de captura e documentação de novos padrões. O desenvolvimento de ferramentas para suporte à busca por padrões existentes tem sido pesquisado [22]. Encontra-se em desenvolvimento no Departamento de Informática da UFV, um projeto de pesquisa que visa a implementação de uma ferramenta CASE para suporte à modelagem conceitual de banco de dados geográfico, com base no modelo UML-GeoFrame, a qual também dará suporte a reutilização de esquemas de banco de dados por meio de padrões de análise.

Agradecimentos

Os autores agradecem as contribuições feitas pela nossa "*shepherd*" Rosana Teresinha Vaccare Braga, as quais possibilitaram melhorar muito os padrões de análise aqui descritos.

REFERÊNCIAS

1. Booch, G.; Jacobson, I.; Rumbaugh, J. The Unified Modeling Language User Guide. Reading: Addison-Wesley, 1998.
2. Borges, K. A. V. A gestão urbana e as tecnologias de informação e comunicação. *Revista IP-Informática Pública,* Belo Horizonte, Vol.2, No.2, 2000.
3. Buschmann, F. et al. *Pattern-Oriented Software Architecture:* a system of patterns. New York: John Wiley & Sons, 1996.
4. Chrisman, N. *Exploring Geographic Information Systems*. NY: J. Wiley & Sons, 1997.

5. Coad, P. *Object Models:* Strategies, Patterns, and Applications. 2.ed. New Jersey: Yourdon Press, 1997.
6. Fernandez, E. B. Building systems using analysis patterns. *Procs. of Int. Software Architecture Workshop (ISAW3),* 1998.
7. Fernandez, E. B.; Yuan, X. An analysis pattern for reservation and use of reusable entities. *Procs. of Workshop in the Conference of Pattern Language of Programs - Plop*, 1999. Available at http://st-www.cs.uiuc.edu/~plop/plop99/ proceedings/.
8. Fernandez, E. B.; Yuan, X. Semantic Analysis Patterns. In: A. H. F. Laender, S. W. Liddle, V. C. Storey (eds): *Procs. of ER2000 Conference,* LNCS 1920, 2000. Springer-Verlag Berlin Heidelberg, 2000.
9. Fowler, M. *Analysis Patterns:* reusable object models. Menlo Park, CA: Addison Wesley Longman, 1997.
10. Gamma, E. et al. *Design Patterns:* elements of reusable object-oriented software. Reading, MA: Addison Wesley, 1994.
11. Goodchild, M. F., Geographical data modeling. *Computers & Geosciences*, Vol 18, No 4, 1992, pp.401-408.
12. Hay, D. C. *Data Model Patterns:* conventions of thought. New York: Dorset House Publishing, 1995.
13. Johannesson, P.; Wohed, P. The deontic pattern – a framework for domain analysis in information systems design. *Data & Knowledge Engineering*, Vol.31, 1999.
14. Lisboa Filho, J.; Iochpe, C.; Beard, K. Applying Analysis Patterns in the GIS Domain. *Procs. of Annual Colloquium of the Spatial Information Research Centre - SIRC*., Dunedin, NZ, 1998.
15. Lisboa Filho, J.; Iochpe, C. Specifying analysis patterns for geographic databases on the basis of a conceptual framework. *Procs. of ACM Symposium on Advances in Geographic Information Systems*, Kansas City, USA, 1999.
16. Lisboa Filho, J.; Iochpe, C. Um estudo sobre modelos conceituais de dados para projeto de bancos de dados geográficos. *Revista IP-Informática Pública,* Belo Horizonte, Vol.1, No.2, 1999a.
17. Marsura, P. *Banking patterns home page.* Available at http://www.joeyoder.com/ marsura/banking/ (03/09/1999).
18. Meszaros, G.; Doble, J. *A pattern language for pattern writing.* Available at http://hillside.net/patterns/Writing/ pattern_index.html (01/12/98).
19. Rawsthorne, D. A. A patterns language for requirements analysis. *Procs. of Workshop in the Conference of Pattern Language of Programs - PLoP*, Monticello-Illinois, 1996.
20. Robertson, S.; Strunch, K. Reusing the products of analysis. Procs. of Int. Workshop on Software Reusability, Lucca, Italy, 1993. Available at http://www.atlsysguild.com/ GuildSite/SQR/reusingAnalysis.html.
21. Weber, E. J.; Lisboa Filho., J.; Iochpe, C.; Hasenack, H. Geospatial metadata in Brazil: an experience in data documentation of an environmental GIS application. *Procs. of Int. Conference. Exhibition on Geographic Information (GISPlanet),* Lisbon, Portugal, 1998.
22. Wohed, P. Tool support for reuse of analysis patterns – a case study. In: A. H. F. Laender, S. W. Liddle, V. C. Storey (eds): *ER2000 Conference,* LNCS 1920, 2000. Springer-Verlag Berlin Heidelberg, 2000.

A PATTERN LANGUAGE FOR FAILURE DETECTION AND RECONFIGURATION OF DISTRIBUTED SERVICES*

Marcelo B. d'Amorim Carlos A. G. Ferraz

Universidade Federal de Pernambuco
Centro de Informática
Caixa Postal 7851, 50640-970, Recife-PE, Brazil
{mbd,cagf}@cin.ufpe.br

Abstract

The trading mechanism enables unprecedented opportunity for dynamic software configuration, which is achieved by resolving dependencies among distributed components. As embedded systems and handheld devices such as PDAs (Personal Digital Assistants) and cellular telephones are becoming very popular, distributed adaptability deserves special the attention of designers. In this context, dealing with failure is a critical issue designers need to tackle in order to preserve reliability. A component, for example, may stop running until it rediscovers a new service instance that implements the same interface as the failed one. This situation may be an essential requirement as long as the client is unable to perform its tasks without a reference to that service, and it is very often referred to as *dependence management*.

1 Introduction

Configuration is frequently associated with evolutionary change of criticial systems with long life duration. It attempts to assure reliability and predictability of components usually under a distributed environment. These systems typically need to evolve along with human needs. Technology and even the application environment change [15], and these changes may range from existing functionalities, network partitioning, to host and service failure.

The main benefits of software configuration are detailed in the following:

- **Incremental Evolution** - By considering reconfiguration as an issue at design time applications obtain higher flexibility to adapt to changes during the long run or even during its development. Critical systems, for example, may require run-time reconfiguration without stopping the entire system. Software prototyping may benefit from the flexibility of easily changing different application components [20]. Actually, different software architectures could be tested with reduced effort.

*This paper was workshoped on the First Latin American Conference on Pattern Languages of Programming - SugarloafPLoP. October 3-5, 2001 - Rio de Janeiro, Brazil.

- **Fault tolerance** - Damage caused by a fault in a distributed application can be minimized by reconfiguring the elements comprising an application. Elements running on the failed node can be re-deployed on new nodes and their cooperating elements instructed on where to redirect their cooperating requests [20].

In order to enable reactive networks and component self-healing, the *trading* mechanism [2] provides a means to dynamically search for components based on its properties and receive event notifications when these components are not available, however, trading systems like Jini [7] and CORBA Trader [23] do not provide built-in support to automatic distributed configuration of systems. In the presence of failures, components have to deal with this issue programmatically in order to be recovered to a consistent state.

In addition, very often clients are designed to use the trading semantics to search for services just at startup - configuration time. In this case, components will crash whenever remote references become invalid, even though alternative implementations are available on the network.

The **Virtual Synchronism** model attempts to keep in each *member* of a multicast group the consistent *view* of the group. This is achieved by the implementation of a *membership* protocol. Using election protocols, guarantees of message ordering, service replication, and state transfering this model provides an elegant approach to tame fault tolerance [1]. This approach deals with fault-tolerance at multicast group level thus relieving clients from the task of reconfiguring their bindings. However, the middleware support, like those provided by the Isis and Horus platforms [12, 27], must be reachable by every node on the system. We believe that in large scale component networks, like JTrader [2] and Ninja [24], it would be difficult to provide virtual synchronism. In this case, client nodes would have to keep valid its bindings.

We argue for the need of distributed components to support reconfiguration in a dynamic environment where new services become available while others fail. Not dealing with this issue properly may introduce non-determinism since we cannot assure that remote operations run correctly in the presence of invalid remote references. Actually, it is very difficult to guarantee full client reliability. Considering that a step before a client calls a remote operation the service fails, that client will indeed fail. In such situation, the step time was not enough to reconfigure the system with a new running service. In critical systems, however, this scenario is unacceptable, so the time frame to system reconfiguration is likely to be smaller.

2 Pattern Language Overview

The pattern language is a set of related patterns. They deal with aspects related to reconfiguration and failure detection. The Component Searcher is a high-level pattern in the sense that it makes use of the other patterns in the language. It represents a repository of valid service references which a client uses when needs to contact a service.

In order to promote dynamic configuration, the framework must verify service availability. When verification fails, it must trigger an event that causes the system to search for a new service instance. There are different approaches to accomplish this task. The pattern language suggests two alternative approaches for failure detection. The first one, named Heart-beat, is intended for systems with a dedicate bandwith and hard-availability constraints. The other is intended for systems with soft constraints, typically

asynchrounous distributed information systems. This one, named the Reverse Lease Subscriber, relieves clients from frequently testing for the validity of service references.

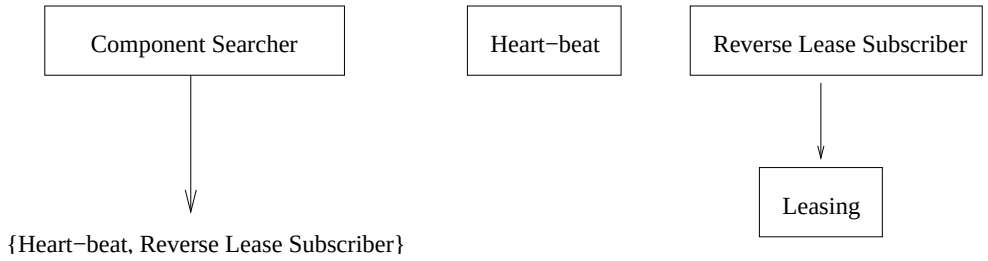


Figure 1: The pattern language

The diagram of figure 1 associates these patterns. The **Component Searcher** depends on a failure detector, which has two alternative policies: the **Heart-Beat** and the **Reverse Lease Subscriber**. Note that RLS uses the **Leasing** pattern. These patterns are described in the following sections.

2.1 The Component Searcher

Overview

Contrasting with the design of fault-tolerant distributed services [18, 16], this pattern does not comply with starting replicas of services on-the-fly in order to *assure* there will be a service running in the moment of a call. It monitors service availability and informs the underlying infrastructure that a new service is required to be bound. It attempts to detect faults [22, 19, 11] and reconfigure dependencies when they occur but it does not assure the client will be serviced. We regard it as *best-effort*. Even though failures due to service unavailability may not be completely removed with this approach, designing distributed systems with this issue in mind reduces failure likelihood and eases system administration.

Different policies to detect failure may be applied by implementing specialised service monitors. Regardless the policy applied, the presented behavior is preserved. These specializations will be represented as independant patterns: the Heart-Beat and the Reverse Lease Pattern (RLS).

Context

Components depending on distributed services that may fail.

Problem

The storage of remote references on the client code causes unpredictable behavior in the presence of failures in the communication path between the client and service.

Forces

Unless the interaction of program modules with the pattern infrastructure is carefully delineated, it becomes difficult to reuse the solution. We must then resolve the following

forces:

1. *Availability of dependent services* - In order to achieve service reliability, we must make every effort to continously provide the required service bindings. A good solution would then **increase** this force.

2. *Management of service references* - Considering failures due to network patitioning and service failure can occur, clients need to decouple the storage of service and name server references of its representation.

Solution

The infrastructure presents a collection of classes aimed at configuring client-server dependencies. A component, named `Searcher` mediates the current service configuration. It is to be called by the `AbstractMonitor` whenever a change on the configuration state is perceived[1]. Neither references nor names to services are stored within the `Searcher`, but only service templates that declare general capabilities that an eligible running services must provide. The properties of a service template are conserved during the long-term execution of an application and are so more reliable than a service binding.

Whenever a service binding failure occurs, the `AbstractMonitor` calls back the `Searcher` via its `IUnavailableService` interface in order to locate a replacement.

Instead of storing persistent references to services, clients must look for a reference on the `Searcher` when they need to contact a service. This, however, does not assure a client has a valid binding. If a service binding some client uses fails, it needs to request another to the `Searcher`, which returns a valid reference, if one exists. Otherwise, it throws an exception.

Static Structure

As presented in Figure 2, four components performing distinct and well defined roles are responsible to provide automatic configuration. Clients interact only with the `Searcher` component which is in charge to mediate interactions between the `DiscoveryProxy`, the `AbstractMonitor`, and the `ServiceCollection`. In the following these components are detailed:

- `ILookup` - This interface enables the `DiscoveryProxy` to implement a two-phase discovery protocol. Requests for services are enqueued on the `DiscoveryProxy`.
- `IDiscovery` - This interface is used to configure the locations where services are hosted.
- `DiscoveryProxy` - The discovery proxy isolates from clients the concept of a name server to provide location transparency and extend the solution to other distribution platforms that provide different discovery protocols. For example, when a Jini lookup service joins on a group or is destroyed, the effects of these events are confined in this component.

[1] Actually, this pattern only deals with failure detection on component connections. A similar strategy could be applied to cope with runtime changes of non-functional requirements, like end-to-end quality of service degradation.

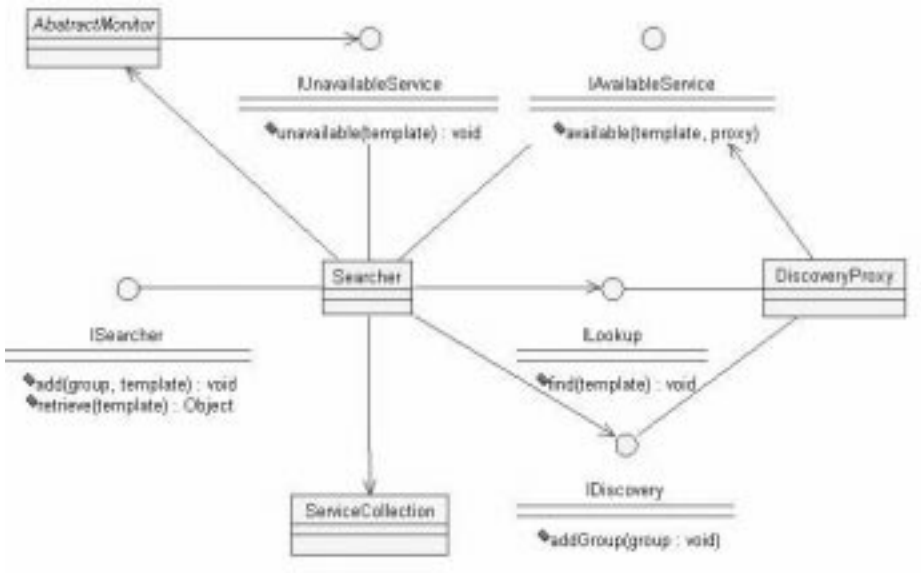


Figure 2: Pattern structural dependencies

It behaves like a federation where all required name servers are reachable[2]. As a result, clients do not interact directly with name server instances.

- `AbstractMonitor` - This class provides a means to service availability awareness. A concrete implementation of this class may refine its behavior, for example, by introducing active verification of remote references in a different thread of execution.
- `ISearcher` - The interface used by clients to configure the searcher and request transient service references.
- `IUnavailableService` - This is a listener interface used by the monitor to notify the searcher of a service failure.
- `IAvailableService` - This is a listener interface used by the `DiscoveryProxy` to notify the searcher of an available service.
- `Searcher` - The searcher stores in a `ServiceCollection` object service templates (a declarative description of a service) and the location (a semantic category of a name service) where these services should be found. This information is updated by the client via the `ISearcher` interface. The service collection update is also triggered by a listener event.
- `ServiceCollection` - An object of this type stores mappings between location objects to service templates, which in turn maps to service proxies.

When a client needs to call a method on a service, it requests the `Searcher` for a reference to that service. The `Searcher` in turn calls the `ServiceCollection` object for the proxy that the template supplied maps to. If that service is not available, the searcher then requests the `DiscoveryProxy`, through its `ILookup` interface, to find a service on the network with the informed template.

[2]In Jini, a name server joins a group in accordance to the kinds of services it contains. For example, a name server that joins on a "devices" group is supposed to contain proxies to devices.

Dynamics

A complete description of this pattern dynamics is presented on the RLS pattern.

Consequences

In the following, we present how the forces mentioned above are addressed by this pattern:

- *Force 1* - The `Searcher` is charged of searching for alternative service bindings whenever a fault occurs. A client requests the `Searcher` to retrieve a valid reference for the service template provided.
- *Force 2* - The framework behaves as a repository of service and name server references. The communication between the framework and clients is done via semantic categories and service properties.

Implementation

The client needs to create a `Searcher` object and inform what kind of policy it wants to apply to detect faults, what services it will need in a near future, and where they are supposed to be located. In the following, we show the initialization process of a client. The code is implemented in Java and used some Jini classes:

```
AbstractMonitor monitor = ...; // a concrete monitor
Searcher searcher = new Searcher(monitor);
String group = "bank_systems";
ServiceID id = null;
Class[] types = {Accounts.class};
Entry[] entries = {};
ServiceTemplate template = new ServiceTemplate(id, types, entries);
searcher = searcher.add(group, template);
```

In a declarative manner, we specified a service by the properties it has, such as its interface. The `ServiceTemplate` class of the Jini API represents this feature. We also informed the `Searcher` to locate the group where a reference to this service is supposed to be stored.

As we will see, the initialization process is not synchronous. That is, references to services are not available instantaneously after initialization. At any moment, however, the client can execute an operation that calls a service. In this case, the client needs to retrieve from the searcher the intended service. If the service is available the client is able to proceed, otherwise, it should throw an exception or repeat the operation after a short-time:

```
public void some_operation() ... {

   Accounts service = (Accounts) searcher.retrieve(template);
   service.operation();  ... }
```

The `Searcher add` operation requests the `DiscoveryProxy` to search for name servers that store references to `bank_systems`. Prior to locating these name servers, it enqueues a request to the `DiscoveryProxy` to find the intended service. Therefore, this operation does not block waiting for the service. When the `DiscoveryProxy` finally finds the requested service, it calls the `Searcher` via its `IAvailableService` inteface. The `Searcher` then

updates the `ServiceCollection` so that the `retrieve` operation will work properly when called.

```
public class Searcher ... {
 ServiceCollection repository;
 IDiscovery disco;
 ILookup lookup;
 ...
 Searcher(AbstractMonitor monitor) {
   this.monitor = monitor;
   repository = new ServiceCollection();
   DiscoveryProxy discoverer = new DiscoveryProxy((IAvailableService) this);
   disco = (IDiscovery) discoverer;
   lookup = (ILookup) discoverer;
 }

 public synchronized void add(String group, ServiceTemplate template) {
    repository.add(group, template);
    disco.addGroups(group);
    lookup.find(template);
 }
}
```

2.2 The Heart-Beat (Patlet)

Intent

To actively verify the availability of resources on the network.

Problem

Services eventually fail, hosts eventually crash, networks eventually partition. Reliable distributed systems must deal with these kinds of problems.

Context

Distributed systems whose resources may fail need to adapt to changes.

Forces

- *Coordination* - A client needs to actively (by its own) detect if a server has failed in order to rearrange its bindings before any operation be performed.
- *Overhead* - In order to achieve an acceptable overall performance, the solution should **minimize** overhead.

Solution

In order to deal with a service failure in advance of an operation request, the application creates a concurrent thread of execution to frequently test if a service and the network are reachable.

Example

Even though RMI [26] does not provide such support on its public API, programmers can simulate a method call by using the `invoke`() method of the `RemoteRef` instance, which is retrieved from the stub's `getRef`() method. If a `ConnectException` is thrown in this call, the test is regarded as failed. Alternatively, it is still possible to construct RMI "ping" packets, as described on JDK1.3 RMI documentation [25], to verify if the RMI server is still alive.

2.3 The Reverse Lease Subscriber (RLS)

Overview

The Reverse Lease Subscriber proposes a means to detect failures on distributed services that a component depends on by requesting these services to renew leases [13] on it. While leases are properly renewed, a component is aware of dependant services availability, otherwise, components need to arrange for a replacement of its bindings and must suspend requests in this between.

Intent

Verify the availability of resources on the network.

Problem

Services eventually fail, host eventually crash, networks eventually partition. Reliable distributed systems must deal with these kinds of problems.

Context

Distributed systems whose resources may fail need to adapt to changes.

Forces

We must resolve the following forces:

1. *Reactivity* - A component needs to detect a failure of any kind in the path between a client and the service.

2. *Peformance* - The solution should achieve an acceptable overall performance.

Solution

This pattern defines a failure detection policy, which uses the concept of resource leasing (see the next patlet). To implement the policy we need a service monitor capable of implementing the leasing strategy. The `ConcreteLeaseMonitor` represents this monitor. It extends the `AbstractMonitor` class, presented on the Component Searcher pattern.

The monitor asks to the service, by means of the `IReverseLeaseSubscriber` interface, to renew a lease it supplies. The service then starts to renew the lease through the `Landlord` object which the lease is able to contact. This object is hidden within the `ConcreteLeaseMonitor` object and it is supposed to be transparent to both the service and the client programmer. When the lease becomes expired, the `ConcreteLeaseMonitor` issues a local event directly to the searcher through the `IUnavailableService` interface in order to locate a replacement.

Even though most lease grantors are server-side components, this is not a requirement at all. In fact, in RLS the service component performs the holder's role and the client, the lease grantor's. This lease pattern of use is here called *reverse lease subscriber* because, in contrast to regular leasing, in RLS the lease grantor starts the lease protocol. By requesting services to renew its leases on the client, such client becomes aware of eventual failures as long as leases become expired.

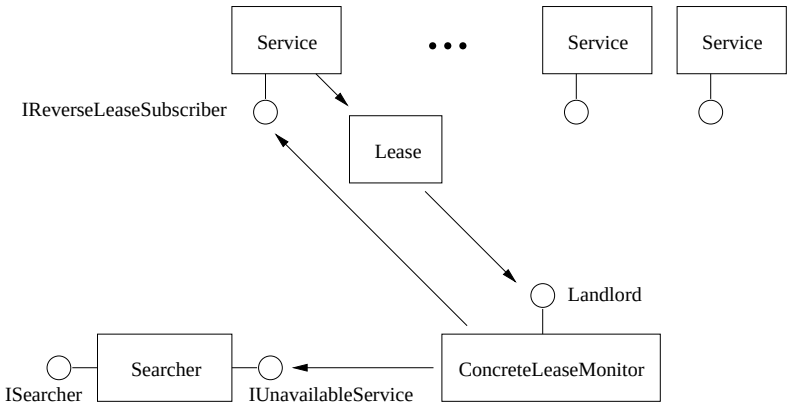


Figure 3: Reverse Lease Subscriber

In the following, we summarize the Reverse Lease Subscriber pattern solution by delineating the tasks it performs:

1. A client locates a service via the searcher.
2. The monitor tells the registered services to maintain a lease with it. The lease monitor is responsible for ensuring a service maintains a lease, indicating that it is available, or else it tells the searcher that the service is no longer available.
3. If a service becomes unavailable, the searcher attempts to locate a replacement.
4. When a client detects a service has failed - this will probably cause a kind of remote exception to be thrown and catched - it requests the searcher for a new reference.

Static Structure

Now we describe the structure of the RLS pattern by describing its core components.

- `Lease` - This object represents a contract between two entities. If the lease expires the contract is no longer valid. The lease holder needs to request the grantor for renewall in order to avoid the lease to become expired.
- `IReverseLeaseSubscriber` - This is the interface which the `ConcreteLeaseMonitor` contacts in order to request a service to start the RLS protocol.
- `Landlord` - The interface contacted to renew a lease. An object of this type should also clean up expired leases in order to maintain a consistent view of resource in use.
- `IUnavailableService` - This is the interface the `ConcreteLeaseMonitor` contacts to inform the `Searcher` that a service previously bound is no longer valid.
- `ConcreteLeaseMonitor` - This type implements a passive and lightweight[3] policy to handle service availability by extending the `AbstractMonitor` class. This component is a lease grantor and is also responsible to start the lease protocol. When requested for testing availability of a given service, it contacts the `IReverseLeaseSubscriber` interface the service is supposed to expose and then invokes its `pleaseRenew`() method, passing a `Lease` object as parameter. Through the `Lease` object, the service should call back the `Landlord` interface the `ConcreteLeaseMonitor` exposes before the lease timeout runs. The `ConcreteLeaseMonitor` actively verifies expired leases it stores. When an expired lease is perceived, this component calls the `IUnavailableService` interface that the `Searcher` exposes, informing which template requires reconfiguration.

Dynamics

Figure 4 presents a practical scenario where a client configure the `Searcher` component to search for service instances and keep valid their references. This scenario uses RLS pattern to implement the monitor and introduces a failure event that avoids the service to renew the lease.

After acquiring a service proxy, the searcher requests the service implementation to lease the proxy at the client. This is accomplished by calling the `pleaseRenew`() method on the `IReverseLeaseSubscriber` interface. As long as leases become expired due to a failure in a service, for example, new service references need to be located and old ones discarded.

Executing under a different thread of control, the `ConcreteLeaseMonitor` frequently scans for expired leases. When some is found, a message is sent to the searcher in order to update the set of available services.

Consequences

The major liability this pattern presents is related to its scalability:

[3]We regard this approach as lightweight because clients are not required to open a connection to the server in order to notice that everything is all right; such responsibility is charged to the service, and so the approach is also passive as the client is relieved from frequently testing the bindings.

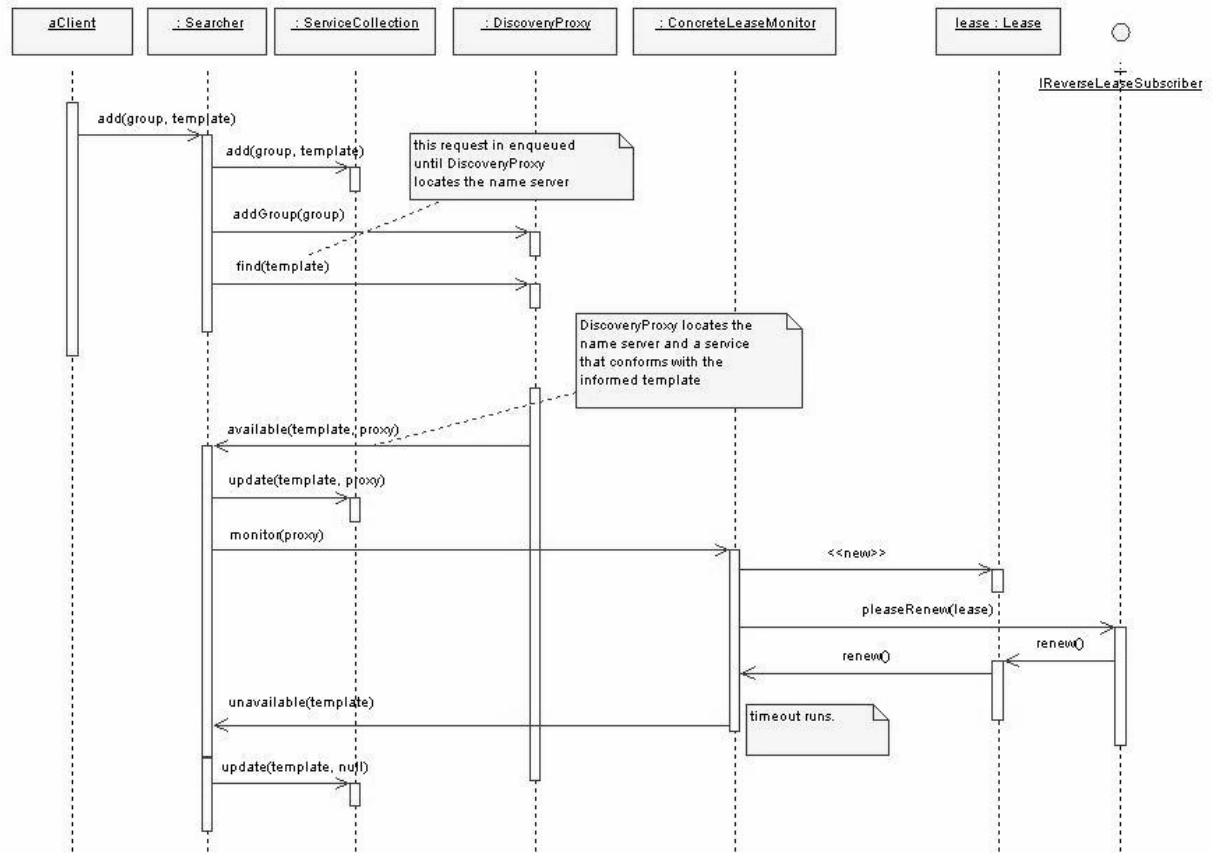


Figure 4: Configuration scenario

- This pattern enforces services to implement an interface the monitor must access in order to start the reverse lease protocol. This is the `IReverseLeaseSubscriber` interface. This requirement may not scale well if a client needs to access services on heterogenous networks, when we cannot enforce which interfaces services should implement.

In the following, we present how the forces mentioned earlier are addressed by this pattern:

- *Force 1* - The monitor is aware of unavailability of service without requiring any communication with it. It is a kind of negative acknowledgement.

- *Force 3* - RLS defines a tradeoff between reliability and performance. Differently from mechanisms based on active tests, with reverse leasing the client component is supposed to suffer lower impact on performance. This occurs because communication overhead is charged to the service component. An active approach, like Heart-Beat, that creates a separate thread of control on the client to test for service availability is likely to consume much more CPU. It also introduces a non-despicable delay because it opens a connection to the server endpoint just to test if it is alive.

2.4 Leasing (Patlet)

Intent

The leasing pattern simplifies resource management by specifying how resource users can get access to a resource from a resource provider for a pre-defined period of time.

Author

Prashant Jain & Michael Kircher

Context

Systems where resource usage needs to be controlled to allow timely release of unused resources.

Forces

- *Simplicity* - The management of resources should be simple by making it optional for the user to explicitly release the resources that it no longer needs.
- *Availability* - Resources not used by a user should be freed as soon as possible to make them available to new users.
- *Actuality* - The system load caused by unused resources must be minimized. Moreover, a user should not use an obsolete version of a resource when a new version becomes available.

Solution

The concept of leasing is that resource access is valid only for a period of time, which is negotiable between the entities involved in the lease protocol. The protocol establishes a contract between parties involved. A lease should be renewed within the agreed time to avoid the contract from being discarded. This causes the resource to be freed. Therefore, if the client of a resource crashes, the resource it used will not remain allocated indefinitely and inconsistently to the client, but only until the lease to that client expires. The participants in the leasing protocol are the *leased resource*, the *lease grantor*, and the *lease holder*. A resource is an abstract concept and may be memory, computation, event notification, among others. A lease grantor is the entity serving the resource, and the lease holder is the client of such resource. In general, the lease holder is a client component and the lease grantor is the service.

Example

For example, consider that service registrations are leased on a name server. Requesting to register the service proxy on the name server causes a lease to be returned. Through such lease the service should reinforce its interest on being located. We can expect that when services fail, registration leases are not renewed. As a result, the name server will clean up offers whenever leases become expired. In this example, the lease resource is

represented by the service offers (proxies) stored on the name server, the lease grantor is represented by the name server; and lease holders by the service implementation.

3 Applicability

When the distribution platform guarantees the availability of a service through some mechanism like that specified in fault-tolerant CORBA [11], this approach does not seem of practical use as the platform introduces a transparent server-side layer to implement fault-tolerance. On the other hand, the distribution platform may indeed use this approach under-the-covers to detect failures of its dependants components. Likewise, this pattern does not seem suitable for services designed in accordance to a *group membership* protocol [17, 18] as this protocol assures the atomicity and consistency of service groups transparently to the user.

We regard this pattern as suitable to monitor and automatically reconfigure distributed systems that comply with a service trading semantics on large-scale networks such as computational grids [10, 8], and service federations [3, 24]. On asynchronous and dynamic environments it is unlikely that clients will know exactly where services are located or the names they should be referenced. We believe that, in this context, services should be located by properties and interfaces.

4 Known uses

The pattern aims at failure-detection on highly-distributed software systems like those supposed to be developed with the support of an Internet service federation. The JTrader system [2] is an example of such a federation of services that is based on the Jini technology. Services and clients are able to join this federation with the support of a toolkit [4] which actually uses this pattern to control availability of the services on the federation.

5 Related patterns

Facade [9, 5] and Proxy [9] - The `DiscoveryProxy` resembles a facade as long as users are not able to access directly name servers instances. Note that name servers may also fail. This component provides a public interface whereby clients can configure which name servers are likely to be contain the intended service offers. Passing a service template, clients can issue a lookup operation as if the local `DiscoveryProxy` were the real name service. So it also behaves like a proxy [9, 6].

Mediator [9] - The `Searcher` complies with the mediator pattern as long as it drives the interaction among dependent components, say `DiscoveryProxy` and `ServiceCollection`.

Lease [13] - According to the lease pattern, the `ConcreteLeaseMonitor` is a lease grantor and the `IReverseLeaseSubscriber` the lease holder.

Publisher-Subscriber [9] - According to the Observer pattern, also known as Publisher-Subscriber, the `Searcher` performs the role of the subscriber and the `AbstractMonitor`, the role of the publisher. This enables asynchronous messaging between these components.

Acceptor-Connector [21] - *"The Acceptor-Connector design pattern decouples connection establishment and service initialization in a distributed system from the processing performed once a service is initialised. This decoupling is achieved with three components: acceptors, connectors, and service handlers"* [21]. According to this pattern, we can regard the `IReverseLease Subscriber` as an instance of an asynchronous *acceptor* component, while the `ConcreteLease Monitor` performs the role of the *connector*. The `Lease` is the *service handler* used to contact the *client handler*, represented by the `Landlord` interface.

Service Configurator [14] and Component Configurer [20] - Similarly to these patterns the Reverse Lease Subscriber aims at configuring components with their dependencies. However, they also present significant differences. For example, the discussed pattern does note tackle the problem of starting new instances when a service fails because we rely on a trading mechanism and thus we expect that a new running service, which implements the required interface, be available on the system. In addition, Service Configurator and Component Configurer assume the existence of a centralized name server (or manager) where the services are supposed to be registered. In turn this pattern relies on the trading semantics where services are discovered in accordance to their characteristics, which include the interfaces they implement. Finally, we observed that these patterns do not aim at supporting adaptability to runtime changes, but just to static or user-directed changes[4], while the Reverse Lease Subscriber deal with fault-detection on component bindings (connections).

6 Acknowledgments

We would like to thank our colleagues at Universidade Federal de Pernambuco - Paulo Borba and Vander Alves - who made suggestions about content to improve this paper. During the PLoP conference Jim O. Coplien and Jerffeson Souza were also very helpful suggesting us to format this pattern as a pattern language. Thanks to all PloPers.

We are also very grateful to Brian Wallis, the sheperd we were granted during the revision process of the SugarloafPLoP conference, for his comments and careful supervision throughout the process.

References

[1] K. Birman. Virtual Synchrony Model. *Reliable Distributed Computing with the Isis Toolkit.*, 1994.

[2] Marcelo B. d'Amorim. Service Trading on the Internet, the JTrader approach, May 2001. Univ. Federal de Pernambuco. M.Sc. dissertation.

[3] Marcelo B. D'Amorim and Carlos A. G. Ferraz. A Design for JTrader: an Internet Trading Service. In Thomas Böhme and Herwig Unger, editors, *Proceedings of the Innovative Internet Computing Systems, International Workshop IICS 2001*, volume

[4] A kind of configuration which is triggered by the intervention of the user, probably a system administrator.

2060 of *Lecture Notes in Computer Science*, pages 159–166, Ilmenau, Germany, 21th-22th June 2001. Springer.

[4] Marcelo B. D'Amorim and Carlos A. G. Ferraz. Designing Jini distributed services: A framework to support the development of reliable component networks. In David H. Lorenz and Vugranam C. Sreedhar, editors, *Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 60–67, Tampa Bay, Florida, October 15 2001. Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115.

[5] Douglas Schmidt et al. *Pattern-Oriented Software Architecture: Patterns for Concurrency and Networked Objects*. John Wiley & Sons, September 2000.

[6] Frank Buschmann et al. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, August 1996.

[7] Ken Arnold et al. *The Jini Specification*. Addison-Wesley, December 1999.

[8] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object Oriented Software*. Addison-Wesley, Jan. 1995.

[10] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Legion: An Operating System for Wide-Area Computing. *IEEE Computer*, 32(5):29–37, May 1999.

[11] Object Management Group. *Fault Tolerant CORBA Specification*, December 1999.

[12] IONA and Isis. *An Introduction to Orbix+Isis*. IONA Technologies Ltd. and Isis Distributed Systems Inc., 1994.

[13] Prashant Jain and Michael Kircher. Leasing. *Pattern Language of Programming - PLOP'2000. Allerton Park, Monticello, Illinois, USA*, 13th–16th Aug. 1996.

[14] Prashant Jain and Douglas C. Schmidt. Service Configurator: A Pattern for Dynamic Configuration and Reconfiguration of Communication Services. In *3rd USENIX Annual Pattern Languages of Programming Conference, Allerton Park, Illinois*, pages 209–219, 1997.

[15] J. Kramer and J. Mageee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, 1985.

[16] S. Landis and S. Maffeis. Building Reliable Distributed Systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[17] Silvano Maffeis. Electra-Making Distributed Programs Object-Oriented. In *Proceedings of the Usenix Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 143–156, San Diego, CA (USA), 1993.

[18] Silvano Maffeis and D. Schmidt. Constructing Reliable Distributed Communications Systems with CORBA. *IEEE Communications Magazine*, 35(2):56–61, 1997.

[19] Balachandran Natarajan, Aniruddha S. Gokhale, Shalini Yajnik, and Douglas C. Schmidt. Applying patterns to improve the performance of fault tolerant CORBA. In *HiPC*, pages 107–120, 2000.

[20] Francisco Assis Rosa and Antonio Rito Silva. Component Configurer. In *Proceedings of the 2nd European Conference on Pattern Languages of Programming - EuroPLoP '97*, pages 209–219, 1997.

[21] Douglas C. Schmidt. Acceptor-Connector: An Object Creational Pattern for Connecting and Initializing Communication Services. In Frank Buschman Robert C. Martin, Dick Riehle, and John Vlissides, editors, *Pattern Languages of Program Design 3*. Addison Wesley, 1997.

[22] António Rito Silva, Fiona Hayes, Francisco Mota, Nino Torres, and Pedro Santos. A pattern language for the perception, design and implementation of distributed application partitioning. October 1996. Proceedings at OOPSLA'96 Workshop on Methodologies for Distributed Objects.

[23] OMG Specification. *Trader Update Package*, 1997. OMG97M CORBA Services Specification, Trader Update Package, OMG document FORMAL/97-618, Available at ftp://ftp.omg.org.

[24] Steven D. Gribble et. al. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Special Issue of IEEE Computer Networks on Pervasive Computing (to appear)*, 2000.

[25] Sun Microsystems. *RMI Wire Protocol, The Acknowledgement Ping Packet*. Available at http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmi-protocol3.html.

[26] Sun Microsystems. *Java Remote Method Invocation Specification*, 1.50 edition, October 1998.

[27] R. van Renesse, K. Birman, B. Glade, K. Guo, M. Hayen, T.M. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus, a Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

Proxy-to-Proxy*

A Design Pattern for Leveraging Security on Highly distributed Internet Applications

Marcelo B. d'Amorim and Carlos Ferraz

{mbd,cagf}@cin.ufpe.br
Universidade Federal de Pernambuco
Centro de Informática
Caixa Postal 7851, 50640-970, Recife-PE, Brazil

Abstract

Internet distributed applications often have to deal with security in design and, depending on the kinds of applications and users they have, different solutions to tame this issue should be applied. When the application is designed in accordance to a strict architectural layering and a single remote object provides every system operation, dealing with security is supposed to be simpler. This is due to the fact that users firstly access this facade object, which should be quite permanently registered on the firewall. This arrangement allows the facade to receive incoming calls through the network and mediate access to other objects. However, when users need to reference remote objects that appear on the network in a non-deterministic basis, an approach to dynamically register and access these objects under the firewall is required. This work defines a pattern to handle security on systems like JTrader, a Jini-based service federation on the Internet, whereby an unpredictable number of remote objects are supposed to be registered. In this case, the federation dictates the way objects are to be made reachable.

Intent

Specify a means to configure distributed services on an application-level gateway [8]. The Proxy-to-Proxy pattern defines a reference monitor [14] to a service and its creational mechanism. The monitor extends a service with new capabilities without affecting its code. It also gives rise for separation of concerns, which are not related to the service functional aspects, like security in a worldwide service federation.

*This paper was presented on the First Latin American Conference on Pattern Languages of Programming - SugarloafPLoP. October 3-5, 2001 - Rio de Janeiro, Brazil.

Motivation

Security is a very important issue to enable distributed computing as it must be resolved without opening ports for hackers nor imposing barriers to developers and users. A suitable approach to handle security in a particular system should consider in what conditions such system should be used. Internet users, for example, are unlikely to configure proxies and firewalls in order to access distributed objects. Even applications where security is considered well handled sometimes face problems when a new component is introduced or a new user is prevented from accessing the system due to a severe security policy or improper configuration. In general, when a very strict and specific security policy is used to enable secure client-server interoperability, users and services may face difficult configuration problems.

In addition, most solutions introduce some kind of overhead to resolve security. For example, in order to control access to enterprise resources, system administrators commonly define a small set of ports which Internet users are able to contact, thus imposing a bottleneck in the system. Actually, very often system administrators open only one port per protocol or maybe none.

In this light, security is very related to two conflicting design issues:

- *Scalability*- The introduction of new elements, such as a new user trying to access the system outside the trusted network or a new server-side component on a large-scale network, may influence security. This issue is especially important concerning a scenario whereby users require accessing a system without previous configuration, such as the Internet.

- *Performance*- Adding new components between clients and servers to tame security is likely to introduce computational and communication overhead.

Security support for client-server applications is supposed to be better handled than that for really distributed ones. There are plenty of application-level gateways[1] [8, 11] which can monitor remote calls to a server through a firewall. The system administrator is only charged with configuring the gateway to accept calls on the service facade object, representing the entry-point of a system [16]. The system administrator could indeed define, in a declarative manner, which operations are allowed access and to which users. Note this approach differs from that of [7] as it attempts to declare permissions in a centralized server, that is the application-level gateway.

However, when there is not a single object to be configured on the gateway but several, when they become available on a non-deterministic basis, and also when these objects do not run on the same machine but are distributed on a local area network; static configuration of an application-level gateway is unlikely to succeed. This work defines a pattern to deal with this dynamism. In the following sections we define its *structure* and *behavior*. That is, the major classes and its dependencies with the framework and, the sequence of tasks that objects should perform to automatically register services with the gateway.

[1] In contrast to circuit-level, application-level gateway are able to monitor an specific kind of protocol communicated between peers and thus, they are only allowed to impose constraints/rules with regard to that protocol.

Forces

The pattern should conform to the following forces:

1. The major concern of this pattern is to enable configuration of remote services on the firewall in a dynamic way. That is, the binding can occur during the application execution.

2. It must resolve limitations related to the service runtime platform. In the case of RMI (Remote Method Invocation) [9], for example, the RMI name server enforces remote objects to execute on the same host it executes, which is not a desirable feature in order to conserve scalability. In this case, all system's objects must have been running on the same host, that is, the host where the name server was running.

3. In order to ease maintenance, and increase legibility and extensibility; it must be easy to isolate responsibilities of components based on the separation of its concerns.

Example

Considering the case an RMI proxy is selected as an application-level gateway, it enables transparent connectivity both in a very restricted configuration where a firewall does not allow outbound or inbound TCP connections, as in an environment that allows these connections by a SOCKS server. Naturally, an RMI proxy also supports connections when there are no firewalls involved, as the case of ISP (Internet Service Provider) POP3 connections. Figure 1 illustrates a configuration where RMI Registry [15, 9] does not execute on the same host as the RMI proxy. Due to load balancing, this organization is well suited for proxy servers that need to host several objects.

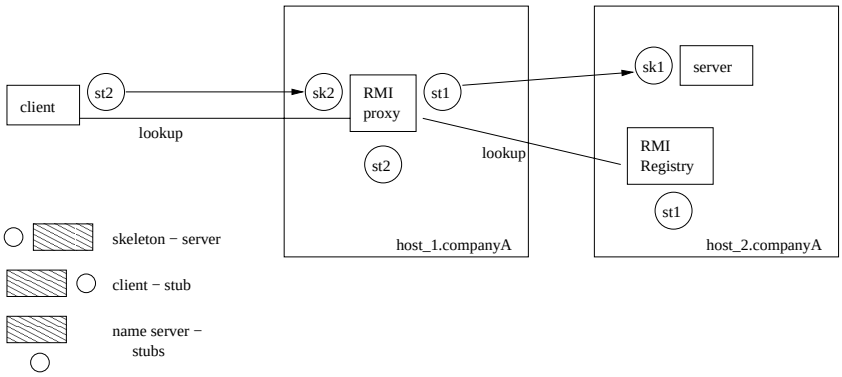


Figure 1: RMI Proxy organization using multiple hosts

In the light of the diagram legends, we conclude the RMI proxy actually performs three tasks: it is a name server and also a server to RMI clients, and it is a client to the actual remote object implementation. This arrangement allows clients to access a single and secure endpoint - the RMI Proxy - that conveys communication traffic to a local network endpoint where the remote object is supposed to be running. As a consequence of being also a name server, Figure 2 presents an organization where the RMI Registry

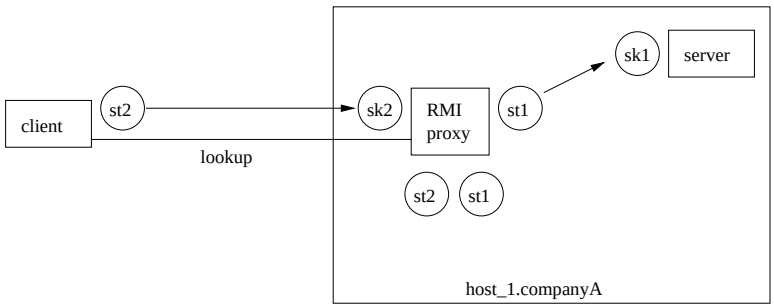


Figure 2: RMI Proxy organization

component is discarded and the remote object uses the RMI Proxy to bind a name. This is only possible because the RMI proxy is also a name server.

As soon as the facade object, representing the service, is exported to the RMI system and properly bound to the RMI proxy, it can be accessed in remote locations. However, this is not enough to resolve communication since the facade object should indeed return other remote object references. If the facade returns to the client a stub to an object running on some network host, the client would unsuccessfully try to access that remote object directly. It should be accomplished through the RMI proxy. Therefore, remote objects returned by the facade should also be registered on the RMI Proxy whenever they are requested. This causes the gateway (RMI Proxy) to create a skeleton object *on-the-fly* (sk2) to receive communication on behalf of the actual server (server), and then returns a remote stub (st2) that refers to this skeleton, rather than to the actual skeleton object (sk1). Another problem occurs when the federation does not have access to remote references in order to register them with the RMI proxy. In fact, this situation is very common when working with Jini [1] as it very often represents remote objects as non-remote proxy objects [2]. In this case, the actual remote reference is stored within the proxy object and we do not have access to it.

The Proxy-to-Proxy approach defines another level of indirection in order to access these objects remotely. The pattern defines a proxy object that stores a reference not to the actual remote object, but to a new remote one that is to be bound to the application-level gateway. This new remote object stores in turn a reference to the original (actual) proxy object, so it delegates operations to it.

Figure 3 sketches the dependency between the additional proxy object and the new remote object created to provide firewall access to a Jini Lookup Service (on the left-side of the diagram), and also the dependency between the original Lookup Service and its proxy (on the right-side of the diagram). Note the `JTServiceRegistrar` has a dependency with a `ServiceRegistrar` type, which is the interface Lookup Service proxies should implement. `ServiceRegistrar` is **not** a remote type, so it is **not** enforced to throw `RemoteException` on its operations. This makes sense as the proxy is a local object, and therefore this type only throws remote exceptions when it requires to access the remote object, represented by the `actual_ServiceRegistrarImpl`.

[2]The service proxy in Jini is defined as a class that implements the service interface and it is not required to conform to any distribution platform. It may be locally implemented, use sockets, HTTP connection, CORBA, RMI, etc. The proxy is supposed to be downloaded to the client's address space prior to the client access to the service.

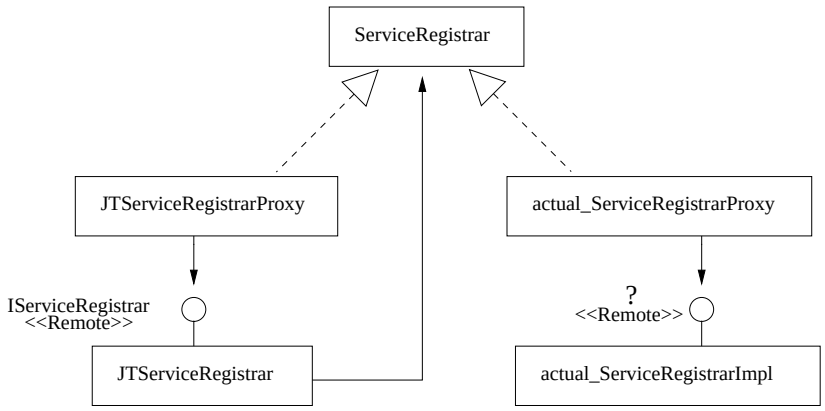


Figure 3: Applying the pattern to the Jini Lookup Service

On the other hand, every operation `JTServiceRegistrarProxy` implements is supposed to call a remote object since it does not store local state due to the need to ensure consistency. So, we have a problem. All `JTServiceRegistrarProxy` operations are remote but not necessarily all `ServiceRegistrar` operation are. Java does not allow overriding operations that throw exceptions not declared in their superclass. This makes sense as a client cannot catch an unexpected exception from an interface implementation. Therefore, it means that `JTServiceRegistrar` cannot implement the `ServiceRegistrar` interface directly. Actually, it implements an equivalent interface that throws remote exceptions in all operations, the `IServiceRegistrar` interface, which is a `Remote` interface. Similarly to the right side of the diagram, the specific protocol that the proxy and the remote object use is well hidden from their clients.

Solution

In most distribution platforms the concept of a service interface is very often related to communication. In CORBA [10], for example, the stub component behaves as a proxy to the service which is located at the client's address space and so, it must implement the service interface and contact the actual remote object through the wire. However, the concept of a service interface may be made independent of the communication platform selected as Jini [1] suggests. In this context, a proxy is a component that implements the service interface and nothing more. The stubs used in CORBA or RMI are indeed proxy instances and therefore they conform to this principle. However, proxies do not need to be necessarily stubs; they could be any kind of object that simply implements the service interface. It could be a local implementation, or even contact different remote objects to implement the interface. In this work we conform to this principle, which recognizes the difference between a **service proxy** and a **stub**.

The pattern extends the service interface by defining a new type - *Proxy* - that does not call the service directly but uses a different remote implementation to accomplish it - the *IRemoteProxy*. As stated before, the remote proxy implementation - *RemoteProxyObject* - does not necessarily implement the same interface as the proxy. In the same way as the Interceptor [13] and Reflection [2] this pattern gives an opportunity to intercept calls made to the original service implementation.

Figure 4 generalizes the structure of the components from the example presented on

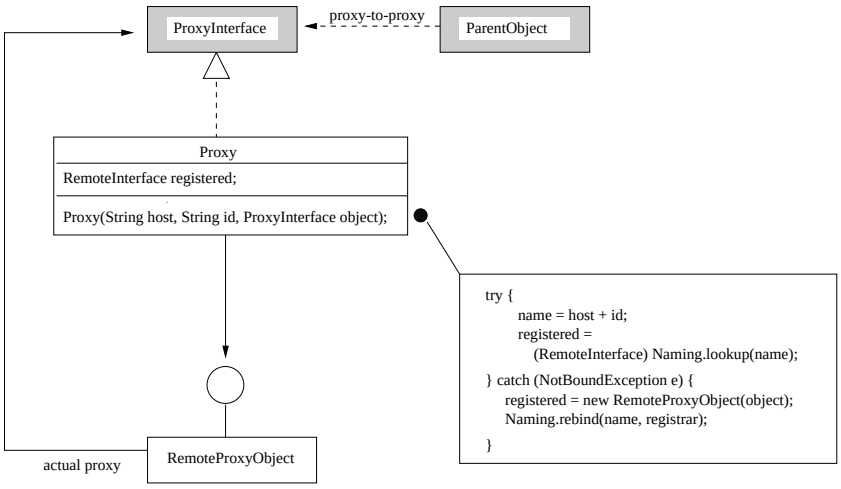


Figure 4: The Proxy-to-Proxy pattern

figure 3.

Applicability

This pattern applies when the number of remote objects managed by the system is relatively large, and therefore static configuration is not well suited because of the non-determinism in which objects appear on the system. The JTrader system [3] is a federation of services based on the semantics of trading, which applies this pattern.

Another common application of the pattern is to enable interception of remote calls. Interception provides, for example, a means to control security, increase fault tolerance and load balancing. However, as stated before, this is not a particular feature of this pattern. In the "Related Patterns" section we contrast the differences between Proxy-to-Proxy and other related patterns.

Static Structure

In the following, we describe classes, their collaborations and responsibilities. The highlighted boxes represent types that already exist prior to the pattern be applied and they play a special role in the pattern behavior.

- **ParentObject** - This type is usually a facade class providing, through its methods, access to other remote objects. Although not presented in this diagram, client components access the `ParentObject` to retrieve references to `ProxyInterface` objects. The parent object in turn should not return references to the *actual proxy* implementation but to a `Proxy` object.

  Responsibilities:

  – Build `Proxy` instances enabled to forward messages to the *actual proxy*[3].

[3]Represented in the association between `RemoteProxyObject` and `ProxyInterface`.

Collaborators: Actual Proxy (not depicted in this diagram), `Proxy`, and `ProxyInterface`.

- **ProxyInterface** - This type appears highlighted in figure 4 because it is the common interface implemented by both the *actual proxy* and the `Proxy`. `ProxyInterface` is the type clients rely on to program their components and it is also a local type since the protocol used to contact a service is supposed to be hidden within the proxy code.

  Responsibilities:

  – Provide a complete set of operation to clients access an object.

  Collaborators: client component (not in the diagram), *actual proxy* (not in the diagram), `Proxy`, `ParentObject`.

- **Proxy** - Proxies implement the `ProxyInterface`.

  Responsibilities:

  – Hide the specific protocol used to contact the remote proxy-to-proxy object.

  – Build an `IRemoteProxy` instance and bind this object to the proper application-level gateway.

  Collaborators: `ProxyInterface` and `IRemoteProxy`.

- **IRemoteProxy** (optional[4]) - This is the remote interface of the proxy-to-proxy remote object. It may declare the same operations as the `ProxyInterface`, but throwing remote exceptions; or declare a completely different protocol.

  Responsibilities:

  – Declare the methods through which a remote object can be called.

  Collaborators: `Proxy` and `RemoteProxyObject`.

- **RemoteProxyObject** - This type, also referred to as the proxy-to-proxy remote object, implements the `IRemoteProxy` interface.

  Responsibilities:

  – Add some capability to the service on top of the original (actual) remote object implementation, which is only supposed to consider the service's functional aspects.

  – Forward calls to the *actual proxy* object.

  Collaborators: `Proxy`, `IRemoteProxy`, and *actual proxy* (not in the diagram).

[4] If every `ProxyInterface` operation is remote or some of its local operation is not supposed to be called, this type may not be required and, in this case, the `RemoteProxyObject` could implement the `ProxyInterface` directly.

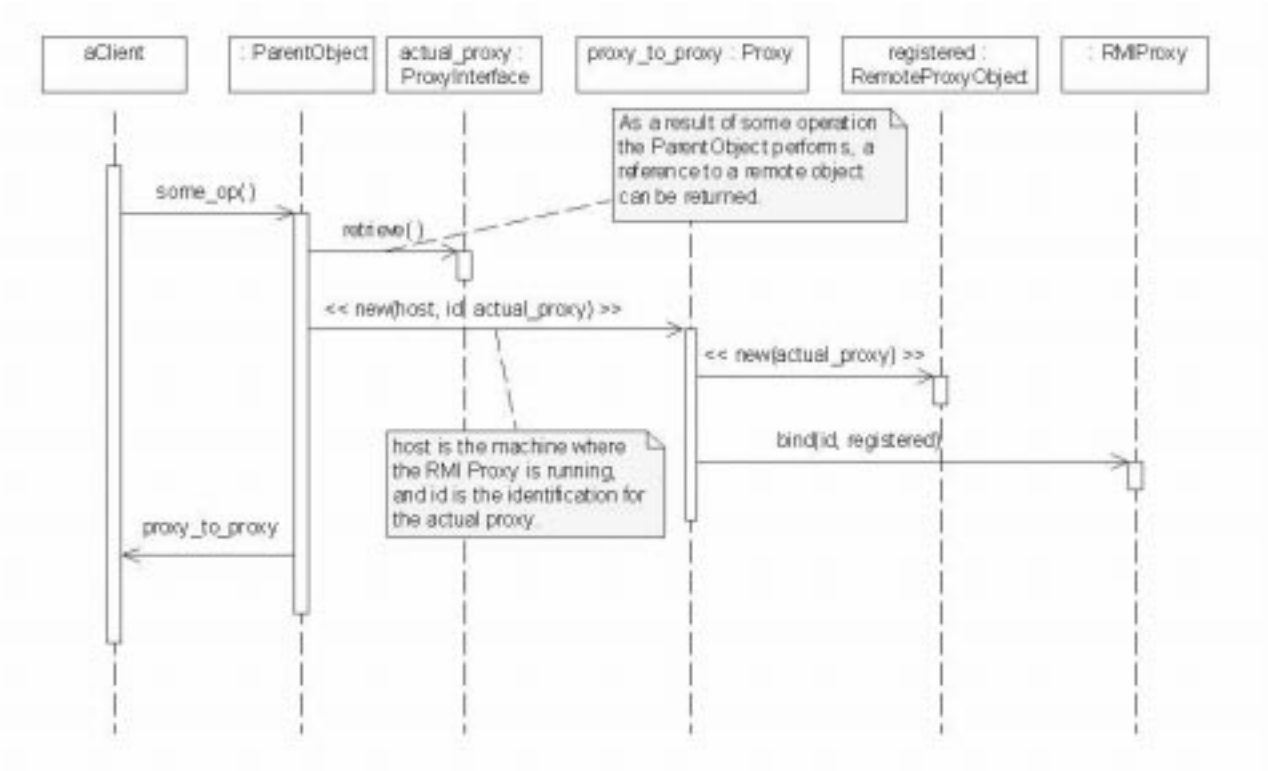


Figure 5: Interaction diagram

Dynamics

The diagram of figure 5 illustrates the interactions between the components mentioned above and the following steps, providing additional meanings to the invocations represented on the diagram.

1. The client invokes an operation, say `some_op`(), on some remote object, represented as an `ParentObject` instance.

2. During the operation execution, the parent object calls a method which returns a reference to another remotely accessible object, here represented by the `ProxyInterface` type.

3. Instead of returning that remote object directly to the user, a new proxy object (`Proxy`) is to be created, passing as parameter the host in which the application-level gateway is running, an identification for the object, and a reference to the actual proxy object (`ProxyInterface`).

4. Create the `RemoteProxyObject` that the `Proxy` refers to.

5. Bind the `RemoteProxyObject` to the application-level gateway.

6. Finally, the `Proxy` object is returned[5] to the client through the method invoked on the `parentObject`. Such object actually stores a reference to a remote object,

[5]Despite we suggest the reader that the proxy is actually a full-fledged object, it may be just a reference to the `RemoteProxyObject`. By the way, this is how remote objects are passed (IOR) in CORBA remote invocations. Note that we have stated earlier that stubs are also a special kind of proxies.

registered on the application-level gateway, which behaves like a reference monitor to the original service.

Consequences

As design is very often an engineering between competing forces, this pattern is not an exception. We describe in the following negative consequences this pattern brings.

- *Garbage Collection.* As long as the `RemoteProxyObject` only forwards messages to the original remote object, such actual remote object can fail and, in this case, the `RemoteProxyObject` instance would remain registered on the application-level gateway (e.g. RMI Proxy). This situation leads to a more difficult procedure for garbage collection because of the dependency between these objects.

- *Performance.* Due to security reasons, an application-level gateway very often uses a single network port to convey communication to objects registered on it. Besides the performance overhead a firewall usually introduce for this reason, the proxy-to-proxy arrangement introduces an additional indirection between a remote object and its clients.

Even though the proxy-to-proxy strategy introduces overhead related to indirection of remote calls and also turns far more difficult managing garbage collection on the firewall, it introduces some relevant benefits:

- *Introduce additional capabilities.* It is possible to modify the default behavior of an object as we can intercept calls made to a proxy. Therefore, an operation can be modified or we can introduce additional capability to the service, such as dealing with faults on remote references or balancing the system load. Moreover, applying some security policy based on a version of the access matrix [12] is direct. The proxy-to-proxy object, for instance, may be in charge to searching for access rights on the matrix on behalf of the actual remote objects.

- *Separation of concerns.* As the original service implementation is well decoupled from the proxy-to-proxy, we can reuse these capabilities in similar applications.

- *Locality Freedom to the Service.* According to the RMI specification [15], RMI server objects must run on the same host as the RMI Registry. It means remote objects should run on the same machine as the RMI proxy. Actually, this requirement imposes a hard constraint that mainly affects scalability on distributed systems. However, the proxy-to-proxy arrangement resolves such limitation. The remote object registered on RMI Proxy is not the remote object itself, but a second one that behaves like the original and forwards messages to it. In other words, the object registered on the RMI proxy is a client of the actual remote object and then, it relieves the original server (*actual proxy*) from running on the application-level gateway. However, the proxy-to-proxy is also a server and so, the object which creates it (`ParentObject`) should be running on the same host as the application-level gateway.

- *Enable Activation*[6]. Even if service proxies allow access to remote object references, the RMI Proxy may not permit to register remote references. This situation happens when the reference is to an activatable remote object. This special kind of remote objects is used very often in Jini service to increase fault-tolerance and avoid resource waste [9], but it is not well supported by current RMI Proxy implementations [11]. When using the proxy-to-proxy approach, this constraint does not apply for the same reason as the *locality freedom* is achieved - the `RemoteProxyObject` registered on the RMI Proxy is a client of the actual proxy and it can be implemented as an `UnicastRemoteObject` [15], rather than an activatable.

Related patterns

Proxy [5, 2] - This pattern can be considered an application of the Proxy pattern [5, 6] as the interceptor proxy implements the same interface as the service interface.

Security patterns [16, 7, 4] - Conversely to security patterns; which provide guidelines on how to control access to resources, provide user authentication, establish roles and sessions, and govern the way exceptions are thrown due to lack of privileges; this work does not cover these aspects at all, but defines an approach to configure remote services on a network firewall by means of delegation.

Reflection [2] and the **Interceptor** [13] - As in these patterns, Proxy-to-Proxy requires an additional indirection between the client and the service provider and thus it has some similarities with both. These patterns depend on the underlying framework to implement indirections. For example, according to the Interceptor pattern [13], a concrete framework must provide a dispatcher with which interceptors are to be registered. Proxy-to-Proxy, however, does not rely on the underlying framework. Even though it means additional complexity to services in order to control the registration of the proxies on the application-level gateway, the solution does not depend on a given framework. In this context, this pattern could be interpreted as a lightweight version of the Interceptor pattern [13] where the `Proxy` would be an Interceptor at the client-side and the `RemoteProxyObject` an Interceptor at the server-side.

Single Access Point [16] - *Security applications should not allow users to get through a back door that allows them to view or edit sensitive data.* **Single Access Point** *helps solve this problem by limiting application entry to one single point* [16]. Proxy-to-Proxy is useful for decentralized component networks, and therefore, it does not rely on any central manager to control security, as a facade [5, 13] object.

Acknowledgments

We would like to thank our coleagues at Universidade Federal de Pernambuco who made suggestions about content and format to this work.

[6] This framework supports persistent remote references, which enable clients to rebind to remote objects after a failure.

We are also very grateful to Eduardo Fernandez, the sheperd we were granted during the conference revision process, for his contributions and careful supervision.

References

[1] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification.* Addison-Wesley, December 1999.

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

[3] Marcelo B. d'Amorim and Carlos Ferraz. A Design for JTrader - an Internet Trading Service. In *proceedings of the Innovative Internet Computing Systems - I2CS. Ilmenau, Germany.* Springer Verlag, Lecture Notes in Computer Science (LNCS), 21th–22th June 2001.

[4] Eduardo B. Fernandez. Metadata and Authorization Patterns. In *TR-CSE-00-16*, May 2000. Dept. of Computer Science and Eng. Florida Atlantic University.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object Oriented Software.* Addison-Wesley, Jan. 1995.

[6] M. Grand. *Patterns in Java, A Catalog of Reusable Design Patterns Illustrated with UML*, volume 1. John Wiley & Sons, New York, NY, USA, 1998.

[7] Viviane Hays, Marc Loutrel, and Eduardo B. Fernandez. The Object Filter and Access Control Framework. In *proceedings of the 7th Conference on Pattern Languages of Programming, Monticello, IL.*, 2000.

[8] Martin W. Murhammer, Orcun Atakan, Stefan Bretz, Larry R. Pugh, Kazunari Suzuki, and David H. Wood. *TCP/IP Tutorial and Technical Overview.* IBM Corporation, International Technical Support Organization, 6th edition, October 1998.

[9] Richard Oberg. *Mastering RMI: Developing Enterprise Applications in Java and EJB.* Wiley, 2001.

[10] Object Management Group. *CORBA/IIOP Specification*, 2.3.1 edition, October 1999.

[11] Esmond Pitt and Neil Belford. *The RMI Proxy.* Telekinesis Inc., 2000.

[12] R. S. Sandhu and G. S. Suri. Implementation Considerations for the Typed Access Matrix Model in a Distributed Environment. In *15th National Computer Security Conference*, pages 221–235, 1992.

[13] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrency and Networked Objects.* John Wiley & Sons, September 2000.

[14] Rita C. Summers. *Secure Computing: Threats and Safeguards.* McGraw-Hill, 1997.

[15] Sun Microsystems. *Java Remote Method Invocation Specification*, 1.50 edition, October 1998.

[16] J. Yoder and J. Barcalow. Architectural Patterns for Enabling Application Security. In *proceedings of the 4th Conference on Pattern Languages of Programming, Monticello, IL.*, September 1997.

Uma Coleção de Padrões para o Gerenciamento de Sessão em Aplicações Internet

MÁRCIO DE OLIVEIRA BARROS **ALEXANDRE LUIS CORREA** **CLÁUDIA MARIA LIMA WERNER**

COPPE / UFRJ – Departamento de Engenharia de Sistemas e Computação
Caixa Postal: 68511 - CEP 21945-970 - Rio de Janeiro – RJ
Telefone: 5521 562-8675 / Fax: 5521 562-8676
{marcio, alexcorr, werner}@cos.ufrj.br

Resumo

Aplicações Internet têm características particulares que as diferenciam de aplicações convencionais. Dentre estas características encontra-se o fato da comunicação entre os clientes e os servidores, na maioria das aplicações, ocorrer através de um protocolo que não possui memória das interações anteriores entre estes elementos. Assim, tornam-se necessários mecanismos de gerenciamento de sessão, que garantem a persistência das informações adquiridas entre as diversas páginas componentes de uma aplicação Internet. Neste artigo apresentamos três padrões que tratam do gerenciamento de sessão em aplicações Internet.

1. Introdução

O crescimento dos mercados de comércio eletrônico, *home-banking* e de prestação de serviços através da Internet fez crescer a demanda pelo desenvolvimento de aplicações acessíveis através desta plataforma. Entretanto, as aplicações Internet possuem características e limitações particulares que as diferenciam de aplicações convencionais. A Figura 1 apresenta os principais elementos envolvidos em uma aplicação Internet.

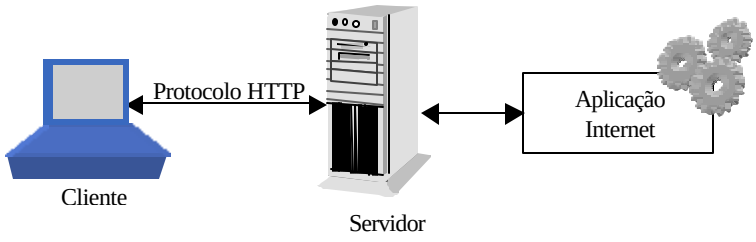


Figura 1 – Principais elementos envolvidos em uma aplicação Internet

- *Cliente:* representa o lado cliente da aplicação Internet. As aplicações Internet seguem o modelo cliente-servidor, onde diversos computadores (atuando como clientes) requisitam os serviços oferecidos por um ou mais servidores. Nas aplicações Internet, o lado cliente geralmente é responsável apenas pela execução do navegador. O navegador recebe páginas e arquivos complementares (imagens, folhas de estilo, sons, entre outros) enviadas pelo lado servidor, apresenta as páginas recebidas e monitora a ativação de suas ligações com outras páginas, requisitando as páginas ativadas ao servidor;
- *Servidor*: representa o lado servidor de uma aplicação Internet, sendo responsável pela transferência das páginas requisitadas pelos navegadores. Enquanto os navegadores

somente recebem e tratam as páginas e seus arquivos complementares, uma requisição ao servidor pode ativar uma aplicação que realiza um processamento, gerando uma resposta para o navegador no lado cliente. A resposta geralmente se constitui de uma página, que pode ser estática ou construída dinamicamente através da utilização de outros recursos do lado servidor, como, por exemplo, bases de dados. O servidor pode ser visto como um *cluster de servidores*, isto é, um conjunto de servidores que atendem pedidos para uma mesma aplicação;

- *Protocolo HTTP* [W3C 1999]: é o protocolo de comunicação entre o cliente e o servidor, através do qual as requisições por páginas são realizadas. Uma importante característica deste protocolo é a falta de memória, isto é, o protocolo não armazena um histórico das páginas previamente requisitadas nem das informações contidas nestas páginas;
- *Aplicação Internet:* representa a aplicação executada no lado servidor, responsável pelo processamento das informações enviadas pelos clientes.

A incapacidade do protocolo HTTP manter memória das interações ocorridas entre clientes e servidores é uma importante limitação a que estão sujeitas as aplicações Internet. Esta limitação exige a criação de mecanismos para o armazenamento das informações coletadas ao longo destas interações. Tais mecanismos são denominados *gerenciamento de sessão*.

A seguir apresentamos três padrões que tratam do gerenciamento de sessão em aplicações Internet. Na construção destes padrões seguimos a proposta de Meszaros e Doble [Mesz 1997], descrita no padrão *Common Problems Highlighted*, que sugere que a estrutura comum compartilhada por um conjunto de padrões seja destacada em um padrão separado. Assim, o primeiro padrão, *Gerenciamento de Sessão na Internet*, contém a estrutura comum aos dois últimos padrões, *Gerenciamento de Sessão Baseado em Cookies* e *Gerenciamento de Sessão Baseado em Parâmetros*.

2. Gerenciamento de Sessão na Internet

2.1. Contexto

Em um sistema de compras pela Internet, um usuário se identifica, fornecendo seu nome e sua senha, seleciona um conjunto de produtos e realiza a compra, indicando outras informações como, por exemplo, a forma de pagamento e o local de entrega. Este processo, com pequenas variações, repete-se em diversas aplicações envolvendo a aquisição de produtos pela Internet, como supermercados virtuais, livrarias, distribuidores de software, entre outros.

O carrinho de compras, que contém os produtos selecionados e suas respectivas quantidades, é uma informação ligada à sessão do usuário. Uma sessão é definida como o uso consistente de uma aplicação Internet por um período de tempo, podendo se estender pelo acesso a diversas páginas que pertençam à aplicação. Por consistente, entendemos que a sessão tem um objetivo principal. No exemplo acima, o objetivo de uma sessão é a aquisição de um conjunto de produtos. A cada nova sessão, o usuário terá um novo carrinho de compras.

2.2. Problema

O problema reside em como garantir a persistência das informações contidas em páginas previamente percorridas pelo usuário, de forma que estas informações possam ser

posteriormente utilizadas pela aplicação Internet, mesmo que esta utilize um protocolo sem memória, como o HTTP.

2.3. Forças

- Aplicações Internet geralmente utilizam informações residentes em páginas previamente percorridas pelo usuário. Entretanto, o principal protocolo utilizado nestas aplicações – o protocolo HTTP – não possui memória destas informações;
- As informações armazenadas no lado cliente de uma aplicação Internet não são seguras, pois podem ser alteradas pelo navegador ou por uma aplicação do cliente;
- Diversas aplicações Internet são executadas dentro de um *cluster* de servidores HTTP. Portanto, dentro de uma mesma sessão, as requisições de um usuário podem ser atendidas por diferentes servidores.

2.4. Solução

A persistência das informações trocadas nas aplicações Internet é garantida com a utilização de mecanismos de gerenciamento de sessão na Internet. Eles permitem que uma aplicação Internet armazene informações sobre as páginas percorridas por um usuário no contexto de uma sessão. Estas informações podem ser utilizadas, por exemplo, para armazenar o carrinho de compras.

Os seguintes mecanismos de gerenciamento de sessão são atualmente utilizados na Internet: *Gerenciamento de Sessão Baseado em Cookies* e *Gerenciamento de Sessão Baseado em Parâmetros*. Estes mecanismos são descritos na Seção 2 e na Seção 3, respectivamente. A Figura 2 apresenta a estrutura geral utilizada pelos mecanismos de *Gerenciamento de Sessão na Internet.*

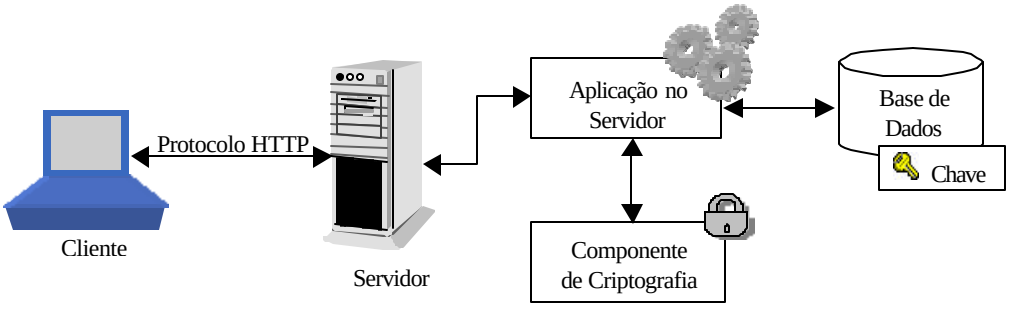


Figura 2 – Estrutura geral da solução *Gerenciamento de Sessão na Internet*

- *Base de Dados*: um repositório de informações utilizado para conter a memória da sessão. Estes dados poderiam ser armazenados na memória do servidor, permitindo um acesso mais rápido à memória da sessão. Entretanto, se a aplicação Internet for executada em um *cluster* de servidores, o algoritmo de distribuição de requisições pode delegar o atendimento das requisições de uma sessão a diferentes servidores, espalhando as informações desta sessão. Desta forma, armazenaremos a memória da sessão em uma base de dados, que pode ser um banco de dados relacional, um servidor especializado em memória de sessão ou um dispositivo de memória compartilhada. A base de dados deve ser acessível para os servidores do *cluster*;
- *Chave*: é a informação necessária para identificar a memória da sessão de um usuário na base de dados. A chave deve ser transferida do cliente para o servidor a cada solicitação e, em função de trafegar pela rede, esta transferência deve ocorrer de forma segura;

- *Componente de Criptografia:* este componente é responsável por tratar uma limitação na transferência de informações entre os lados cliente e servidor: a questão de segurança. Informações provenientes do lado cliente, mesmo que tenham sido previamente enviadas pelo servidor, podem ser alteradas por aplicações do usuário. Isto pode provocar problemas na aplicação Internet, permitindo que a memória da sessão de um usuário seja acessada de forma distinta da originalmente planejada. Para que as informações que passam pelo lado cliente sejam tratadas de forma segura, elas devem ser criptografadas no servidor antes de enviadas e decodificadas quando recebidas.

2.5. Contexto Resultante

Como resultado da aplicação deste padrão criou-se uma memória de sessão identificada por uma chave e funcional tanto em um cluster de servidores como em um servidor independente. A chave é criptografada de forma a ser transferida para o lado cliente da aplicação Internet com segurança.

2.6. Padrões Relacionados

A aplicação dos mecanismos de *Gerenciamento de Sessão na Internet* está relacionada com a aplicação dos seguintes padrões mais específicos que se diferenciam na forma como o tratamento da chave de acesso à memória da sessão é realizado:

- *Gerenciamento de Sessão Baseado em Cookies* armazena a chave de acesso à memória da sessão em cookies no lado cliente.
- *Gerenciamento de Sessão Baseado em Parâmetros* armazena a chave de acesso à memória da sessão em campos escondidos de formulários e em parâmetros embutidos nas ligações estáticas das páginas enviadas para o navegador cliente.

2.7. Usos Conhecidos

Assim como no exemplo do carrinho de compras citado no contexto, outras aplicações Internet, como portais de *home-banking*, páginas de busca, *webmail* e jogos, se baseiam em informações residentes em páginas previamente percorridas pelo usuário e, portanto, utilizam mecanismos de gerenciamento de sessão.

3. Gerenciamento de Sessão Baseado em Cookies

3.1. Contexto

O contexto deste padrão é resultante da aplicação do *Gerenciamento de Sessão na Internet* onde as informações da sessão de um usuário são acessadas através de uma chave.

3.2. Problema

O problema consiste em garantir a persistência da chave utilizada na aplicação do *Gerenciamento de Sessão na Internet* entre os diversos acessos a diferentes páginas sem ocasionar um aumento no código das mesmas.

3.3. Forças

- Aplicações Internet geralmente utilizam informações residentes em páginas previamente percorridas pelo usuário. Entretanto, o principal protocolo utilizado nestas aplicações – o protocolo HTTP – não possui memória destas informações;
- Quando o tratamento da persistência da chave envolve a criação de campos escondidos e parâmetros embutidos nas páginas transferidas do servidor para o cliente, ocorre uma inserção de código específico para tal nestas páginas.

3.4. Solução

A solução consiste em estender a solução do *Gerenciamento de Sessão na Internet*, armazenando a chave criptografada em um cookie no lado cliente [Rubin 1998]. A Figura 3 apresenta os elementos envolvidos na solução.

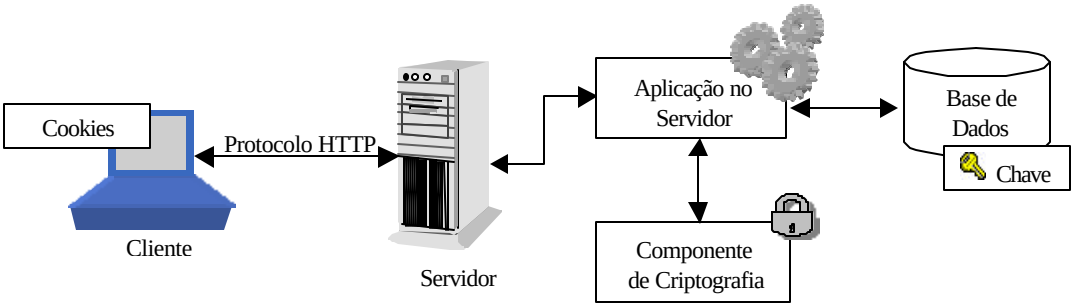


Figura 3 - Estrutura geral da solução *Gerenciamento de Sessão baseado em Cookies*

Um *cookie* é um conjunto de informações armazenadas no lado cliente de uma aplicação Internet. O cookie contém um pequeno trecho de dados, geralmente limitados a quatro Kbytes, sendo composto por pares de "nome-valor", ambos representados por seqüências de caracteres. Um cookie é associado a um conjunto de páginas Internet: sempre que o navegador envia uma requisição de página para o servidor, o cookie associado à página pedida também é enviado. Da mesma forma, o servidor pode enviar um cookie junto com uma página transferida para um navegador. Um cookie possui uma data de expiração, indicada pelo servidor quando este envia o cookie para o navegador cliente. O cookie pode ser programado para expirar no fechamento do navegador ou em uma data fixa.
No primeiro instante em que o servidor identificar a necessidade de guardar memória da sessão do usuário, ele cria uma chave de identificação para a sessão, aplica a criptografia sobre a chave, enviando-a na forma de um cookie para o cliente.

3.5. Dinâmica

A dinâmica de colaboração entre os componentes participantes desta solução é ilustrada pelos diagramas das Figuras 4 e 5. Esta dinâmica pode ser dividida em dois instantes distintos:

a) Cliente realiza uma requisição que faz com que o servidor identifique a necessidade de guardar memória da sessão (Figura 4):
   - Cliente envia uma requisição de página para o servidor contendo informações que deverão ser guardadas para as próximas interações;
   - O servidor identifica a aplicação Internet que deve ser ativada, enviando as informações recebidas do cliente para esta aplicação;

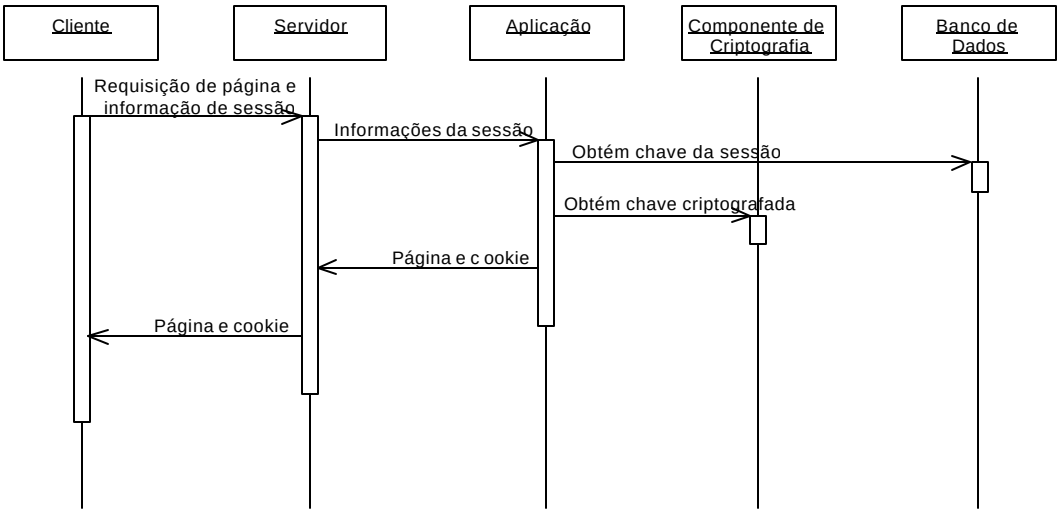


Figura 4 – Diagrama de seqüência para a criação de memória da sessão

- A aplicação Internet requisita que o banco de dados crie uma nova memória da sessão, retornando sua chave;
- A aplicação Internet requisita ao componente de criptografia que codifique a chave da memória da sessão, retornando seu valor criptografado;
- A aplicação Internet compõe um cookie com a chave criptografada e o envia para o servidor, junto com a página resultante do processamento. A expiração do cookie é programada para o fechamento do navegador;
- O servidor envia a página e o cookie recebidos da aplicação Internet para o navegador no lado cliente.

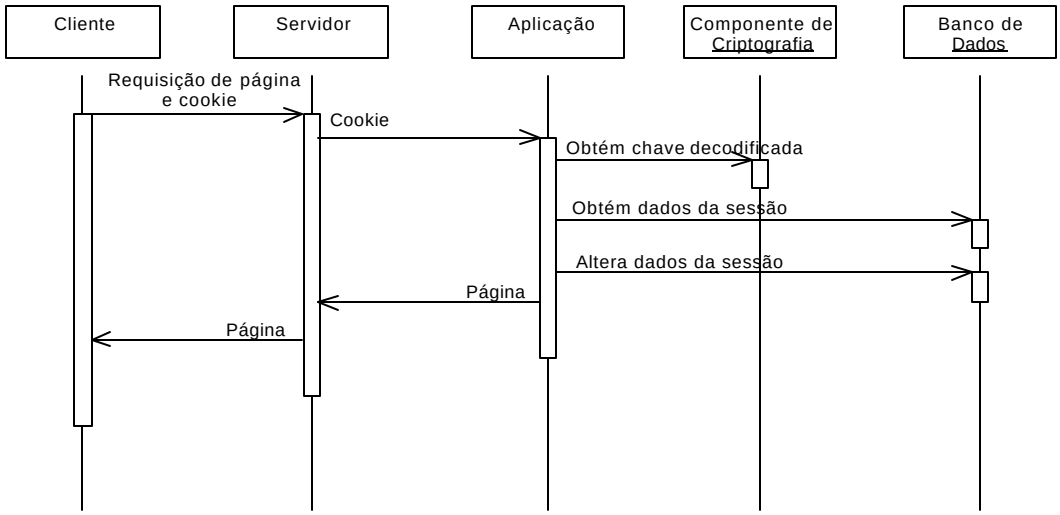


Figura 5 – Diagrama de seqüência para consulta e alteração da memória da sessão

b) Requisições realizadas pelo cliente após a criação do cookie (Figura 5):

- Cliente envia uma requisição de página para o servidor, acompanhada de um cookie com a chave criptografada para a memória da sessão do usuário;

- O servidor identifica a aplicação Internet que deve ser ativada e passa o cookie para esta aplicação;
- A aplicação Internet utiliza o componente de criptografia para recuperar o valor original da chave de acesso à memória da sessão;
- A aplicação Internet consulta e, eventualmente, atualiza os dados da memória da sessão no banco de dados, utilizando a chave de acesso;
- A aplicação Internet utiliza a memória da sessão para compor a página de resposta para o cliente, enviando-a para o servidor;
- O servidor envia a página recebida da aplicação Internet para o navegador no lado cliente.

3.6. Conseqüências

A principal vantagem desta solução é que o código das páginas enviadas para o navegador cliente não precisa ser alterado para o armazenamento da memória da sessão. Além disso, nenhuma lógica é necessária no lado cliente, uma vez que o tratamento dos cookies é realizado automaticamente pelos navegadores.
A principal limitação desta solução é a possibilidade do navegador cliente estar programado para rejeitar cookies. Neste caso, a aplicação Internet não recebe a chave para a memória da sessão do usuário, ficando os dados desassociados da sessão atual.

3.7. Padrões Relacionados

Este padrão é normalmente aplicado para solucionar o problema de armazenamento e transferência da chave que surge quando da aplicação do *Gerenciamento de Sessão na Internet*. *Gerenciamento de Sessão Baseado em Parâmetros* armazena a chave de acesso à memória da sessão em campos escondidos de formulários e em parâmetros embutidos nas ligações estáticas das páginas enviadas para o navegador cliente.

4. Gerenciamento de Sessão Baseado em Parâmetros

4.1. Contexto

O contexto deste padrão é resultante da aplicação do *Gerenciamento de Sessão na Internet onde* as informações da sessão de um usuário são acessadas através de uma chave.

4.2. Problema

O problema consiste em garantir a persistência da chave utilizada na aplicação do *Gerenciamento de Sessão na Internet* entre os diversos acessos a diferentes páginas de forma a atingir indiscriminadamente navegadores clientes na Internet.

4.3. Forças

- Campos escondidos são aceitos por qualquer navegador cliente, sendo enviados para o servidor quando o formulário é submetido;
- Ligações estáticas podem conter parâmetros, que são enviados para o servidor quando a ligação é ativada;
- Um navegador cliente pode estar programado para não aceitar cookies enviados pelo servidor junto com uma página;

4.4. Solução

A solução consiste em enviar a chave de acesso à memória da sessão para o servidor na forma de um parâmetro. Este parâmetro deve ser enviado em todas as requisições do navegador cliente. Assim como nos cookies, os parâmetros também trafegam pela rede, devendo ser criptografados no lado servidor antes de enviados para o lado cliente. A Figura 6 apresenta a estrutura da solução proposta pelo padrão.

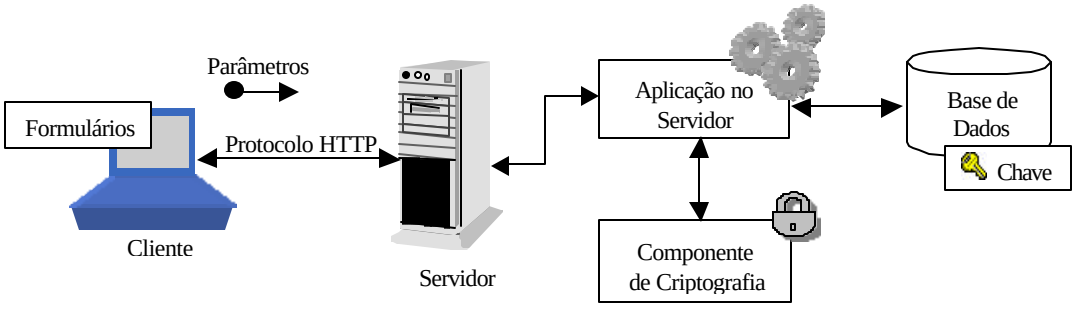


Figura 6 – Estrutura da solução de gerenciamento de sessão baseado em parâmetros

- *Formulários*: as páginas de uma aplicação Internet podem conter formulários, que são preenchidos pelo usuário e enviados para o servidor. Formulários constituem o principal mecanismo de captura de informações no lado cliente para posterior processamento no servidor. Cada formulário é composto por diversos campos, como linhas de edição, listas, botões, entre outros. Um campo especial, chamado de campo escondido, pode conter uma informação programada pelo servidor que lhe será remetida quando o formulário for submetido;
- *Parâmetros*: são informações que complementam a requisição de uma página no servidor. Em qualquer requisição, o lado cliente pode enviar um conjunto de parâmetros, que podem ser utilizados pelo servidor na produção da página de resposta. Um parâmetro é definido por um par "nome-valor", ambos tratados como seqüências de caracteres. Os dados preenchidos em um formulário são enviados para o servidor na forma de parâmetros.

Um cliente pode requisitar uma página do servidor através da ativação de uma ligação estática ou pela submissão de formulários. Para tratar as requisições através de ligações estáticas, a aplicação Internet deve incluir a chave criptografada de acesso à memória da sessão como um parâmetro embutido em todas as ligações das páginas que enviar para o cliente. Assim, sempre que uma ligação for ativada no navegador, o parâmetro (i.e., a chave de acesso) será enviado para o servidor.

Para tratar as requisições através de formulários, a aplicação Internet deve criar um campo escondido nos formulários das páginas enviadas para o cliente, contendo a chave criptografada de acesso à memória da sessão.

No primeiro instante em que o servidor identificar a necessidade de guardar memória da sessão do usuário, ele cria uma chave de identificação para a sessão, enviando-a, a partir deste instante, nos formulários e nas ligações estáticas. Quando a informação de um formulário for submetida para o servidor ou quando uma ligação estática for ativada, a chave da memória da sessão também será enviada.

4.5. Dinâmica

A dinâmica de colaboração entre os componentes participantes desta solução é ilustrada pelos diagramas das Figuras 7 e 8. Esta dinâmica pode ser dividida em dois instantes distintos:

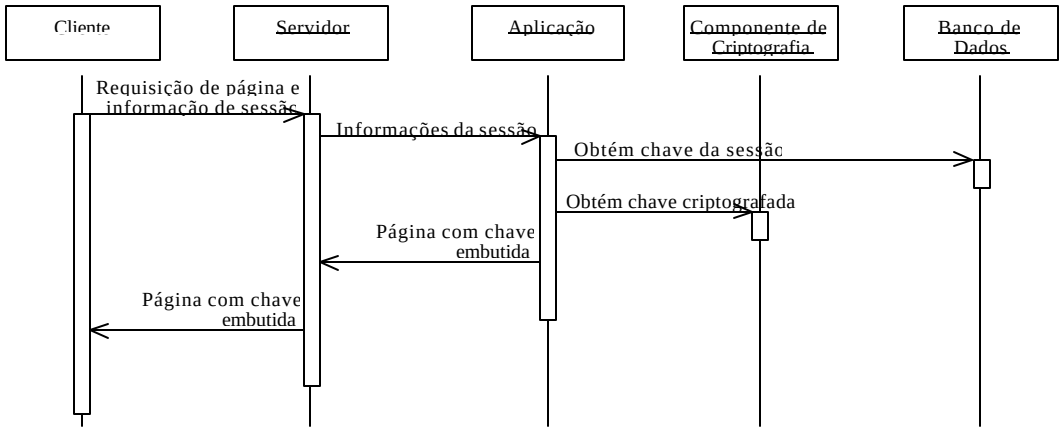


Figura 7 – Diagrama de seqüência para a criação de memória da sessão

a) Cliente realiza uma requisição que faz com que o servidor identifique a necessidade de guardar memória da sessão (Figura 7):
   - Cliente envia uma requisição de página para o servidor contendo informações que deverão ser guardadas para as próximas interações;
   - O servidor identifica a aplicação Internet que deve ser ativada as informações recebidas do cliente para esta aplicação;
   - A aplicação Internet requisita que o banco de dados crie uma nova memória da sessão, retornando sua chave;
   - A aplicação Internet requisita ao componente de criptografia que codifique a chave da memória da sessão, retornando seu valor criptografado;
   - A aplicação Internet cria a página de resultado, adicionando um campo escondido em cada um de seus formulários e um parâmetro embutido em cada uma de suas ligações estáticas para conter a chave criptografada. A página é enviada para o servidor;
   - O servidor envia a página recebida da aplicação Internet para o navegador no lado cliente.

b) Requisições realizadas pelo cliente após a criação da memória da sessão (Figura 8):
   - Cliente envia uma requisição para o servidor, acompanhada dos parâmetros advindos de um formulário ou de uma ligação estática;
   - O servidor identifica a aplicação Internet que deve ser ativada e passa os parâmetros recebidos para esta aplicação;
   - A chave criptografada de acesso à memória da sessão do usuário se encontra entre os parâmetros, sendo seu nome conhecido pela aplicação Internet;
   - A aplicação utiliza o componente de criptografia para recuperar o valor original da chave de acesso à memória da sessão;
   - A aplicação Internet consulta e, eventualmente, atualiza os dados da memória da sessão no banco de dados, utilizando a chave de acesso;

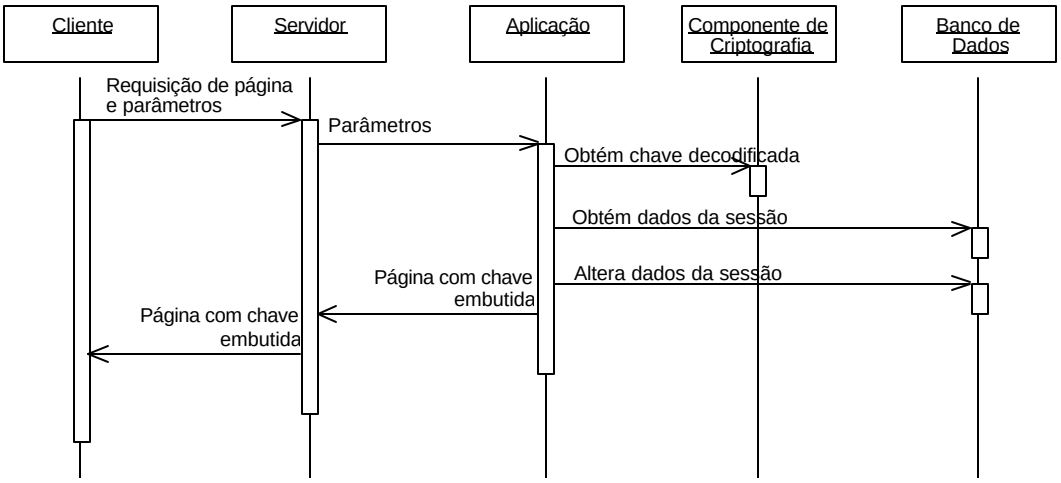


Figura 8 – Diagrama de seqüência para consulta e alteração da memória da sessão

- A aplicação Internet utiliza a memória da sessão para compor a página de resposta. A aplicação inclui a chave criptografada em campos escondidos de seus formulários e em parâmetros embutidos de suas ligações estáticas. A página resultante é enviada para o servidor;
- O servidor envia a página recebida da aplicação Internet para o navegador no lado cliente.

4.6. Conseqüências

A principal vantagem desta solução é a independência do mecanismo de cookies, o que permite que ela funcione mesmo que o usuário desabilite o tratamento de cookies em seu navegador.

A principal limitação desta solução é o crescimento do código das páginas devido aos parâmetros embutidos e campos escondidos. A solução é intrusiva, pois depende de alterações no código das páginas.

4.7. Padrões Relacionados

Este padrão é normalmente aplicado para solucionar o problema de armazenamento e transferência da chave que surge quando da aplicação do *Gerenciamento de Sessão na Internet*. *Gerenciamento de Sessão Baseado em Cookies* armazena a chave de acesso à memória da sessão em cookies no lado cliente.

5. Conclusão

O armazenamento de memória da sessão em uma aplicação Internet não depende somente da natureza das informações armazenadas, mas também do conhecimento das limitações impostas pela plataforma. Os padrões de gerenciamento de sessão documentam este conhecimento, oferecendo soluções para estas limitações.

Três padrões foram apresentados para a implementação de gerenciamento de sessão. O *Gerenciamento de Sessão na Internet* contém a solução genérica para o problema. Os outros dois padrões são utilizados para resolver a questão do armazenamento da chave de acesso à memória da sessão. O *Gerenciamento de Sessão Baseado em Cookies* pode ser aplicado quando o público alvo da aplicação Internet é relativamente controlado e o administrador da aplicação pode garantir que os navegadores de seus usuários aceitem cookies. O

Gerenciamento de Sessão Baseado em Parâmetros assume um cenário mais restrito, onde os limites do servidor são explorados em prol de uma maior flexibilidade do lado cliente.

Agradecimentos

Os autores gostariam de agradecer a CAPES e ao CNPq pelo apoio financeiro a este trabalho, a nossa *shepperd* Rossana Maria pela criteriosa revisão do artigo e ao grupo número 1 do SugarLoafPLOP pelas críticas que permitiram a sua redação final.

Referências Bibliográficas

[Kristol 1997] Kristol, D.M., "HTTP State Management Mechanism", 1998. IETF RFC 2109, 1997.

[Mesz 1997] Meszaros, G.; Doble, J., "A Pattern Language for Pattern Writing", em: Pattern Languages of Program Design 3 (Software Pattern Series), Addison Wesley Longman Inc. – 1997. Disponível em http://hillside.net/patterns/Writing/patterns.html.

[Powell, 2000] Powell, T.A., "HTML: The Complete Reference", 2000. McGraw-Hill Professional Publishing.

[Rubin 1998] Rubin, J.H.; "An in-depth analysis of cookies", 1998. Disponível em: http://headcase.syr.edu/NEW/Research/cookies.html

[W3C 1999] W3C Consortium, "Hypertext Transfer Protocol -- HTTP/1.1", 1999. Disponível em: ftp://ftp.isi.edu/in-notes/rfc2616.txt

Concurrency Manager

Sérgio Soares* and Paulo Borba†
Centro de Informática
Universidade Federal de Pernambuco

Intent

Provide an alternative to method synchronization with the aim of increasing system performance. *Concurrency Manager* uses knowledge about the semantics of the methods in order to block only conflicting execution flows, allowing the non–conflicting ones to execute concurrently.

Motivation

The advent of web–based information systems significantly increased the number of concurrent programs. Concurrent programs must control concurrency to guarantee safe implementations, which avoid interference that lead systems to inconsistent states and behaviors. To implement some of these controls we need to use programming language features, such as blocking methods to avoid their concurrent execution in the same object. In the Java [6] programming language we can do this synchronizing methods with the `synchronized` method modifier, which forbid concurrent execution of methods within an object.

However, implementation of such features brings performance overhead, serializing the execution of some operations. There are several approaches concerned about guaranteeing performance increasing, removing unnecessary synchronization of Java concurrent programs [3, 1, 5]. They show the negative impact in efficiency of the Java concurrency control mechanisms. This negative impact demands alternatives to increase programs' performance.

Method synchronization guarantees that all concurrent execution of a method within an object will be serialized. With this approach we can allow or block all concurrent execution of a method. However, if some execution flows cannot be concurrently executed, but others can, we need an intermediary approach.

Example

Consider an address book application with a class `AddressBook` that has a method to register addresses. The method verifies, before registering an address in the system, if

*Supported by CAPES. Email: scbs@cin.ufpe.br
†Partially supported by CNPq, grant 521994/96–9. Email: phmb@cin.ufpe.br

there is an address with the same email of the address being registered. Two concurrent executions that try to register addresses with the same email may get the same answer: there is not an address with the email of the objects being registered. In this case both execution flows will try to register the objects, which may turn the system to an inconsistent state, or raise an unexpected error. We can conclude that this method cannot be concurrently executed if the objects (addresses) being registered have the same email, otherwise concurrent execution is allowed. Therefore the address' email can be used to decide if an execution flow can or cannot be concurrently executed. Figure 1 shows an UML [4] class diagram of this application.

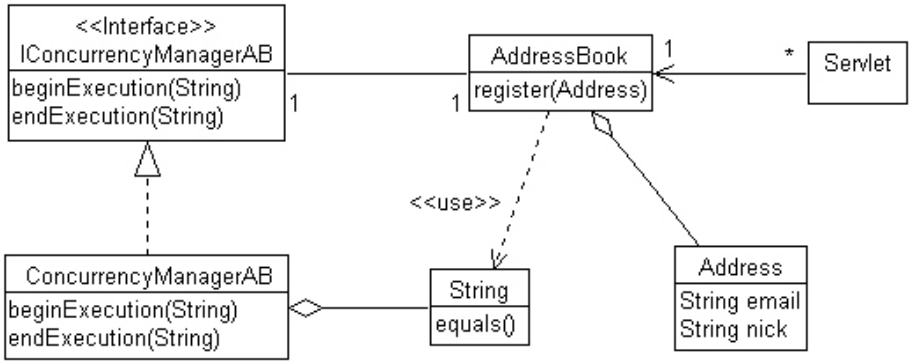


Figure 1: Address Book application's class diagram.

The class `ConcurrencyManager` is user to control the `AddressBook` `register` method execution. The method should ask permission to the `ConcurrencyManager` before executing, calling the `beginExecution` method with the address's email as the argument. If another address is being registered with the same email by other execution flow, this execution is blocked until the executing flow terminates.

The email information is part of the method semantics, as the method's parameters and the object's state. The *Concurrency Manager* pattern uses such information to block only the conflicting execution flows.

Either persistent application that uses databases management systems (DBMS) must make some concurrency controls. So this pattern can be also used in such systems, besides the idea that persistent systems already make all concurrency controls using the DBMS features, which is not true [9].

Applicability

Use *Concurrency Manager* when

- You need to control concurrent access to an object, blocking just some concurrent execution of a method within the object, allowing others. In the previous example concurrent addresses registration can be executed since the addresses have not the same email. Only the registrations of addresses with the same emails must be serialized (synchronized).

- You need to control concurrency in more than one method; some flows can execute concurrently in the methods and others cannot. For example, in a banking appli-

cation you can concurrently execute the methods deposit and withdraw, but for different accounts. The execution of deposit and withdraw for a same account must be serialized to avoid inconsistencies.

- You need to control methods in different objects. The objects must have the same instance of the pattern to manage the concurrent execution in the objects. In this case, one instance of the pattern manages methods execution in several objects.

Structure

The structure of *Concurrency Manager* is presented in the Figure 2 using an UML [4] class diagram. This diagram is a generic diagram, if compared with the diagram presented in Figure 1, which defines a class to encapsulate the information used to decide if an execution flow should be blocked.

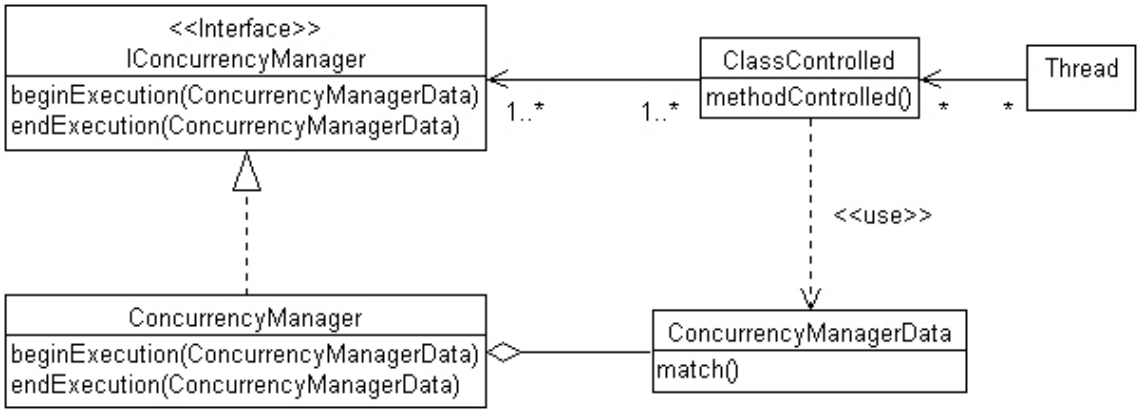


Figure 2: *Concurrency Manager* pattern's class diagram.

Participants

The participants of the pattern are

- `ClassControlled`. A concrete class with methods that can be concurrently executed in some cases and cannot in others. The classes must have one or more instances of the `ConcurrencyManager` to synchronize its methods.

- `Thread`. The thread that executes the controlled method(s) of the *ClassControlled* objects.

- `IConcurrencyManager`. An interface responsible to abstract the pattern implementations and its extensions.

- `ConcurrencyManager`. A concrete class, which implements the `IConcurrencyManager` interface, and is responsible to control the concurrent execution of the `ClassControlled` objects. This control is made based in the operation's semantics, which is encapsulated in the `ConcurrencyManagerData` object.

- `ConcurrencyManagerData`. A concrete class with the information used to forbid a method execution. The class also has a method `match` to compare two instances of the class. This method implementation is responsible to define, based in the information of their attributes, if a `ConcurrencyManagerData` object matches any `ConcurrencyManagerData` object already inserted in the manager, which means that the current execution flow may conflict with another one, and hence must be blocked.

Collaborations

Figure 3 shows a collaboration diagram modeling how a method can use the concurrency manager to control concurrent execution. After being called by an execution flow (message 1), the controlled method creates a `ConcurrencyManagerData` object with the relevant information to decide if an execution flow can execute concurrently (message 1.1). After that, the manager's `beginExecution` method is called with the created data as argument (1.2). Based in the stored data objects, the manager verifies if there is a data object that matches the object passed by the controlled method (1.2.1). If the objects match, the controlled method is blocked (1.2.2); otherwise, the data object is stored in the manager (1.2.3) and the controlled method executes. Just before terminating, the controlled method calls the manager's `endExecution` method with the same data object created in the beginning (1.3). This method call removes the data from the manager (1.3.1) and notifies the blocked execution flows (1.3.2), which become ready to execute again.

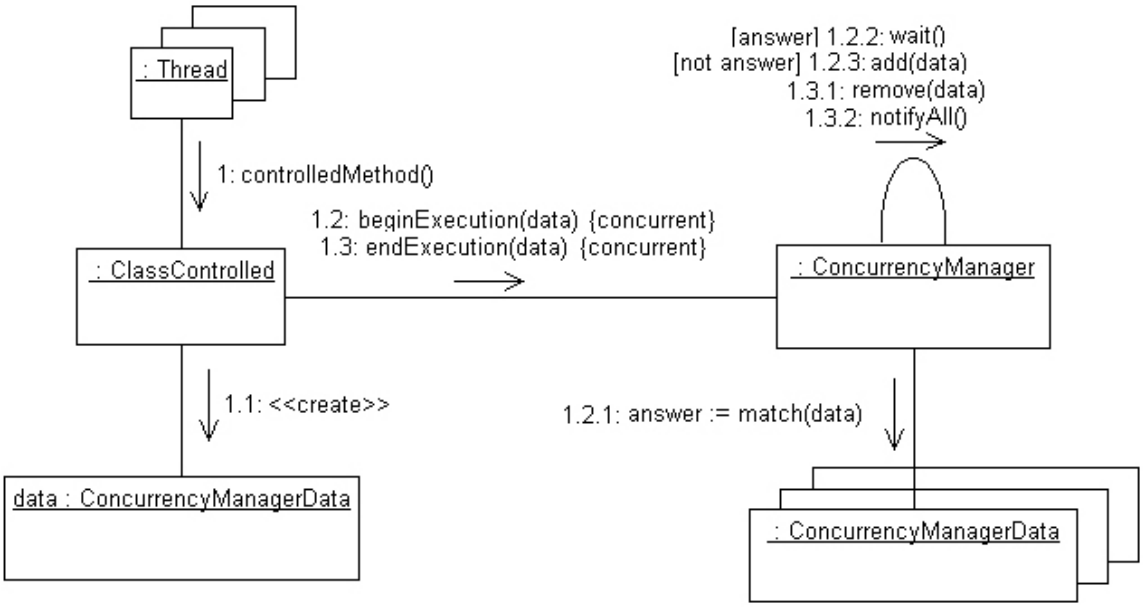


Figure 3: *Concurrency Manager*'s collaboration diagram.

In the collaboration diagram (Figure 3) the constructs { *concurrent* } (messages 1.2 and 1.3) means that in the presence of multiple flows of control, the operation will be treated as atomic. Java supports this construct with the `synchronized` method modifier.

Figure 4 shows a sequence diagrams that specifies an execution of a controlled method without conflicting flows.

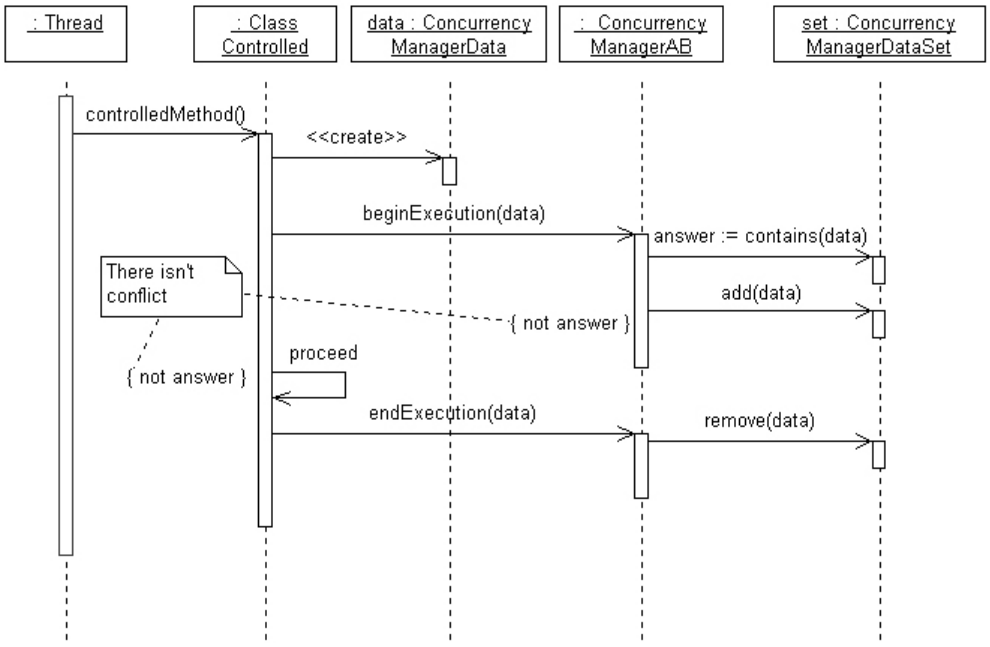


Figure 4: *Concurrency Manager*'s sequence diagram of an execution flow without conflict.

Other scenario is presented in Figure 5, which shows a sequence diagram that specifies an execution of a conflicting execution flow.

Consequences

The benefits of the pattern are:

- *Performance increase.* The system performance is increased by the elimination of unnecessary synchronization. The pattern uses the system operations' semantics to block only conflicting concurrent execution. Performance tests made to analyze the efficiency impact of using this pattern showed that the performance increasing was about 20%, depending of some aspects, such as the operations workload and the number of concurrent threads. With high workload, we can see a low synchronization overhead, on the other hand, with high number of concurrent threads the synchronization overhead will be greater [9]. These aspects variation take the performance increasing in a range from 5% to 60%.

- *Reuse.* The manager class and the class responsible for the data used by the manager can be reused in several systems.

- *Extensibility and maintainability.* The pattern uses an interface to abstract different implementations of the manager. So, we can have different implementations of the pattern, for example, an implementation to be used in sequential environments, which does not make any concurrent control, simplifying its implementation before migrating the system to the concurrent environment. Note that the pattern's structure allows changing the concurrency control without making modifications in business classes. This is possible because the use of the `ConcurrencyManagerData`

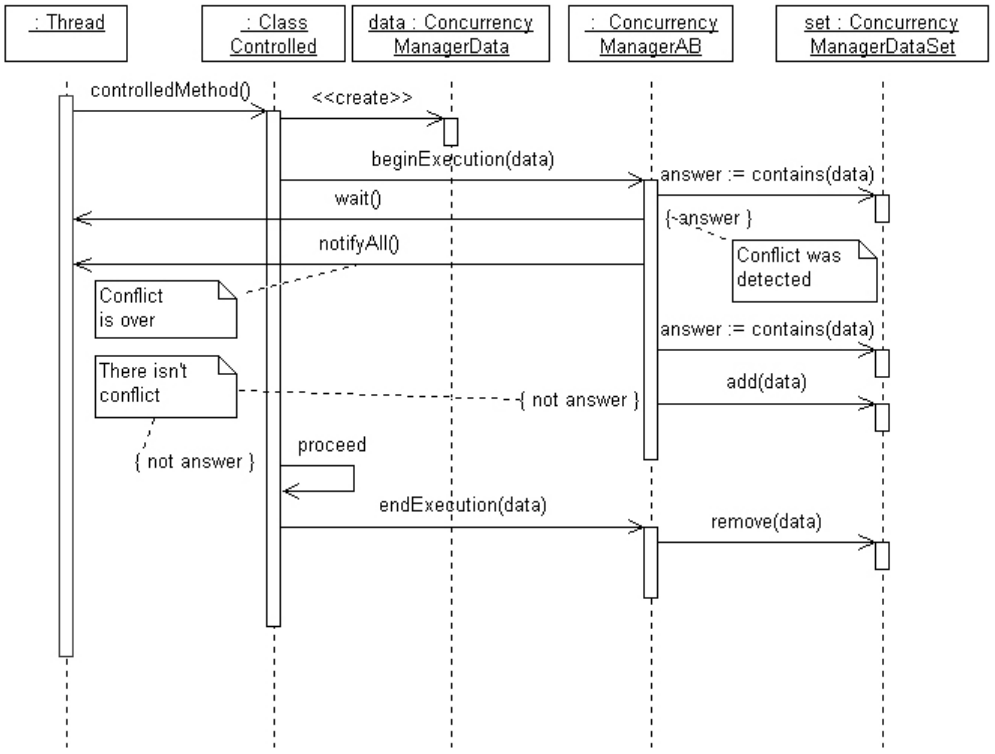


Figure 5: *Concurrency Manager*'s sequence diagram of a conflicting execution flow.

and `ConcurrencyManager` classes remove the code responsible for the concurrent control from the business classes, such as the `Address` class. This separation of concerns [7] (business and concurrency control) helps the system extensibility and maintainability.

The liabilities of the pattern are:

- *Increased number of classes.* The class hierarchy becomes more complex because new classes and interfaces are added, decreasing legibility and maintainability.
- *Increased indirection.* In order to introduce our control technique we must delegate some calls to methods, which seems to decrease system performance. In fact, this lost of efficiency is recompensed because only conflicting execution flows are blocked.
- *Complexity.* The controlled method's code is more complex than using the `synchronized` modifier, because to implement the pattern we must add about four new lines of code. This contributes to decrease the system legibility and maintainability.
- *Risk of deadlock.* When applying the concurrency control technique to a method, the programmer may forget to properly call the manager's `endExecution` method allowing execution flows to block indefinitely.

Implementation

To implement the *Concurrency Manager* we can use several approaches.

- `ConcurrencyManagerData` set. The simplest approach is the one where the manager keeps a set of `ConcurrencyManagerData` objects. When a thread asks permission to execute a method passing a `ConcurrencyManagerData` object, the manager uses the `ConcurrencyManagerData match` method to verify if there is another object that matches this `ConcurrencyManagerData`. If there is not, the `ConcurrencyManagerData` object is inserted in the `ConcurrencyManagerData` set and the execution may proceed. If there is any, the execution flow is blocked until the execution that inserted the object matched by the `ConcurrencyManagerData` finishes. The `ConcurrencyManagerData` class may store a simple key, as a string, or more complex information that is necessary to decide if an execution flow can execute concurrently with others.

- *State machine.* We can also implement a state machine in the `ConcurrencyManager` class. The manager should have a table to store all the system execution. Probably the `ConcurrencyManagerData` class has to store the name of the method to execute and the object id of the object being executed. With this information, the manager can decide if this execution can be done at this time, verifying if this execution sequence is according with the state machine definition.

Sample Code

Consider the application that registers addresses and verifies, before register an address, if there is an object in the system with the same email of the object to be registered, see Figure 1. A possible concurrent execution that tries to register two objects with the same email may turn the system to an inconsistent state. Consider that in a concurrent execution both threads make the email verification and receive a reply that there is not an object with the email of the ones being registered, so, the threads will try to insert the objects. Note that these execution flows cannot execute concurrently, but the flows that try to insert addresses with different emails can be concurrently executed. Therefore, we can implement the *Concurrency Manager* pattern to control these executions.

The following implementation of the *Concurrency Manager* makes a simplification of the pattern. The `ConcurrencyManager` class keeps a keyword set (`String` set) that is used to control concurrent execution over then, instead to keep a `ConcurrencyManagerData` set. To implement this set we use the `java.util.HashSet`, a class that implements an object set without any concurrency control.

```
public class ConcurrencyManager {
    private HashSet keys;
    public ConcurrencyManager() {
        keys = new HashSet();
    }
```

We need to define a method to receive a `String` parameter to inform that a thread will start to execute some operation over this `String`. If this keyword is already in the set the execution is blocked, meaning that another thread is executing over this keyword. The `HashSet` class has a method to verify if there is an object in the set. We use this method to find if the keyword is already in the keyword set. We also use the method

`wait` inherited from `Object` class, superclass of all Java classes. This method blocks an execution until being notified to resume it.

```
    public synchronized void beginExecution(String keyword) {
        try {
            while (!keys.add(keyword)) {
                wait();
            }
        }
        catch(InterruptedException ex) {
            throw new RuntimeException("Unexpected error");
        }
    }
```

We also need a method to inform that the execution over a `String` (keyword) is finished. This method removes the keyword of the set, using the `HashSet remove` method, and releases a thread that is blocked waiting to execute by calling the method `notifyAll`, other method inherited from `Object`.

```
    public synchronized void endExecution(String keyword) {
        try {
            if (!keys.remove(keyword)) {
                throw new RuntimeException("Keyword not found");
            }
        }
        finally {
            notifyAll();
        }
    }
}
```

Now we need programming in the business class the code that negotiates with the manager. The conflicting executions are the ones that try to register objects with a same email. So we use the `ConcurrencyManager` class to synchronize only the conflicting execution, which are the ones registering addresses with the same email. The keyword to be used is the object's email, so if two threads try to register two objects with the same email, the second one's execution is blocked until be released by the first one. The following example shows how the `ConcurrentManager` class is used in this example.

```
public class AddressBook {
    private AddressCollection addresses;
    private ConcurrencyManager manager;
    ...
```

The `AddressBook` class defines an `AddressCollection`, which is responsible to store the `Address` objects, and a `ConcurrencyManager` that is responsible to control the concurrency over the method `register`.

```
       public void register(Address address) throws EmailException {
 1:        String email = address.getEmail();
 2:        try {
 3:            manager.beginExecution(email);
 4:            if (!addresses.hasEmail(email)) {
 5:                addresses.insert(address);
 6:            }
 7:            else {
 8:                throw new EmailException();
 9:            }
10:        }
11:        finally {
12:            manager.endExecution(email);
13:        }
       }
   }
```

As we sad before, this example makes a simplification when do not use a `Concurrency–ManagerData` object to send the method information to the *Concurrency Manager*. In the `register` method of the `AddressBook` class we use the method's semantics, in this case the address' email, to avoid invalid concurrent execution, as described before, calling the `beginExecution` method of the `ConcurrencyManager` class (line `3`). This method call must be done before execute the method in order to ask the manager permission to continue the execution. Therefore, other thread that tries to register another address with the same email will be blocked. Just before terminating the execution we must call the `endExecution` method (line `12`) telling the *Concurrency Manager* to remove the key added in the manager's set and to release the blocked threads, if there are any. This execution is made inside a `finally` clause, which guarantees that the command will be executed, independent of what happen in the method execution, since the method execution may raise an exception before finishes its execution [6]. If this occurs and programmer forgot to call the `endExecution` method inside a `finally` block, the key will not be removed from the manager's set, which allows the executions flows, blocked because of this key, to block indefinitely.

Known Uses

The *Concurrency Manager* pattern uses the idea of semantics–based concurrency control [2]. This approach uses the operations' semantics to improve performance, decreasing operations' serialization. For example, "two operations conflict if they both operate on the same data item and one of them is a write". The concurrency pattern differs from this approach because the programmer must define what is the semantics of conflicting operations, when implementing the concurrency manager data.

A potential use of the pattern to control the concurrency is in web–based systems with a software architecture that has a business collection and a data collection for each basic class. The business collections are classes where the system policies are implemented, for example, the `AdressBook` class used in the previous sections. The data collections are classes responsible for data storage, as the `AddressCollection` class, and basic classes

are classes that model the system's basic objects, for example, the `Address` class. We can mentioned many real web–based systems that use this architecture:

- A system to manage a telecommunication company's clients. The system is able to register mobile telephones and change clients and telephones services configurations. The system can be used over the Internet.
- A system for performing on–line exams. This system has been used to offer different kinds of exams, as simulations based on previous university entry exams, which help students to evaluate their knowledge before the real exams.
- A complex supermarket system. A system responsible to control the sales in a market. This system has been used in several supermarkets and other kinds of stores.
- A system for registering health system complaints. The system allows citizens to complaint about diseases problems and to retrieve information about the public health system, such the location or the specialties of a health unit.

Our approach can be used in the business collection classes of these systems, where business polices may race conditions, as the one in the address book example.

We made performance tests [9] in a toy system that implements the pattern in order to analyze the efficiency impact. As we say in the Consequences Section, the performance increasing goes from 5% to 60%, depending on some aspects such as method workload and the number of concurrent threads.

Related Patterns

A related pattern is the *Monitor Object* [8] that synchronizes the execution of methods. This pattern also allows methods to cooperate scheduling their execution sequences by waiting and notifying each other via monitor conditions. The monitor conditions determine when a method should suspend, and when resume. In the *Monitor Object* approach, the controlled methods has to choose what is the monitor condition to wait or to notify. In the *Concurrency Manager*, the manager is the responsible to say when a method can or cannot execute concurrently. This decision is encapsulated in the manager's definition. In fact, the main *Concurrency Manager*'s goal is to allow as many as possible concurrent execution, to increase the system efficiency, on the other hand, the main *Monitor Object*'s goal is to synchronize objects methods execution. This similarity allows us to classify our pattern as a *Concurrency Pattern* [8], like the *Monitor Object* pattern.

The *Concurrency Manager* pattern may implement the Singleton design pattern to guarantee that there is a single instance of the manager. This is necessary if we try to centralize all concurrency controls in a single manager to control all the system executions.

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 Gunter Mussbacher for the suggestions made at the conference.

References

[1] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 207–222, November 1999.

[2] B. R. Badrinath and Krithi Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, 1992.

[3] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 35–46, November 1999.

[4] Grady Booch, Ivar Jacobson, and James Rumbaugh. *Unified Modeling Language – User's Guide*. Addison–Wesley, 1999.

[5] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM, November 1999.

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

[7] David L. Parnas et al. Using documentation as a software design medium. *The Bell System Technical Journal*, 60(8):1941–1977, October 1981.

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

[9] Sérgio Soares and Paulo Borba. Concurrency Control with Java and Relacional Databases (in portuguese). In *V Brazilian Symposium on Programmig Languages*, pages 252–267, Curitiba, Brazil, 23th–25th May 2001.

Distributed Adapters Pattern: A Design Pattern for Object-Oriented Distributed Applications

Vander Alves* Paulo Borba†
Centro de Informática
Universidade Federal de Pernambuco

1 Introduction

We introduce the Distributed Adapters Pattern (DAP) in the context of remote communication between two components, where it is intended that these components be decoupled from specific communication Application Programming Interfaces (API).

2 Context

In order to accomplish their tasks, components in a distributed system communicate with each other by means of an inter-process communication mechanism. When the components handle communication themselves we obtain applications where the core functionality of its components is interwoven with communication tasks. Therefore, the application becomes dependent on a particular communication mechanism, and its components are hard to reuse and extend.

In order to illustrate the use of DAP, we take a banking example as a concrete context. The banking service stores entities such as account and customer records, and has operations for manipulating these entities, such as `deposit` and `addAccount`. These operations are to be provided remotely to clients of the service, and thus its implementation must rely on a distribution platform. Additionally, it is expected that such implementation follows an incremental method: a non-distributed version is implemented before the distributed one. Another assumption is that it may be desirable to change the distribution platform.

3 Problem

Avoiding tangled communication and business code in order to provide reusability and extensibility.

*Supported in part by CNPq. Electronic mail: `vra@cin.ufpe.br`.

†Supported in part by CNPq, grant 521994/96–9. WWW: `http://www.cin.ufpe.br/~phmb`. Electronic mail: `phmb@cin.ufpe.br`.

4 Forces

DAP balances the following forces:

- Separation of concerns;
- The components should be independent from the communication API;
- The code modification in components to support communication should be minimized;
- Changing the communication mechanism should be a simple task, minimizing the impact on business code;
- Adequate communication performance;
- Development productivity must not be significantly affected.

5 Solution

Introduce a pair of object adapters [4] to achieve decoupling of components in distributed architectures. The adapters basically encapsulate the API that is necessary for allowing the distributed or remote access of Target objects (hereafter *Target object* refers to a business object providing services to other business objects). In this way, Source objects (hereafter *Source object* denotes a business object acting as a client of a Target object) of an application become autonomous with respect to the distribution layer, so that changes in the latter do not impact the former.

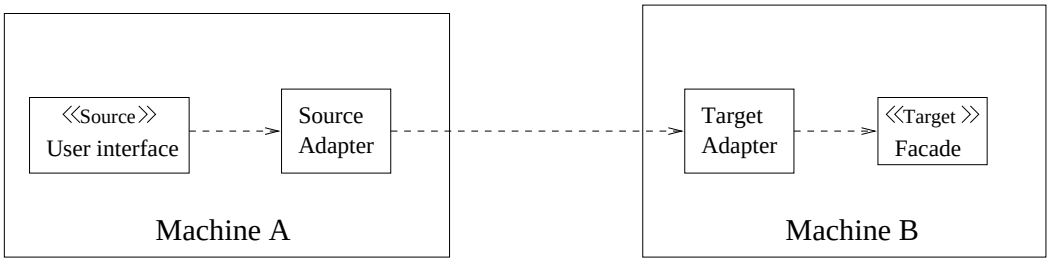


Figure 1: An example of DAP.

There are two kinds of adapters: *source adapters* and *target adapters*. Roughly, the latter wraps Target objects in the places where they are located, and the former represents those objects in remote locations. In a typical interaction, a user interface object (a GUI, for instance) in one machine would request the services of a source adapter located in the same machine. The source adapter would then request the services of a corresponding target adapter residing in a remote machine. Finally, the target adapter would request the services of a Facade [4] object co-located with the target adapter. Figure 1 illustrates this example.

Source and target adapters provide a higher level of abstraction than stub and skeletons do. The adapters isolate user interface and business code from distribution API, whereas

stubs and skeletons isolate user interface and business code from the implementation of distribution issues, but not from distribution API. Source adapters delegate lower level distribution issues such as marshalling to stubs, and target adapters delegate such issues to skeletons.

As another example, the banking application of Section 2 is structured according to DAP as Figure 2 illustrates.

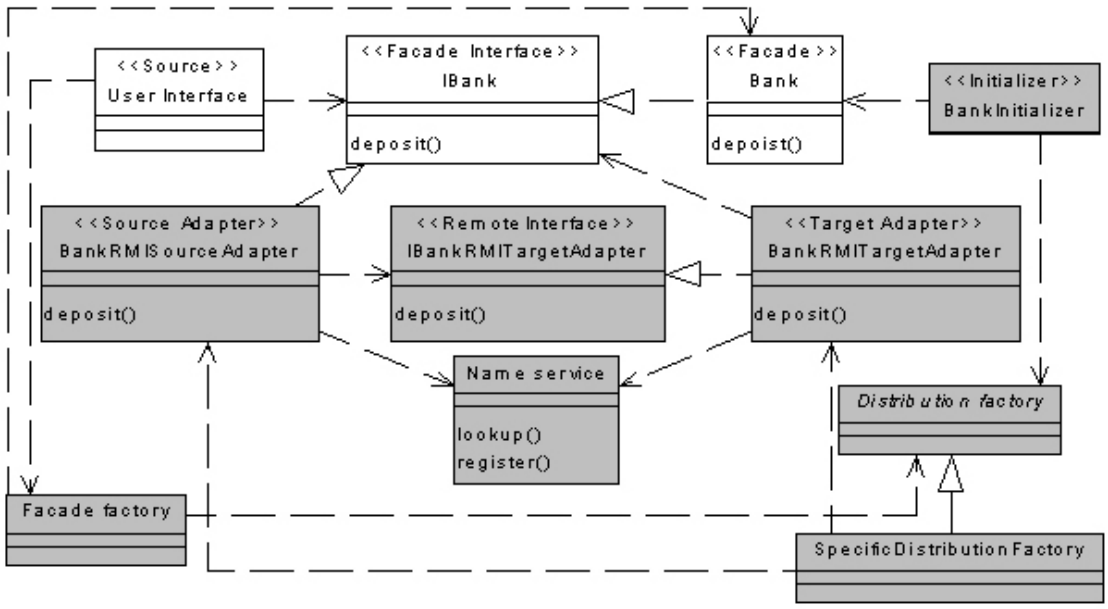


Figure 2: Class diagram of a banking application according to DAP.

The uncolored elements deal with the business aspects of the application, whose Facade is the `Bank` class, which unifies all services of the application. The gray elements denote the adapters and their collaborators. Essentially, these gray elements hide distribution API from user interface and business code. In the following section, each element is described abstractly; their implementation is sketched in Section 8.

5.1 Structure

Figure 3 details the structure of DAP by means of a class diagram. The `Source` and `Facade` classes abstract business components as mentioned previously. The `Facade` class is named after the Facade design pattern [4]. The `Facade Interface` abstracts the behavior of the `Facade` class in a distributed scenario. However, this interface, the `Source` and `Facade` classes have no communication code. These three elements constitute a distribution-independent layer in the pattern. The remaining elements of the pattern deal with this aspect.

The core elements of the pattern handling distribution itself are `Source Adapter` and `Target Adapter`. These are tied to a specific distribution API and encapsulate the communication details. `Source Adapter` is an adapter [4], isolating the `Source` class

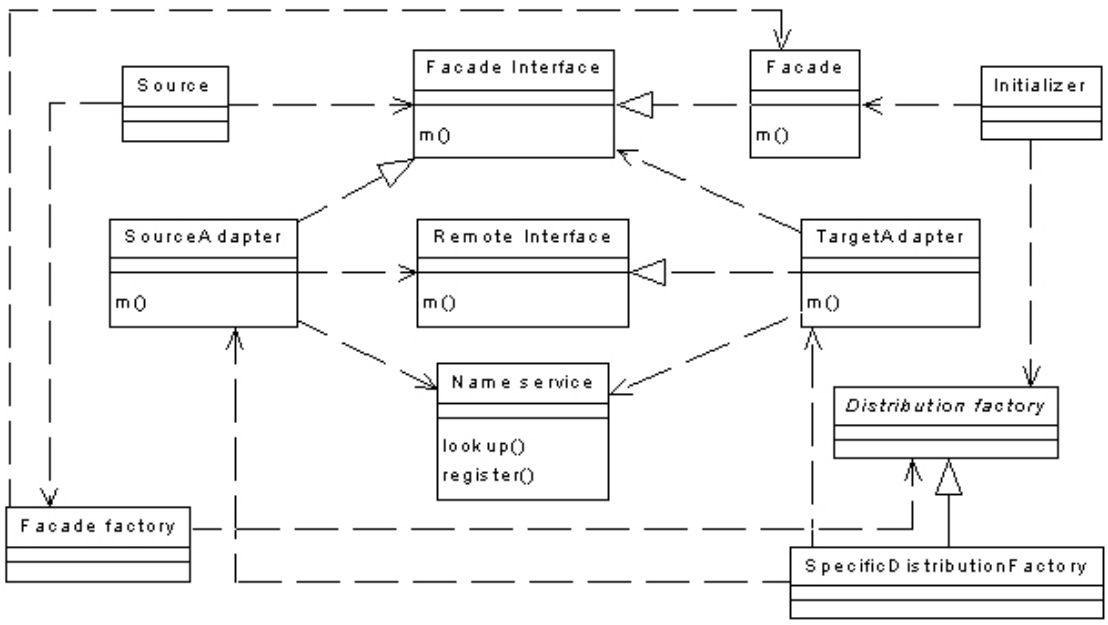


Figure 3: Class diagram of DAP.

from distribution code. It resides on the same machine as the `Source` and also works as proxy [4] to `Target Adapter`. This latter may reside on another machine and is also an adapter, isolating the `Facade` class from distribution code. Since `Source Adapter` and `Target Adapter` usually reside in different machines, and thus do not interact directly, `Target Adapter` implements `Remote Interface`, on which `Source Adapter` depends.

The `Name Service` class has operations for registering and looking up a remote object; both adapters use this class, which represents a generic name service and is common to most distribution platforms. The `Initializer` class also resides in the same machine as `Target Adapter` and `Facade`, and is responsible for creating `Facade` and `Target Adapter` objects. Its importance lies in the fact that it allows the same `Facade` object to be accessed at the same time by different target adapters, representing different distribution technologies. Concurrency control is orthogonal to distribution and can be studied in books such as [5]. The factories in the pattern are useful for configuration purposes: they are used in the creation of `Facade` and of the adapters. In particular, the factories isolate business code from the creation of adapters for a specific distribution platform.

5.2 Dynamics

Figure 4 shows the sequence diagram of a typical scenario for DAP. The `Initializer` creates a `Facade` object and a `Target Adapter`[1], passing to the latter a reference to the former. `Target Adapter` registers itself as a distributed object in the `Name Service` by

[1]Actually, `Initializer` delegates the creation of this adapter to `DistributionFactory`. We omit it here for simplicity.

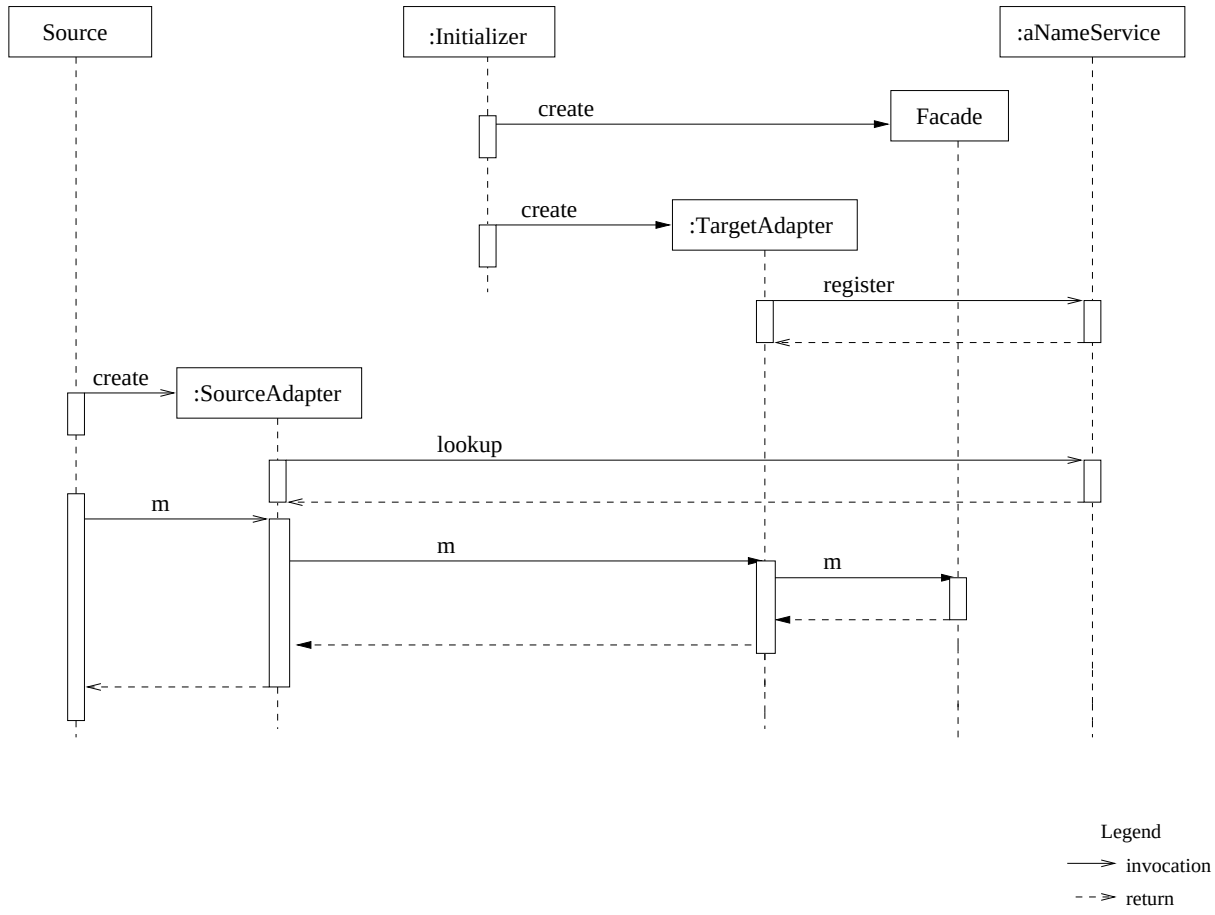


Figure 4: Dynamics of DAP.

invoking its `register` method. During initialization, `Source` creates a `Source Adapter`[2], which performs a `lookup` operation on `Name Service` to obtain a reference to the remote service offered by `Target Adapter`. `Source` then invokes the local `m` operation on `Source Adapter`, which in turn calls the remote `m` operation of `Target Adapter`; this latter delegates the call locally to `Facade`.

6 Consequences

DAP provides the following benefits:

- *Modularity.* This pattern separates concerns by structuring distribution aspects modularly, promoting loose coupling between the different layers of an application's architecture: distribution, business, and user interface layers.

- *Reuse and extensibility.* Due to the modularity provided by the pattern, developers can reuse the Source and Target components easily in other applications based on other APIs and middleware technologies. In addition, changes to the middleware aspects are simpler, since these are restricted to the distribution layer.

[2]In fact, `Source` delegates this to `FacadeFactory`.

- *Incremental implementation.* The pattern supports incremental implementation. During the early phases in development, developers construct a functionally complete prototype, where the Source component (a GUI, for example) depends directly on the Target component (a business Facade, for example). Later, developers add the distribution layer seamlessly, since this latter implements the same interface as the Target component.

This pattern has the following drawbacks:

- *Increased number of classes.* A pair of adapters, three factories, and an initializer are necessary; however, their structure is simple and their generation could be mostly automated by tools.

- *Extra indirection.* The pair of adapters introduces two additional method calls for each remote request. However, both of these additional calls are local, which are much less expensive than the remote one. The work in [1] shows empirical data analyzing the impact on efficiency caused by the adapters; the analysis reveals that such impact is minimum.

7 Implementation

For example, here we consider how to implement the Distributed Adapters Pattern using RMI [9] as the distribution technology. Consider the following implementation issues:

- *Serialization of business objects.* As RMI supports a value parameter passing mechanism for local objects, the classes of these objects must implement the `Serializable` interface [9]. There are no methods in this interface and it simply indicates to the RMI system that an object may be transformed into a stream of bytes in order to be transmitted over a network. However, this is not a negative dependence between the business and the distribution layers since the former calls no method on the latter; in fact, no change on the latter will affect the former.

- *Additional non-functional requirements.* RMI is a simple distribution platform and does not offer fault-tolerance and caching. Such extended behavior can be implemented in DAP's adapters (a detailed implementation is presented in [1]).

8 Sample Code

We now provide sample code for the core elements in the pattern, using the simple banking application mentioned in Section 2 as an example (a full implementation is given in [1]). This application is structured according to DAP as shown by Figure 2. The `Bank` class is a `Facade`, and it keeps references to entities such as account and customer records, and has operations for manipulating these entities:

```
class Bank implements IBank {
   private AccountRecord accounts;
   void deposit(String accountNumber, double value)
```

```
            throws UnknownAccountException {
        accounts.deposit(accountNumber,value);
   } ...
}
```

where `AccountRecord` provides services for manipulating a record of accounts (insertion, updating, querying, deletion, etc.) and also for depositing to or withdrawing from them. The exception `UnknownAccountException` is specific to the banking application. The `IBank` interface implemented by the banking facade is a `Facade Interface`. It abstracts the behavior of the application:

```
interface IBank {
   void deposit(String accountNumber, double value)
          throws CommunicationException,
                 UnknownAccountException;...
}
```

where `CommunicationException` is a general exception representing failure in the distribution layer. This exception does not depend on any particular distribution technology and is defined since the application will eventually become distributed.

A `User interface` object simply creates a `BankRMISourceAdapter` and forwards client requests to it. The RMI source adapter implements `IBank` so that the `User interface` class is unaware of the specific middleware technology. The constructor obtains a reference to the target adapter, by invoking the `connect` method:

```
public class BankRMISourceAdapter implements IBank {
   private IBankRMITargetAdapter  bank;
   public BankRMISourceAdapter(String whereServer)
             throws CommunicationException {
        connect(whereServer);
   }
   public void connect(String server) throws CommunicationException {
      try {
           bank = (IBankRMITargetAdapter) Naming.lookup(server);
      } catch (Exception e) {
          throw new CommunicationException (...);
        }
   }
```

A `User interface` object can call the `connect` method later in case the connection with the target adapter fails (in fact, the source adapter itself may implement fault-tolerant behavior as described in [1]). The `deposit` method forwards `User interface` deposit requests to the target adapter:

```
   public void deposit (String accountNumber, double value)
            throws CommunicationException,
                   UnknownAccountException {
      try {
          bank.deposit(accountNumber,value);
```

```
      } catch (RemoteException e) {
           throw new CommunicationException (...);
        }
   }
} //end of BankRMISourceAdapter
```

Note that, both in the constructor and in the `deposit` method, the source adapter replaces an RMI specific exception with the general `CommunicationException`. The `IBankRMITargetAdapter` interface is the type of the reference to the target adapter and its methods must also raise `RemoteException`:

```
public interface IBankRMITargetAdapter extends Remote {
   void deposit(String accountNumber, double value)
      throws CommunicationException, UnknownAccountException,
             RemoteException;
}
```

where `Remote` is an RMI interface used to identify remote object types [9].

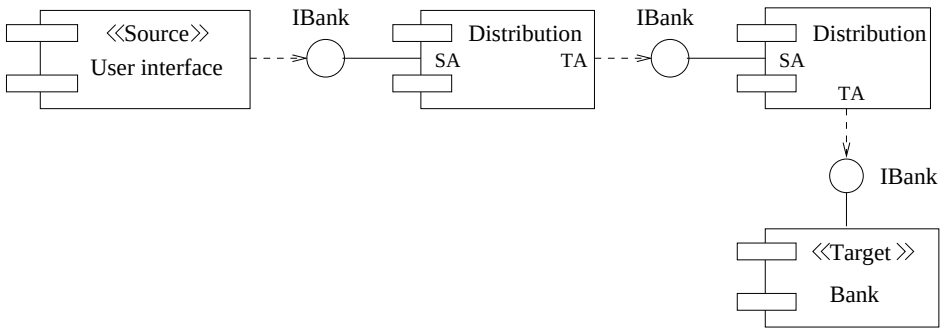
The target adapter becomes an RMI remote object by inheriting from the `UnicastRemoteObject` [9]. It implements the `IBankRMITargetAdapter` remote interface, so that the source adapter can call its methods remotely. The constructor of the target adapter receives a facade object as an argument and registers the adapter itself in the name service:

```
public class BankRMITargetAdapter extends UnicastRemoteObject
          implements IBankRMITargetAdapter {
   private IBank bank;
   public BankRMITargetAdapter(IBank bank)
              throws  CommunicationException {
      try {
           this.bank = bank;
          Naming.rebind("BankServer", this);
      } catch (Exception e){ throw new CommunicationException(...);}
   }
```

The source adapter invokes the `deposit` method on the target adapter, and this operation forwards the call to the corresponding method in the facade object:

```
   public void deposit(String accountNumber, double value)
            throws CommunicationException, RemoteException,
                   UnknownAccountException {
      bank.deposit(accountNumber, value);
   }
} // end of BankRMITargetAdapter
```

Note that the type of the target adapter's attribute is `IBank` and not `Bank`. The rationale is that, since either a facade or a source adapter implements `IBank`, the target adapter, which depends on this interface, may refer either to a facade or to another



This figure illustrates an application with two levels of distribution. Each distribution component abstracts both adapters. SA and TA denote Source Adapter and Target Adapter, respectively.

Figure 5: Additional levels of distribution.

source adapter. This latter case accounts for flexible configurations where there are additional levels of distribution. Figure 5 illustrates this situation. As mentioned previously, the methods of the `IBank` business facade interface declare `CommunicationException`. Therefore, methods in the target adapter and in its remote interface must also declare this exception.

9 Known Uses

DAP has been used in the implementation of a Web based information system, where the adapters are used between the web server, in which servlets [10] act as clients of the source adapter, and the application server, in which the target adapter interacts with the facade. The facade is not in the web server due to security and performance reasons.

Another use of DAP in Web based information systems employs the adapters between an applet, in a client Web browser, and a facade, in a remote machine. The adapters hide the communication details, which use HTTP [12], from the applet and the facade.

The work in [8] and [2] reveals that developers have been using patterns that have some relation to DAP. In particular, the pattern in the first work is similar to DAP's source adapter; the pattern in the second work is similar to the DAP's target adapter.

10 Related Patterns

- *Distributed Proxy Pattern [6].* This pattern and DAP have similar objectives. However, following to [11], DAP does not attempt to make the incorporation of distribution totally transparent. Indeed, a client of a source adapter in DAP must be prepared to handle the general `CommunicationException`. DAP makes transparent the use of a particular distribution technology, not distribution itself. In order to achieve it, DAP uses adapters (instead of proxies), which replace specific distribution code by general code, for example by turning `java.rmi.RemoteException` into

`CommunicationException`. Moreover, the adapters in DAP may implement additional non-functional requirements, such as fault-tolerance and caching, and may also be used to achieve `n` levels of distribution (as shown in Figure 5), each of which may be implemented by a different technology.

- *Wrapper-Facade [7]* and DAP have the common 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 DAP encapsulates object-oriented distribution APIs, such as RMI and CORBA.

- *Adapter, Facade, and Abstract Factory.* DAP is implemented using the Adapter, the Facade, and the Abstract Factory design patterns [4].

- *Broker and Trader.* Well known patterns for structuring distributed systems already exist. The Broker [3] and Trader [3] 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.

- *Chain of Responsibility [4]* is similar to DAP in the sense that it decouples the sender of a request from its receiver by giving more than one object the chance to handle the request. This indirection is similar to the DAP's adapters; these, however, also perform interface filtering, isolating the distribution platform's API, which is not done by Chain of Responsability.

- *Model-View-Controller (MVC) [3]* 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. DAP is used in the context of distributed applications and aims at making changes to the distribution platform a simple task.

Acknowledgements

We would like to thank our shepherd, Eduardo Fernández, for all the work he put into commenting on this paper and the great suggestions for improvement he made. During the writer's workshop at SugarLoafPLOP'2001, Jorge Ortega Arjona, Gunter Mussbacher and Sérgio Soares have also made several interesting comments that helped to improve this paper.

References

[1] Vander Alves. Progressive development of distributed object-oriented applications. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, Feb. 2001.

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

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

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

[5] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1999.

[6] Antonio RitoSilva, 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.

[7] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern Oriented Software Architecture*, volume 2. John Wiley & Sons, 2000.

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

[9] Sun Microsystems. *Java Remote Method Invocation Specification*, 1.50 edition, October 1998.

[10] Sun Microsystems. Java Servlet Specification, Abril 2000.

[11] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems, November 1994.

[12] The World Wide Web Consortium. *Hypertext Transfer Protocol Specification*, 1.1 edition, jun. 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html.