Rule Object 2001: A Pattern Language for Adaptive and Scalable Business Rule Construction

Ali Arsanjani
IBM National EAD Center of Competency, Raleigh, NC, USA
Maharishi University of Management, Fairfield, Iowa, USA
Arsanjan@us.ibm.com

Abstract

Rules are changing everyday in the face of rapidly volatile business requirements. How do we handle this change while keeping our systems efficiently maintainable, reusable and extensible? How do we model and handle (represent) rules, for greater reuse, maintainability, performance?

Business rules tend to change more frequently than the rest of the business object with which they are associated. These rules are typically implemented within the rule methods of a business object. Rules also refer to other business objects that their encompassing business object associates with; creating a web of implicit and increasingly unmaintainable dependencies. Thus, changing a business rule can impact the set of objects dependent upon that rule. Entropy increases even more when the code that is implementing a rule is scattered across several methods within a class, or across several methods of collaborating classes. This lack of centralization leads to ripple effects; the impact of changing a rule's constituent if-else statements leads to side-effects.

The Rule Object Pattern Language contains twenty-two patterns that cover a spectrum of solutions to common problems encountered in the realm of modeling, design and implementation (three levels) of handling business rules. For brevity only seven of these have been included in this paper.. We have however, chosen to focus our discussion around what we consider to be the more salient pattern, the Rule Object, and discuss the other patterns such as Rule Method, Simple, Composite and Compound Rule Object, Assessor, Action in light of it. This pattern language balances the forces in the problem domain of modeling, design, implementation and placement of business rules, workflow, routing in distribnuted, object-oriented and component-based applications. As the need arises for more scalability, flexibility and performance, additional patterns are introduced that balance the new forces that come into the picture as a result of additional functional or non-functional requirements. The Rule Object pattern language can best be seen to resolve forces in the greater context of the Service Provider domain pattern [Arsanjani99a;b].

Prelude

This pattern language continues to grow and expand to address new problems and issues that have been encountered in over a year since it was first written. Thus, each project has added its own flavor of forces and problems that have been abstracted and mined out of similar project engagements. Unlike many individual patterns, a pattern language provides a holistic solution to set of recurring problems and thus needs to be maintained and refactored ("weeded out") on a periodic (e.g., yearly) basis. The current effort is the result of such refactoring and weeding out. Also, the patterns in the language have been augmented to handle a wider variety of situations and resolve new forces resulting in significantly new contributions.

Introduction: Rule Design

Rules are encountered in most portions of software systems. They are of various types, scopes and scales. They are often changed, and changed often, to accommodate new requirements. Various authors have provided taxonomies of rules, particularly business rules [Odell96]. Rules also tend to evolve along with the rapid pace of business change and evolution.

Let us define a Simple Rule, to be of the form: "*if* <condition> *then* <action> *else* <action>". This general type can express most of business rules but not all business constraints which are better expressed with such this as EBNF, pre-post conditions and object constraint language. But no matter how they are expressed at the business/specification level, long chains of nested if-else statements in various parts of code that implements a class tend to clutter the code and render it virtually un-maintainable as repeated changes are made over time. In order to avoid ripple effects, based on the principle of variation-oriented design [GHJV95], [Arsanjani99;b], we encapsulate that which tends to change frequently and in a non-uniform fashion. We therefore need to extricate these chains of *if*-statements and *case*-statements from within the application layer code (e.g., business logic); to enable rapid alterations to rule structures as dictated by business needs.

This allows us to insulate the system, to a large degree, against ripple effects that result from changes to individual objects and sets of collaborating objects, collectively fulfilling a business objective (clusters).

Motivation for the Pattern Language

In modeling and implementing rules in component-based and object-oriented systems, there is no "one solution" to business rules: *the solution will typically cover a spectrum* of solutions that are designed to resolve a set of forces that occur in disparate locations within an application: middle-tier versus database, communication protocols, routing to back-end legacy systems, building enterprise scale components with externalized business rules, etc. When a project has a large set of business rules (greater than 7+- 2 per use-case[1]) then it is recommended to have a Rule Analysis phase in which identifies the Rule Categories, their dependencies and complexity, non-functional requirements. This helps in determining Rule Placement as pin-point which solution(s)-- within the spectrum of solutions-- should be applied at a given tier or component location within the software architecture. One of the main contributions of this paper is to present that spectrum of solutions.

Thus, the answer to "should we use a rule engine for this application?" may not necessarily have a straightforward "yes/no" answer; if we want an extensible and robust architecture and not box ourselves into a corner. The decision may depend on further factors that introduce new forces into the picture. As new forces are introduced and decisions are made to deal with forces (one way or another), the solution starts to evolve out of a spectrum of possibilities and can be pin-pointed. It grows clearer where in the spectrum we will be resolving or balancing [some of] the forces. Thus, it is important to identify which point in the rule spectrum you are targeting your solution.

Here is the Rule Pattern Language spectrum of solutions:

1. **Scattered Conditions and Actions (scattered and disparate bits of business logic)**
2. **Rule Method (isEligibleForLoan())**
3. **Rule Object**
4. **Rule Object + Condition = Assessor**
5. **Rule Object + Action = Action**
6. **Simple Rule = Rule Object + Condition + Action**
7. **Compound Rule = Composite Rule Object + Composite Condition + Composite Action**
8. Configurable Profile
9. Externalize Rules (XML)
10. Hash and Cache
11. Configurable Workflow
12. Persistent Rules
13. Rule Types = Type Object + Association Object + Rule Objects
14. Rule Strategy
15. Conflict Resolution
16. Rule Engine
17. Generate Rules (Code Generation)
18. Rule Server (Load balanced multi-threaded, optimized)
19. Rule Engine
20. Contract
21. Rule Language

[1] Personal heuristic of the author, based on multiple project experience, nothing more.

22. Components have Manners
23. Pluggable Manners = Component + Type Object + Rule Objects
24. Rule Analysis

In this paper we will specifically deal with patterns 1-7 of the above spectrum for brevity.

One of the basic tents of the Rule Pattern Language is that Rules are not merely a technical issue, they are foremost a business issue that must be closely managed and organized. For an overview of the business and managerial/organizational aspects of business rules refer to Appendix C. The remainder of the paper assumes the upfront work has been done and rules have been organized in a repository or are at least readily accessible and explicit within the business domain you are investigating/developing software . For example, we start with simple rules, identify and manage them in a repository, discover Rule Types and conduct Rule Analysis (dependency and complexity analysis, deciding which clusters require which of the Rule Pattern Language solutions), Partition the Rules to appropriate rule implementation technologies (Rule Object Frameworks and Rule Engines-- not all rules should go in an engine, not all rules should be hand-coded. Consider using a code generation tool to jump-start your prototype; beware of the trap of falling into the generated tool trap: you have to regenerate every time for a simple change. Consider using a configurable profile along with your (for example) Pricing or Rating Component.

The combination of approaches is akin to keeping some data on the drop downs of a GUI using JavaScript, some reference data in a second tier persistent cache; some data in a cache that is refreshed periodically and some in a back-end database (e.g., Entity Beans). These are not different diverging solutions: they are all along a spectrum that points in the same direction: scalability, performance, reliability and consistency (we have one approach).

The same is true of the approach outlined above for handling the design and implementation of business rules.

Therefore, conduct Rule Analysis and Externalize Rules based on a decision to use a solution in the Rule Spectrum for a given Rule Placement.

For example, placement of rules in the GUI, middle-tier and back-end database requires different rule designs.

Scattered Rules

Rules are usually implemented within the body of various methods that an object implements. These typically scattered and often-nested *If-else* statements tend to clutter code and render it unintelligible; thus making it difficult to identify, change and redeploy components within an application.

In this situation, the rules have been coded rather haphazardly, wherever needed without regard for maintainability or for reuse. Changing one condition or action can cause side effects and maintenance is compromised. Updates to frequently changing rules are difficult and error-prone making the overall life-cycle of introducing new business requirements, finding where to change the code and where to add new code to reflect the new requirements unacceptably time-consuming and extremely difficult.

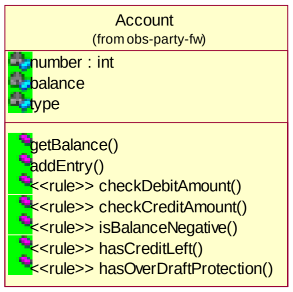
Rule Method Pattern: “Rules as Methods”

Therefore, for each of the rules you have identified (and those you haven’t and run into within the code!) and can find in the code (sometimes using automated tools and sometimes “eye-balling the code”) separate that code out into its own Rule Method. Thus, to begin to properly implement them, Business Rules that are first identified will be listed, categorized and then implemented by a single method. This typically returns a Boolean value to acknowledge whether the rule applied or not. If the rule is complex enough, this *rule method* may call other rule methods and try to enforce the rule through collaboration with the rules of other business objects. So, for example, in our property insurance case study we may have something called a Debris Removal Coverage. This is an additional coverage that can be used if:

- building coverage is part of the policy, and
- inflation guard is not part of the policy, and
- a SpecialProvision for business service has not been applied, and DebrisRemoval's limit has been increased.

As a first step, find what the rules are by working with the business and refactor the existing scattered rules within the code to find and isolate individual rules, conditions and actions. Put each of them in their own method inside the business object that the rule belongs too (Business Objects have Rules) and is most relevant to the business object or component (fine-grained or large-grained). Sometimes you will have to invent a Mediator and assign the rules to it, because some rules will remain unassigned to any of the existing business objects. In this case, we have what is called a Reified Collaboration.

Use "is", "has",etc., prefixes for the Rule Methods. You may want to prefix them with a <<rule>> stereotype in your diagrams. You may end up with the following structure:

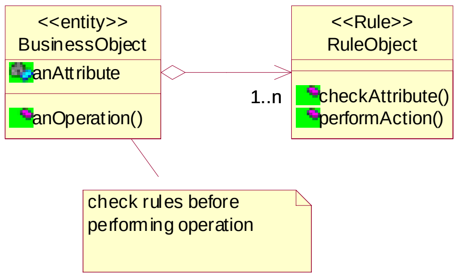

Account
(from obs-party-fw)
number : int
balance
type
getBalance()
addEntry()
<<rule>> checkDebitAmount()
<<rule>> checkCreditAmount()
<<rule>> isBalanceNegative()
<<rule>> hasCreditLeft()
<<rule>> hasOverDraftProtection()

For example,

```
public boolean hasConflictWithSpecialProvision(Policy aPolicy)
{
        SpecialProvision sp = aPolicy.getSpecialProvision();
        if (sp == null)
                return false;
        else return (sp.hasBusinessServiceProvision() ||  sp.hasChurchProvision());
}
```

Sequentially Going from a Simple Rule Object to a Compound Rule Object

You have now separated the rules into Rule Methods. But will this solution scale? Frequent changes to the business rules need to be dealt with by isolating the changing element from the non-changing element (first principle of variation-oriented design [Arsanjani2001a]). How can you design a business rule component that you can use in a simple case and yet start scaling your design?



Therefore, Make the design of business logic extensible and adaptable, without endangering them with intrusive changes, by reifying the rules and making them pluggable. Reify the changing part of the business component into a Rule Object. This has a structure that is best understood through a sequential unfolding of complexity that attempts to balance forces that require additional classes to help resolve additional forces that come into play when new requirements are incumbent upon the Rule Object.

Make the rules governing an object's behavior extensible and adaptable through transparently attaching rule objects, each one representing a rule the object has to enforce in its domain. The object manages its rule object(s) dynamically. By representing rules as individual objects, different business process flows are kept separate from the rules governing them and their interactions. Thus the process of changing them is simplified.

These unfolding layers of layers of complexity and functionality with their corresponding graduated set of solutions are presented below in the section entitled Rule Object Pattern. Here, we deal with structure of a precursor to the Simple Rule Object: Rule Object starts out as a simple Validator. A Validator is a Rule object that uses methods for conditions and actions. The Validator has a method which accesses each of its condition methods, and if they are all evaluated to true, calls one or more action methods.

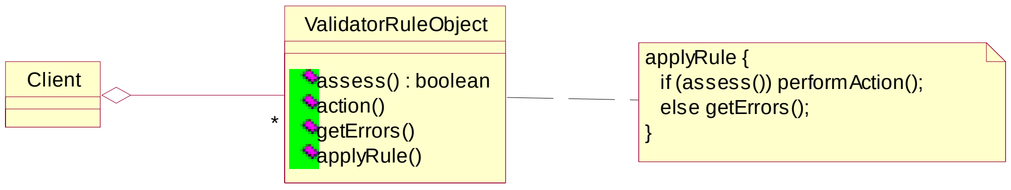


Figure 1: Validator

The next step is a Simple Rule Object. As we add complexity to the forces involved we require additional classes to resolve the corresponding forces and end up with Composite and Compound Rule Objects. All of these mutations are covered in the Rule Object Pattern below.

Rule Object

Intent

Make the design and implementation of computerized business processes extensible and adaptable, without endangering them with intrusive changes, by externalizing the rules governing them and making them pluggable.

Motivation: An Insurance Example

You are designing a Property Insurance Application. The Insurance company has a number of client s(the "insured") who have purchased a number of policies. Each policy has a number of Coverages associated with it by which the Insured will protect their buildings against some Cause of Loss (COL). Buildings are located on a given premise within a geographic region. Coverage is specialized in four types: Building Coverage, Personal Property Coverage, Debris Removal and Special Provision. Each of these subclasses have a set number of COLs associated with them.

At the heart of the requirements, are the business rules. You will find countless rules governing a business application such as this. These are often scattered across tiers: GUIs, middle-tier business logic supporting business processes and database tiers may each have their own set of rules.

Let's consider a few business tier rules that do not require a lengthy explanation of the domain:

- There can be no duplicate coverage.
- In order to add a Personal Property Coverage to a Policy, there must be an existing Building Coverage.
- Or take InflationGuard; this is an additional coverage that can be used if a Building Coverage is part of the Policy and the Policy does not already contain a Debris Removal Coverage.
- Policies should not have Coverage with overlapping dates.

Policies, Coverage and rules associated with individual Coverage are often changing. The business needs to change rules to ensure market share and profit; not to mention survival by prudently changing business processes to accommodate rapidly changing business needs and competition. In order to rapidly change the design of business requirements, analysis invariants (collectively, business rules) , we need to locate and modify rules without suffering side-effects. We may typically want to restore the previous conditions of the rule at a later time, for example when a promotional offering expires. Thus the requirement for variation and change, entails the requirement for pluggability and adaptability. But firstly, the rules must be well organized in order to do this: if rules are scattered throughout the application, changing them without side-effects can be a cumbersome manual endeavor, increasing project time and cost. In some cases this manual process is actually infeasible. Business process adaptability is thus compromised.

Instead of enforcing the rules using a set of hard-coded if-statements that are hard to identify, change and deploy, we may begin by concentrating the rule in Rule Methods, each handling one rule. As the need to make changes (variability) increases, the Rule Method is refactored into a Rule Object with methods for conditions and actions. As these conditions and actions tend to increase in number and variability (new conditions have to added and deleted, updated, new actions are sought to be implemented) we separate these out into their own classes that belong to the Rule Object. If we want to avoid doing the same thing over and over again on multiple projects (all projects have business rules) we set up a tiny framework which will allow us to do rule checking by plugging-in conditions and actions when the business feeds us with new requirements[2].

Rules, however, seldom change in their entirety; usually, new conditions are added to old conditions[3], new or modified actions need to be replaced or recombined in new combinations. Therefore, the requirement to componentize rules for pluggability emerges. We can start by encapsulating their constituent elements:

[2] See the implementation section for snippets of this framework code; contact the author for a complete source code listing.
[3] This is a potential conflict area, where a force emerges that is not necessarily resolved. For further detail see the CommonRules Framework by Ben Grosof for rule conflict resolution in general.

conditions, actions, properties passed to them (context), result objects along with the rule itself as an adaptable component.

This furnishes us with the ability to rapidly change, extend, recombine and reuse rules and rule constituents and components. For example, it would be convenient to be able to plug in some conditions to be checked for a new type of Property Insurance, say Personal Property. If the new conditions apply, we would like to perform some action, even something as simple as displaying a warning message to inform the user that they have entered overlapping Coverages that cannot be inserted into the same policy.
Now you find yourself having to define many/ complex methods for assessing (evaluating) conditions; and these conditions may start changing; so you would like to encapsulate the methods so you can change them easily. The same discussion applies to Actions. You therefore decide to reify the conditions as an Assessor(s) and the actions as an Action(s) Classes, respectively. The Rule Object then acts as a Mediator ([Gof]) between the Assessors and the Actions. It may use a Factory (Builder or Abstract Factory or plain Factory Method) to create appropriate Assessors and Actions from Serializer Objects (see [Riehle98]). Other patterns in the Rule Pattern Language provide solutions to “Hash and Cache”; “Wrap and Map” the Rule Object or its Constituent Assessors and Actions so they may be created or plugged into the Rule Object on-demand. Now you have the following design. Please note that SimpleRuleObject is an abstract class, Assessor and Action can be interfaces or abstract classes with some default behavior.

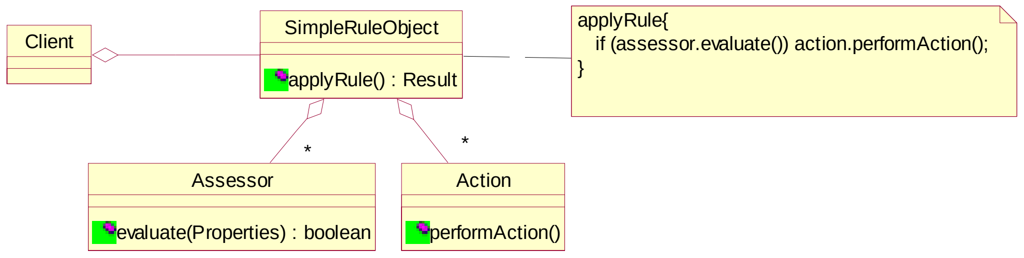


Figure 2: Simple Rule Object with Assessors and Actions

The third level of complexity/scalablity may result from having to constantly change the fields or properties which undergo assessment. It is convenient to use Beck’s Variable State or Yoder and Foote’s Property List patterns to create a set of name-value pairs. These are then handed to the Assessor for evaluation. The Assessor’s code is thus less tangled with references to individual (view-level) instances and has references to a model-level set of name-value pairs that reflect the set of fields or properties that will be supplied as input into the Rule Object’s Assessor. You may also find that your Action(s) need to update the fields in some way: expand an abbreviation for a state to a full name, take a zip code and supply the state, etc. Another step of complexity might lead to the following design:

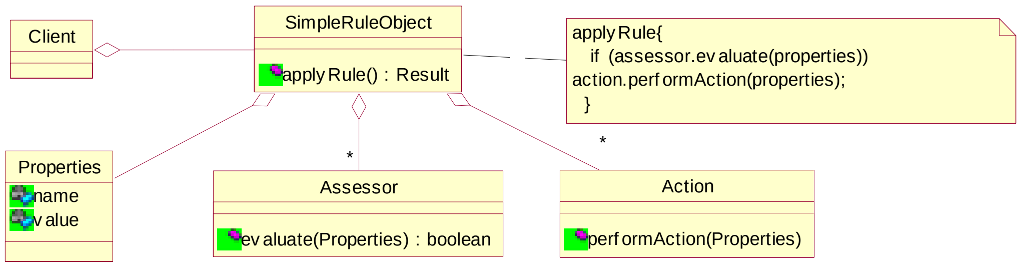

Client
SimpleRuleObject
apply Rule() : Result
apply Rule{
if (assessor.ev aluate(properties))
action.perf ormAction(properties);
}
Properties
name
v alue
*
*
Assessor
ev aluate(Properties) : boolean
Action
perf ormAction(Properties)

Figure 3: Simple Rule Object with Properties
The Fourth Step would be when you need to log and return the outcome of processing through intelligent messages. Thus, when a Rule Object Assesses a set of Properties, the Result may need to be logged as an Assessment Result. For example:

```
if( assessor.evaluate("BillPayMethod") )
       action.performAction();
else assessmentResult.addErrorFor("BillPayMethod");
```

Users are more interested to know *what* went wrong in their data entry, so they can fix it. There are many more reasons for recording and reporting results. These include debugging (programmers) , performance (architects), data-mining (marketing, business) , problem-resolution (users), etc.

Here is a simple adornment to the Simple Rule Object:

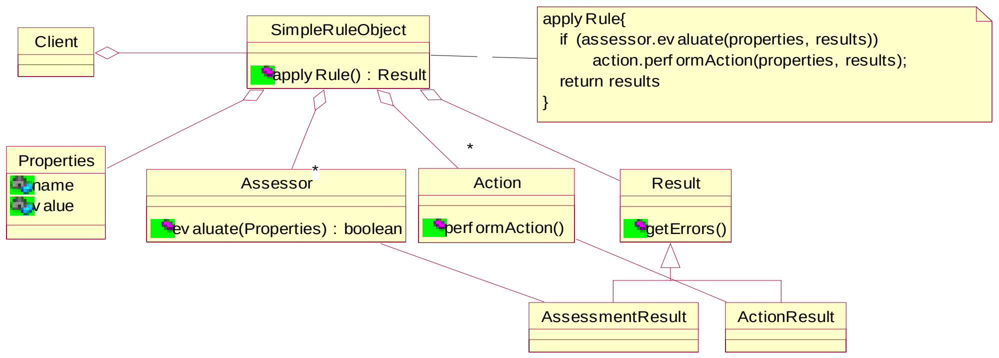


Figure 4: Rule Object with Result Hierarchy

Variations in implementation of this pattern include the Rule Object having a reference to AssessmentResults and ActionResults, rather than the Assessor or Action.

Lets call the above cluster of classes, the Simple Rule Object Cluster and show it like this:

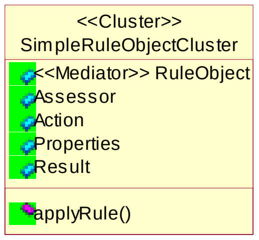


Figure 5: The Simple Rule Object Cluster

This signifies that the cluster acts as a façade with method applyRule(); and has constituent elements or collaborators Rule Object, Assessor, Action, Properties and Result. The cluster may be implemented as a component that can reflectively ask participants in the cluster for their public methods.

Compound Rule Object

So far we have been exploring the four levels of complexity comprising the structure of a Simple Rule Object. In more complex cases, when an organization seeks to store its rules and gain access to a set of reusable conditions (assessors) and actions, the Composite design pattern [GoF95] can be used to design such a Compound Rule Object structure:

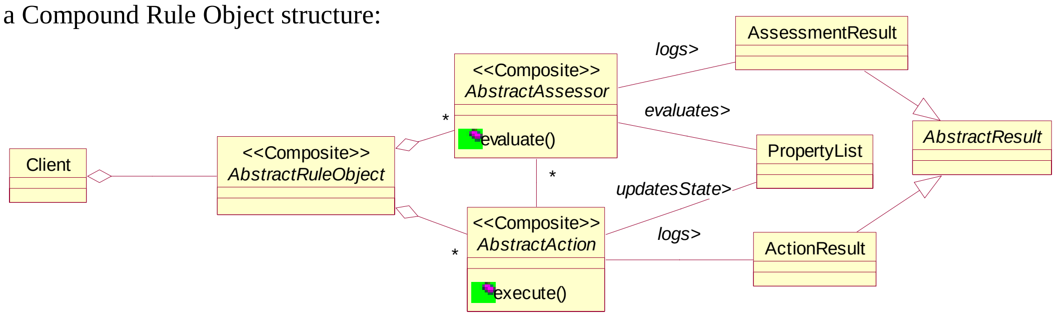


Figure 6 : Compound Rule Object, View 1

The more expanded view would look like this: The CompoundRuleObjectCluster would contain a Composite Abstract Rule Object whose leaf node would be the Simple Rule Object already discussed and its components may include nested aggregate structures of Rule Objects with potentially Composite Assessors and Actions. One of the uses of this more complex scheme was used to describe and drive

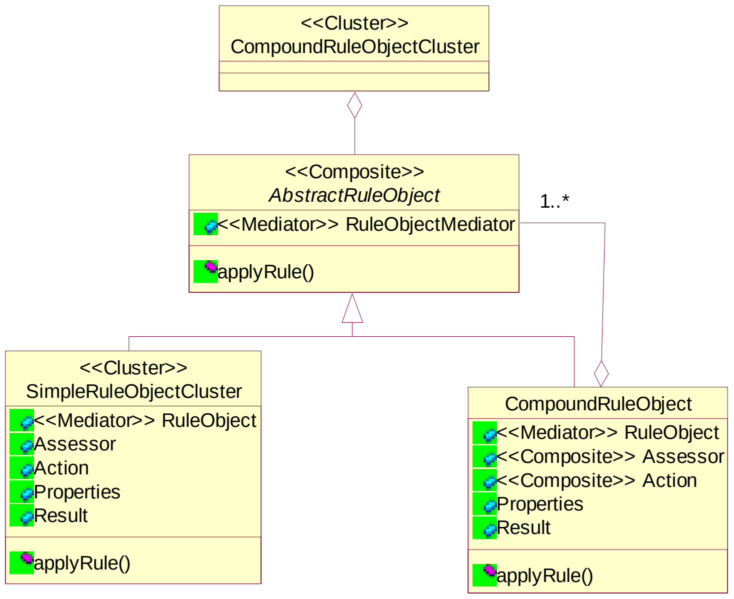


Figure 7: Compound Rule Object Cluster

Telecommunications Provisioning where a nested set of Rule Objects were needed along with their nested set of appropriate Assessors and Actions.

Forces

- **Changes to business requirements entails changes to the design and implementation of rules**. Business rules are expected to change more frequently than the rest of the business object. At the analysis, architectural, design and implementation level, Business Rules may need to be redefined and updated to reflect the changes in policy and business. **Business Rules tend to evolve** over time as a result of new business requirements. Changes to existing rules must be such as to leave the integrity of the system of rules intact and in a consistent state. Consistency implies that modifications to a set of conditions not adversely impact the rest of the system and produce unwanted side-effects.

- **Rules may be time-sensitive**. This is when the rules in a domain are time-constrained; for example, they may pertain to Service Offerings that are offered for only a limited duration for a promotion. Such domains have a rapidly expanding and changing set of rules that may frequently change; sometimes on a day-to-day basis. Although necessary, changing rules in a program are typically costly, as these changes are usually intrusive and will thus have side-effects that will require further debugging and testing**. Making intrusive changes to production code is unsafe and costly**; the potential of side-effects leads to the need for extensive regression testing and re-certification of components.

- **Rules should be centralized**, making them easy to locate and change. Frequently, **Rules are found scattered throughout the design and implementation**; typically in nested if-then-else clauses with many dependencies. Tracing rule requirements to rule design and implementation can be a very error-prone and resource-intensive project.

- **Rules should be de-centralized** and assigned to their relevant classes and clusters. Objects have manners. Manners govern the ways in which their methods should be invoked to guarantee a consistent and valid set of states. Thus, rules can be assigned to classes. Each such class (aka, business object) has a set of methods. Rules govern their valid state, and valid sequences of message-sends to their collaborators and self methods. Rules usually check the state of an object or set of collaborating objects. They operate on data that has been submitted to them via a context. We need to track the status of the application of rules to this set of data (properties). **Rules should know about relevant properties** (data); but the properties should be oblivious to the rules that check their collective integrity from a business perspective.

- **Rules should be scalable.** The same rule structure designed for a small application should **scale** up to the needs and non-functional requirements of a larger application. They start out in a deceptively small context and pretty soon need to be scaled up to handle larger scale transactions. Thus, rules should be simple yet designed to be scaleable.

- **Non-uniform treatment of rule types**. Treating different types of rules differently leads to a variety of unwanted consequences. We need to strive to have uniformity with respect to the fact that every business object should carry its own manners. There are **different types of rules** from different perspectives that apply to different layers in the user-interface, application logic tier and persistence layer. Rules need to have a holistic view of all their types whether they are a set of data elements; knowing (report back) their valid and invalid combinations and values or complex business logic at the middle-tier of an application server.

- **Code clutter: Nested If-then-else statements tend to clutter code** and make it difficult to maintain. Rules are frequently implemented as [nested] if-then-else structures. Rules are usually grown through piecemeal growth [Foote96] rather than designed top-down.

- **Architectural Layers and Rules: Different rules may apply at different layers of the architecture**. Rules may apply differently at each layer: GUI, Webserver, Application, Middleware and Database. Rules are present at different layers and require different mechanisms for their implementation; simple validations in the GUI, complex cross-rule checking at the application layer, stored procedures and triggers at the database layer. **Yet their essential structure needs to be similar to be easily traceable and uniformly applicable**.

- **Business domains contain business processes that are governed by a set of business rules**. These rules are captured as part of company policy and workflow; operations and procedures. They need to be standard procedure, but also need to be updated frequently. They are frequently modeled in information systems whose design needs to change along with the needs of the business.

- Rules need to be **created by and visible to management**; rules **and rule changes need to be visible to programmers**. Programmers implement business rules in code and must change them when executives change them. Management must know which rules have been implemented.

- **Rules are** the valid ways in which objects within a domain are allowed to interact and change state. They should thus be considered as **first-class constructs of the object paradigm** and be identified very early in the analysis process; not as an afterthought. In this new paradigm, in addition to having identity, state and behavior, a class has ***manners*** (i.e., rules that govern the behavior or methods). Manners are rules plus methods (object behavior) plus the meta-data needed to apply the rules and govern the interaction and collaboration protocols of an object.

Applicability

This section discusses the situations in which you might choose to use a Rule Object or one of its variations. Use Rule Object

- To add/modify conditions, actions and rules to business objects dynamically and transparently, that is, without affecting other objects or rules.

- When complexity and scalability makes the use of Rule Method obsolete. Rules can be implemented as if-statements or as methods (e.g., isCompatible()) returning Boolean. **When they grow beyond a proportion of simple checks, it is time to reify them into Rule Objects**. This is especially true if they tend to change frequently.

- When maintaining a system by implementing changes of requirements that can be captured as rules, invariants, business requirements (predicates and conditions), policies, terms of agreements, etc. and we would like to make non-intrusive changes; to reuse existing conditions and actions or create a new rule that is "slightly" different from an existing one, but which will only be in effect for a "short duration" (highly volatile requirements).

- Business objects should know their own manners: how to use their methods in concert with other collaborating business objects; what are valid states and what are invalid combinations of states among business objects. Thus, the laws governing the use of methods, the meta-data that may be required to store this information and the reification of the conditions and actions that embody these "laws" are collectively called a business objects' "manners". [Arsanjani 99;a] Instead of rules being solely accessible through their expression as logic within each business object, rules are collated together into their own objects to facilitate non-intrusive changeability and pluggability. Business objects would then contain a set of business rules.

- When features that a set of business objects need to portray need to be adapted and customized to meet different scalability and complexity constraints. Rule Object can be reduced to a simple Validator for a GUI data entry field or it can scale to a Composite Rule Object with Composite Assessors (Conditions) and Actions; with Properties, and Results. Here is the full spectrum.

Structure

Business rules are expected to change more frequently than the rest of the business object. The impact of these changes will be minimized if the rules are encapsulated in their own classes and held separately; ready to be plugged in and reused. *Therefore, reify rules and their conditions and actions; making them interchangeable and pluggable.*

Maintain rules by making non-intrusive changes to Rule Objects. Add or change existing conditions and actions; add properties (aka, a *context*) to be inspected for valid or invalid combinations (State) by using Rule Objects. A Rule Object Cluster is a Composite of Condition (Assessor) and Action Composites along with their helper classes. These helper classes are Properties that provide a name value set of "fields" that the Condition(s) (Assessor) much check. The result of this Assessment may be recorded in an Assessment Result. If the Conditions all apply, then Actions are performed to change the state of the object or other objects. The Result of the application of these Actions can be recorded in Action Results for reporting or analysis purposes.

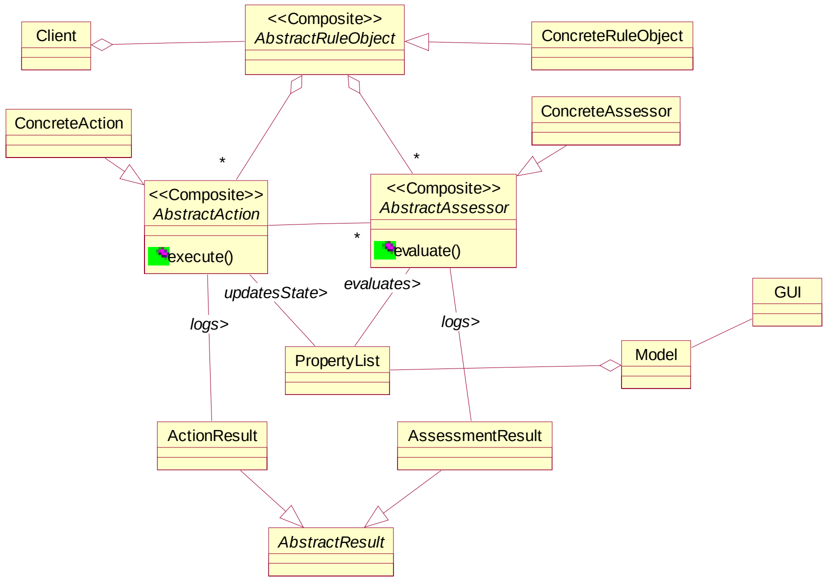


Figure 8: Compound Rule Object

Participants

❍ **Abstract Rule Object**
defines the interface for objects that can have rules added to them dynamically.

❍ **Concrete Rule Object**
defines an object to which additional rules can be attached.

❍ **Rule Object**
Mediate between conditions and actions; apply rules

❍ **Assessor**
check conditions and store results

❍ **Action**
perform actions based on success or failure of corresponding Assessors; record results and change state of collaborating objects

❍ **Properties (aka Context)**
pass in information and State for evaluation (Assessor) or update of State (Action)

❍ **Result**
Provide a super class for AssessorResult, ActionResult and Errors so other participants can record results and report back to client.

❍ **AssessorResult**

The specialization of Result

- **ActionResult**
  A specialization of Result used to record results of actions
- **MediationStrategy**
  Determine how we will execute the Assessors and Actions or even Rule Objects in the case of Compound Rule Objects, Assessors and Actions.

Simple Rule Object: Static View

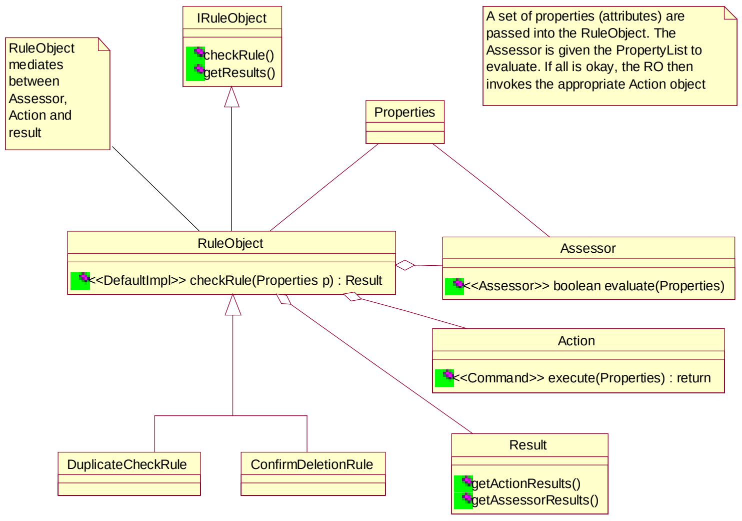

RuleObject mediates between Assessor, Action and result
IRuleObject
checkRule()
getResults()
A set of properties (attributes) are passed into the RuleObject. The Assessor is given the PropertyList to evaluate. If all is okay, the RO then invokes the appropriate Action object
Properties
RuleObject
<<DefaultImpl>> checkRule(Properties p) : Result
Assessor
<<Assessor>> boolean evaluate(Properties)
Action
<<Command>> execute(Properties) : return
DuplicateCheckRule
ConfirmDeletionRule
Result
getActionResults()
getAssessorResults()

Figure 9: Static View of Rule Object and its Collaborators

Compound Rule Object: Static View

The Rule Cluster holds a cluster of Composite Rules; each with their its potential Composite Conditions and Actions. A Simple Rule is a leaf node of the composite and can exist by itself to handle (for example) GUI field editing and validation.

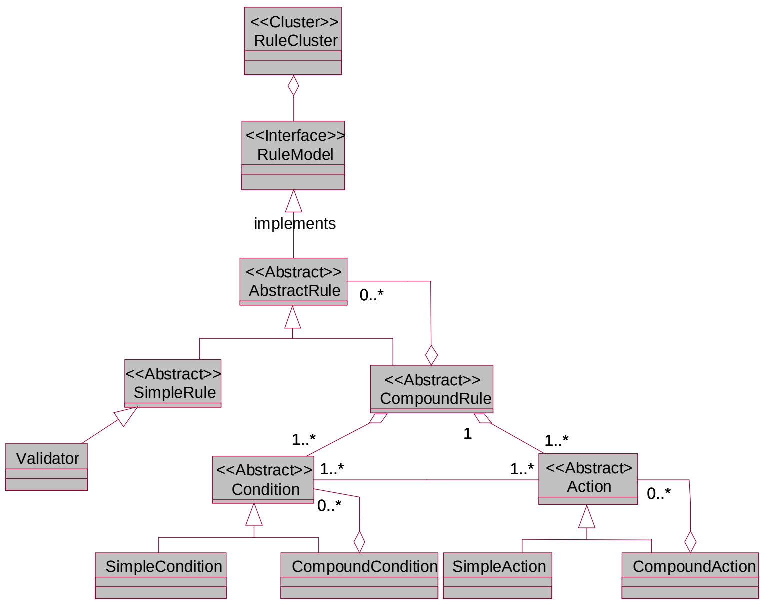
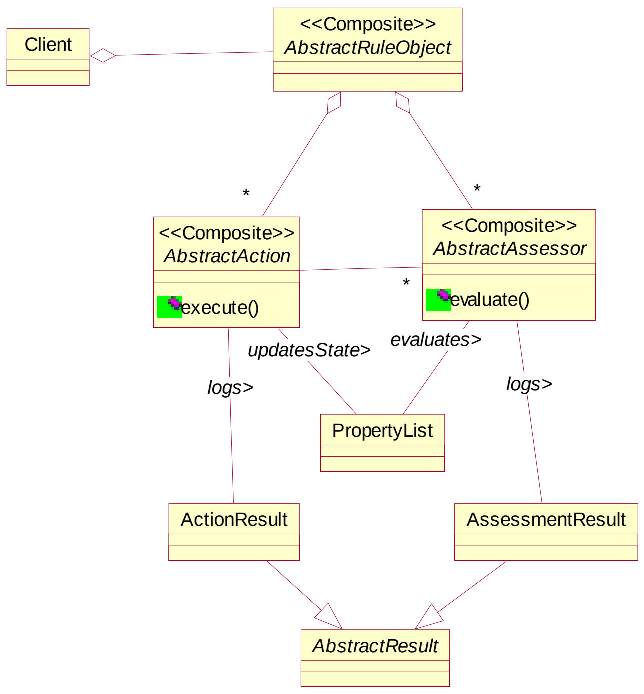


Figure 10: Static View of Compound Rule Object and its Composite Hierarchy

Here is an alternative view of the above diagram. The concentration is on the dynamically pluggable nature of the Rule Object and its constituent parts rather than on its Composite nature as depicted above.

Figure 11: Alternative View of Compound Rule Object



Collaborations

Here is a set of sample collaborations:

1. Set up rules (e.g., cache in a hashtable: "cache and hash")
2. Submit candidate state (pass in property or hashtable or just parameters for state you want to check consistency of)
3. Check conditions on submitted state using Assessor(s) . An Assessor will usually go through the list or hashtable and check each condition, or have a Strategy that will check each condition using an algorithm that in the simplest default case is round robin, but you can choose or define your own optimized Assessor

algorithm by creating, extending a given Assessment Strategy.

4. For each set of assessors, you have a corresponding set of Action(s) that must be performed or ResultState or ErrorLog that must be written to, created, reported back. This may be a Null Object by default for non-important or unimplemented ErrorResults or ResultLogs.

The following sequence diagram depicts the Set up Rules step above:

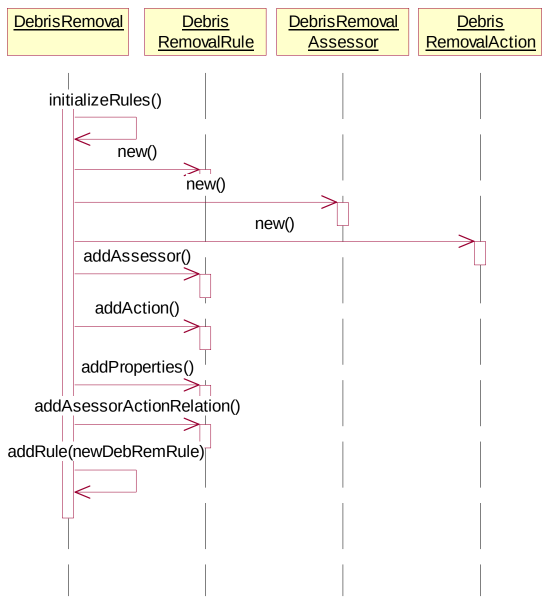


Figure 12: Setting Up Rules

Here are some variations in default collaborations:

One Strategy for Rule Object: (many to many)
If (assessorList.assess(inState))
actionList.perform(inState, outState)
else errorLog.reportOutcome(outState);
	Simple Strategy: (one for one)
	if (assessor.evaluate(inState))
	action.performAction(inState, outState); else
	errorLog.reportErrors(outState);

ValidCombinations:
A Combination is a State which is coupled with a Strategy and a Rule Object. An initially InvalidState State is transitioned into a ValidState if the rule applies. Otherwise, errorLog records the fact and reasons for the invalid State.
	Invalid Combinations:
	If this inState is an InvalidState then
	errorLog.logThisAsInvalidState();
	
	Go through all Rule Objects (Using specified or default [optimizing] Strategy). Determine if this inState matches an InvalidState Combination, if so, reportInvalidState using the
	outState = assessor.assess(inState);
	if (outState.isINvalidState())
	errorLog.reportErrors(outState) else continue; // with next check for next possible invalid state

Consequences

1. *Rules become more easily changeable and reusable*; simpler to maintain. Non-intrusive changes can be made to maintain rules and their Assessors and Actions. The individual Assessors and Actions can be potentially reused in multiple different scenarios. Systems built this way tend to *adhere more to the*

open/closed principle [Meyer84]. Changes to Business Rules have much less of a ripple effect; they are encapsulated within a Rule Object. New Rules can be added by adding more Rule Objects, by creating a Compound Rule within a Rule Object context or by changing the Strategy for a Validator in the case of a Simple Rule. [Client has Abstract Rule Object].

2. *Rules become scalable*. As Composites, their rules, assessors and actions can be stored in a database or cached to account for increased volume, demand and availability, in proporation to the growth of individual rules.
3. *Uniform treatment of Rule Types*. There is no "best " way to design and implement business rules. Actually, rules come in various categories: business rules, validation rules, usage rules, collaboration rules. Validation Rules usually apply to a GUI; some fields are mandatory, some have to be within a given range, some fields are related in an interdependent way: enter values in some, others are enabled or disabled (state-based). Business Rules check valid combinations of inputs; check against invalid combinations . Rules at each layer are treated in a similar manner; though provisions for the uniqueness of each layer is made through customizing a rule. For example, instead of a composite rule, a simple rule (Validator) can be used to validate a textbox in a GUI. Alternatively, a Composite Rule with Composite Assessors and Composite Actions can be used to implement requirements for relating customer care and telecommunications provisioning.

4. *Rules become more easily testable*. Following a rigorous requirements approach for business rules also means that each business rule captured must have a means of being tested.

5. New subsystems will be needed to hold Rule Objects and to allow the rules to be changed by *privileged users*.

6. *Company Policy Repository*. Rule Object prepares the infrastructure for having a *central repository of rules* within a software development organization or within a corporation. Although the Rules (and *Company Policies*) are scattered /distributed within the structure of the organization, they can be centrally managed and browsed, defined and changed from a central location, allowing all interested parties who have registered interest in the Rule or Rule Type to receive notification of its change. This is done using Observer or Publish-Subscribe.

7. *Rule Object Repository* allows corporate executives to be able to define and manipulate rules as policies from a GUI-based Rule Browser. This can then be propagated within an entire organizational structure so that the programmers who will ultimately responsible for implementing the rules in business objects will have a common basis or reference point of traceability.

Implementation

Consider the following implementation issues:

1. Avoid putting simple logic in Rule Objects unless they are apt to change on a frequent basis; Rule Methods are a simpler way of handling rules if they are not needed to be pluggable, adaptable or extensible.
2. Reuse of existing conditions and actions is a good sign of opting to use a Rule Object.
3. Setting up Rule Objects may take a bit of effort but once the framework is set up (see example code below in the Sample Code section) then defining new rules, actions and conditions becomes simple.
4. Strategy can be used to check rules because there may be a family of rules that are related, and need to be applied depending on the state of a given object. Command may also be used to check state, or to execute an action after a condition is checked. Assessor reifies a set of conditions that need to be checked. For example, an cluster's state consists of a number of states of its objects. Each needs to be checked for valid conditions, or they may alternatively (an entirely different kind of effort) be checked for the presence of invalid conditions, so permutations of valid and invalid combinations need to be checked.
5. Results of rule application. The application of rules, leaves a trace: results of the application of rules; their conditions and actions may need to be logged and recorded. Efficiency considerations disallow the logging of every single condition and action set.

6. **Rule Complexity: Rules can be simple or complex** (compound). They can be seen as Composites; or as simple Validators. They may be implemented as a Composite, a Strategy or a Command.

One of the less thought-about factors in business rule design is the fact that error handling is closely coupled to it. It is as important, perhaps, to know why a rule failed as it is to know that it succeeded. Therefore error handling will be considered as part of the implementation consideration section.

Guidelines on Implementation

Each Business Rule is encapsulated within its own class. This can be Simple Rule Object or a Compound Rule Object. A Composite Rule object consists of a Composite Structure containing Assessors and Actions. The RuleObject contains a Composite Assessor, A Composite Action and an ErrorResult. The ErrorResult is there in order to specify what the error was that disallowed the rule to fire the action after evaluating the Assessor. The Assesor is similar to a Command. It has a method "assess()" which returns boolean. If it is a Composite, the Assessor must successfully assess all of its constituent elements before returning true. If there is a problem, the Assessor logs this as an error message/condition in the ErrorResult.

If all is well and the Assessor has evaluated the conditions to be true, then the Action is invoked. The Action is a Command, possibly Composite that will either change the state of the current Client object which is using, containing the Rule Object, or more likely, will collaborate with other objects to create a valid state for the system, based on the evaluation by the Assessor. If at any time there are error conditions arising, these will be logged within the ErrorResult object to which Action has a reference.

ntation 2

A Rule Object usually forms a cluster. This Cluster consists of {RuleObject, Properties, Assessor, Action, Result}. The attributes or state that is passed into the RuleObject is assessed or evaluated by the Assessor. If this evaluation is successful, RuleObject (acting as a Mediator between Property, Assessor, Action and Result) sets the results of the Assessment in the result Object then asks Action to execute on the Properties, potentially changing their State.

The results of the execution are set in the Result object. This may provide meta-data as to the fact that, for example, a System State change took place. Properties are used to encapsulate State from a collaboration or a single object's attribute values. Therefore, not only are we interested in evaluating a set of attributes and performing actions based on the result of the evaluation, we frequently would like to know of the intermediate results of the evaluation and the execution of each action as intermediate steps. This information is recorded in the Result object. The State or set of attributes and values which are used as the basis for evaluation by the Assessor are passed into the Rule Object or the Rule Object aggregates and creates its own Properties object(s) and passes them to its Assessors and Actions on a demand-driven basis.

Sample Code

Here is an example of a Simple Rule Object that can be applied at the GUI layer. Although this demonstrates Simple Rule Object for a GUI layer, Rule Object is applicable in all layers: application, protocol and persistence layers.

```
class BusinessRule implements ActionListener{
           private JDialog theDialog;
           private Frame theFrame;

           public void setTheDialogJDialog aDialog){
               theDialog = aDialog;
                   }
           public BusinessRule(){
           }
           public boolean assess(int number){
               if (MIN_SERVICES <=number && number<= MAX_SERVICES)
                   return true;
               return false;
           }

           public void actionPerformed(ActionEvent event){
                if (event.getActionCommand() == "SubmitButton"){
                    theDialog.dispose();
                }
            }
        }
```

This is the simplest case where the Rule Object is merely one Class and acts as the Listener. Each time a key is typed in a field in a Dialog, assess() is called to assess whether a valid value has been entered or not. In this case it is a range that is being checked.

Below, is an instance of use.

```
/**
  * Register the field for entering the number of people
  * @param field The field used to enter the number people
  */
public void registerNumberField(final JTextComponent field) {
            numberOfServices = field;
            DocumentAdapter documentAdapter = new DocumentAdapter() {
                protected void parseDocument() {
                    int count = 0;
                    try {
                        count = Integer.parseInt(field.getText());
                    } catch (NumberFormatException e) {
                    }
                    if (rangeRule.assess(count) )
                      serviceCount =  count;
                    else
                      serviceCount = 0;
                } // parseDocument()
              };
            field.getDocument().addDocumentListener(documentAdapter);
        } // registerserviceCountField(JTextComponent)

Register the Rule Object as the button’s listener:

public void registerOKButton(final JButton btn){
                submitButton = btn;
                submitButton.addActionListener(rangeRule);
                submitButton.setActionCommand("SubmitButton");
                rangeRule.setTheDialog(myParentDialog);
            }
```

The Rule Object Framework

Define a Rule Object that contains its Assessors and Action; Properties and an ActionAssessor Map to help determine which Actions relate to which Assessors.

```
public abstract class RuleObject
{
        private Vector assessorVector;
        private Vector actionVector;
        private RuleProperties ruleProperties;
        private ActionAssessorMap actionAssessorMap;

public RuleObject()
{
        actionVector = new Vector();
        assessorVector = new Vector();
        actionAssessorMap = new ActionAssessorMap();
        ruleProperties = new RuleProperties();
}
public boolean applyRule(){
boolean assessorReturns = true;
        Assessor tempAssessor;
        Action tempAction;

        // get an enumeration of the assessor vector
        Vector tempVector = getAssessors();
        Enumeration e = tempVector.elements();

        // for each assessor in the enumeration, call the evaluate
        // method and get the return value
        while(e.hasMoreElements())
        {
                tempAssessor = (Assessor)(e.nextElement());
                assessorReturns              =             assessorReturns              &&
tempAssessor.evaluateAssessor(getRuleProperties());

        // determine if there are any actions that need to be executed for this assessor
                Vector tempActionsVector = getActionsForAssessor(tempAssessor);
                if(tempActionsVector != null)
                {
                        Enumeration tempEnum = tempActionsVector.elements();
                        while(tempEnum.hasMoreElements())
                        {
                        tempAction = (Action)(tempEnum.nextElement());
                        tempAction.performAction(getRuleProperties(),
                                        new boolean(assessorReturns));
                        }
                }
        }

        return assessorReturns;
}
/**

}
```

Known Uses

The Rule Object has been used on several projects by various teams that the author has been involved with. Domains include Telecommunications (Customer Care and Billing), Healthcare, Insurance, Automotive, Higher Education industries, Sales, e-brokerage.

Rule Object has been used in the implementation of the Java Business Frameworks [Arsanjani99b]

IBM San Francisco uses Policy Common Business Object and Pattern which use a similar concept.

Rule Object motivated and was used in the “If-Then-Else” Framework, by Paul Corazza [Corazza].

IBM WebSphere Application Server Enterprise Edition, Component Broker’s Managed Object Framework implements Rule Object.

David Taylor mentions a similar scheme in his Object Magazine column on Business Rules.

Related Patterns

Peer Patterns[4]

Rule Object acts as a **Mediator** to Assessors and Actions; Properties (Context) and Results. Rule Objects determine which Assessors should be used in the assessment of Properties, with a possible recording of Results in an Assessment result. If the assessment was successful, the appropriate Actions are invoked, possibly updating or changing the State of Properties. Results may be recorded in an Action Result. For a more in-depth explanation see Appendix B: Peer Patterns.

[4] Peer Patterns are patterns in the same pattern language that work together to resolve and balance forces.

Related Patterns

Rule Object uses several more fundamental design patterns. It can therefore be considered to be a Compound Pattern (or a Composite Pattern [Riehle98]). But not all Compound Patterns are Patterns themselves; Compound Patterns are merely a namespace that labels a set of patterns that are repeatedly found to work in concert in many different occasions. To implement the Condition participant, we suggest the use of an **Assessor** [Arsanjani98]. To implement the actions (Simple or Compound) use a **Command**. The Compound Rule is itself a **Composite**. SimpleRule uses a **Strategy** to implement its Validator participant. The Rule Cluster has a **Builder**, which uses an **Abstract Factory** to produce individual instances of Rules, Conditions and Actions.

The **Assessor** is really a special case of a Command that is found to recur in multiple contexts. Instead of executing a command, the Assessor has an assess() method which returns either a Boolean (in the case of a simple assessor) or a composite (in the case of a Compound Assessor). The Assessor may further be implemented as an Interpreter if it needs to determine the validity of a “rule string”; i.e., a string containing sentences of a “Rule Language”.

Rule Object is connected with **Visitor** in that it shares the following applicability to a large degree: “many distinct and unrelated operations need to be performed on objects in an object structure, and you want to void "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them. ”

Alternatively, Assessor may be used in the context of Grammar-Oriented Programming [Arsanjani89] where domain analysis determines a domain language. The domain language is then described in terms of a domain grammar. The interaction between the domain objects is fully described by the domain grammar. Use-cases that trigger collaborations trigger the domain grammar and the message is passed as an input stream into the parser that is interpreting or parsing the grammar. Object’s “manners” are described in terms of the meta-model that is represented as a grammar.

This is expressed in the following pattern “configurable workflow” (a version of Rule Language):

Pattern Name

1. Configurable Workflow

Also Known As

Grammar-based collaboration, reified collaboration, adaptive workflow

Context

Every application has a flow: dataflow, workflow and object message flow. A business process typically has a workflow that is captured by a part of the application flow. We need to represent this workflow between components in an enterprise application. There may be a central point of control or controls have been distributed. Although the general thread of business logic may be similar, different deployments of an application may require different customizations in terms of the sequence of operations for workflow. Although the set of objects that collaborate maybe the same, the sequence in which the Objects collaborate maybe dependent upon things such as user access rights, [Pree 99] (system State). But the functionality required to support the business domain changes frequently. This change in requirements is reflected in changes in code. This results in changes to business objects; usually disrupting the stability of the software architecture. This leads to imbalances in the software architecture that need to be re-balanced [4]. One of the ways in which to re-balance the lopsided software architecture after introducing new functionality is to apply the changes to a resilient, adaptive software architecture in the first place. The changes need to be absorbed into a resilient software architecture. The question remains on how to explore the changes in a cost-effective manner without impacting the performance of the software architecture.

Problem

How do you plan and engineer flexibility to adapt to changing circumstances for variants of a business flow?

How can we change the workflow with minimal disruption to the current software architecture (current processing)? How can we design a workflow that is adaptive to change?

Discussion: The changes may be to functional or non-functional attributes of the SA.

How can you design a workflow that is easy to change and customize for different realizations of a business process (product lines and business lines or geographical areas)?

Further problem statements: How do you adaptively alter the state, collaboration and self-description of a component that must evolve to meet new business challenges?

Forces

- **Change the internals vs. Change the externals**: should we allow intrusive changes to be made into a component or should we externalize those change points using Variation oriented design? In this case, how can we non-intrusively change the workflow, collaborations, and message-flow?
- **Using reflection vs. run-time startup**: should reinitialize the workflow at system startup or should we reflected the Inquirer about the configuration of a given set of components at run-time?
- Writing an application for **central office vs. configuring it for local offices**
- **Hard-coding the workflow vs. creating an adaptive collaboration**
- **Using aspects or using a reified collaboration**: aspect oriented programming, subject oriented programming and alike introduce special constructs at the programming language level. Should the architect opt for such a decision or should he bases decision on the concept of a reified collaboration which allows the properties of dynamic configuration, collaboration and self-description?
- **Stable vs. changing software architectures**: should changes in the business impact the software architecture or should the changes be absorbed in resilient software architecture?

Solution

Model a domain-specific language (DSL) to model the business domain. Externalize this language as a configurable workflow based on Variation oriented Design, so that changes to the component may be done independently of the component active code base. The component reads in its configurable workflow at startup, and configures itself dynamically for a new set of collaborations based on the new requirements reflected in the updated workflow. This process is called grammar oriented Object Design: starting from a domain specific language, externalizing it in the grammar, defining it is a configurable workflow for a component that exhibits the properties of dynamic configuration, dynamic collaboration and self-description. Variation oriented Design separates the changing from the non-changing aspects of the domain specific language that allows externalization of hot spots in the domain specific language.

Consequences

You now have a software architecture and an application which is resilient to change. After a subsequent period of variation oriented Analysis, changes have been identified and the major ones have been externalized as a business domain language. Now, updates the business logic and the rules governing the collaboration between maintained by updating the domain specific language that has been externalized.

The workflow has been declaratively externalized in a set of production rules in the style of grammar-oriented object design [].

Example

You have an order-processing system that you would like to implement for an e-business scenario. The workflow for this application needs to be customized for each locale it will be implemented in. In order to define a configurable workflow that can be adaptively modified to changing needs and customized based on a particular configuration need, we represent the domain's business rules and flow in a domain-specific language. The production rules that represent this language govern the behavior of business objects in this domain.

Note, that the application has been partitioned into subsystems following a domain analysis and a subsystem analysis. The "manners" for the system as a whole are represented in the following EBNF-like notation:

```
Online Purchase = {Identification, Presentation, Selection, Purchase, Confirmation, Order Fulfillment}
Identification = {Challenge User with Login, Verify UserID and Password}

Presentation = {Display Menu}

Selection = {Browse Product Catalog, Select a Product Item, Shopping Cart Operation, Selection}

Shopping Cart Operation = { {Add Item to Shopping Cart | Delete Item From Shopping Cart | Save Shopping Cart |
[Shopping Cart Operation] }, Checkout }

Checkout = {Complete Order Info}

Complete Order Info = {{Verify Billing and Shipping Address| Select Billing and Shipping Addresses}, {Verify Shipping
Method | Select Shipping Method}}.

Purchase = {Review Order, Review Terms of Agreement  {Acknowledge Terms of Agreement, Submit Order | Cancel
Order | Change Order to Quote}}.

Confirmation = {Send confirmation number to user}

Order Fulfillment = {Pick and Ship Order}
```

This domain-specific language can be used to represent the workflow of the application. Each subsystem will have a set of rules governing its behavior. The manners for each subsystem can also be represented as a similar domain-specific language that governs only the subsystem's mode of behavior.

This workflow can be externalized as an XML file and adaptively modified at runtime to accommodate new changes to requirements. This is called a Configurable Workflow and the design principles used are based on Grammar-oriented Object Design.

Related Patterns

The **Configurable Workflow** is closely related to **Configurable Profile**. The latter consists of a set of personalized settings, rules and filters that users create to personalize their experience of a portal or, increasingly, of user-aware applications. Access control and application security are obvious applications of a **Configurable Profile**. A Configurable Profile is related to **Properties** [Foote97]. These two patterns are special cases of a meta-pattern called ***Configurator***, which provides the ability to dynamically configure and re-configure static and dynamic (e.g., workflow and collaborations) of a software [sub] system.

References

[**Arsanjani99**;a] Ali Arsanjani. "Service Provider: A Domain Pattern and Its Business Framework Implementation," presented to PloP '99. http://st-www.cs.uiuc.edu/~plop/plop99/proceedings/Arsanjani/provider3.pdf

[**Arsanjani99**;b] Ali Arsanjani. "Analysis, Design, and Implementation of Distributed Java Business Frameworks Using Domain Patterns" in Proceedings of Technology of Object-oriented Languages and Systems 30, *IEEE Computer Society Press* 1999, pp. 490-500.

[**Arsanjani89**] Concepts of Grammar-Oriented Programming, Azad University Technical Report, 1989.

[Arsanjani2001a]) Proceedings of the 39th TOOLS conference, August 2001.

[**Corazza**] Paul Corazza. Using the if-then-else framework, Part 1: Code maintainable branching logic with the if-then-else framework. Available at: http://www.javaworld.com/javaworld/jw-03-2000/jw-0324-ifthenelse.html

[Fow96] Martin Fowler. *Analysis Patterns.* Reading, MA: Addison-Wesley, 1996.

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

[MRB98] Robert Martin, Dirk Riehle, and Frank Buschmann (eds.). *Pattern Languages of Program Design 3.* Reading, MA: Addison-Wesley, 1998.

[Odell96] James Odell and James Martin, *Object-oriented Methods: Pragmatic Considerations*. Prentice-Hall, 1996.

[Riehle98] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. "Role Object." In *Proceedings of the 1997 Conference on Pattern Languages of Programs* (PLoP '97). Technical Report WUCS-97-34. Washington University Dept. of Computer Science, 1997. Paper 2.1, 10 pages.

[VCK96] John M. Vlissides, James O. Coplien, and Norman L. Kerth (eds.). *Pattern Languages of Program Design 2.* Reading, MA: Addison-Wesley, 1996.

Appendix A: Rule Object: A Pattern Language for Adaptive and Scalable Rule Design and Construction (Management)

The following table summarizes the patterns in this pattern language and provides an initial definition and context for each one. The next section in this appendix outlines a map of the pattern language showing how the patterns relate to one another; supplying transition criteria (for going from one pattern to another) and the forces that will be encountered before the transition and once the transition is made.

Rule Pattern Language Summary

1. **Rule Object** – ***You want to*** Provide extensibility and adaptability to business processes, without endangering them with intrusive changes, by making the rules governing them pluggable. ***Therefore***, externalize the rules during design time. Separate the definition of the rules from the objects that are governed by those rules. Business objects tend to participate in more than one configuration, collaboration and context. Therefore, avoid statically binding a business object set to a given context by embedding the rules within it. Separate the rules governing an object's behavior from the object itself. Thus, you can re-define and re-configure how the object should/will behave in new contexts and in the face of new requirements. ***For example***, Define new products and services based on the set of standard, atomic product and service types without stunting the growth and resilience of the product definition process; geared to meet and overcome new demands of clients, marketplace and competition.
2. **Assessor** – ***You want to*** check conditions that tend to vary a great deal. You don't want to rewrite the code every time, with a slightly novel twist. New requirements call for new assessment, evaluations, based on which you take action; to accept an order, to submit a loan to register for a service based on eligibility. Your conditions can be Boolean expressions or events that are guards in astate transition diagram. ***Therefore***, separate and reify the conditions from the action parts of a Rule. Assess a set of conditions based on an input set of Properties (Context); record results of evaluations in a Result.
3. **Action** – ***You want to*** manage the changing set of actions you have to take when a condition has been checked. It started out to be straightforward; but now there are multiple actions, each with a new twsit; some are radically different. ***You want to*** avoid cluttering your code and want to be able to you're your actions in response to events or conditions rapidly. ***Therefore***, separate and reify the action part of a Rule from its condition part. Perform actions in continuation of the results of the Assessment of conditions, record results and update Properties (Context; Context is Updated; Context is not Updated) and State (Rule changes Object State; Rule maintains object State) of pertinent objects in collaboration.
4. **Rule Cluster** – Components the Composite definition and application of Rule Objects; optimize rule application through the definition and selection of a rule application policy.
5. **Rules have State** – Maintain State between rule checks and applications
6. **Rules are Tracked** – track history, changes, condition/action set pairs
7. Document **Rules as Patterns**—capture rules as patterns to track and report reasons for solution of issues and consequences
8. **Rule Object Repository**: Centralize Rules in Corporate Repository
9. **Rule Access Rights** – managers should be able to create rules; give access rights to control unwanted or accidental corruption of rules
10. **Rules Change Process** – New rules impact old processes
11. **Components Have Manners** – clusters of collaborating objects have laws governing their behavior and meta-data about these rules (laws).
12. **Rules as First-class Constructs** -- conducting analysis and design based on object "manners"
13. **Rules as Production Rules** of the Application Domain Grammar – Grammar-oriented object design; define a domain language and grammar for a domain; implement it using a parser accepting input from an application running in that domain
14. **Persistent Rules**—Handle proliferation of subclasses and objects as "data"
15. **Hash and Cache**—provide efficient and quick access to subclasses and objects as the numbers increase

16. **Remedy Rule Proliferation**—Handle object proliferation syndrome
17. **Rules Evolve** – Rule evolution for business survival
18. **Rule Change Impacts Architecture** – information system architecture, functional and non-functional requirements are impacted by changes in rules.
19. Rule Engine – The number of rules, their dependencies and potential conflicts are becoming unmanageable. You have separated out the rules within the same program or you have externalized them in a Configurable Profile or an External Rule, as an XML file or a EBNF-grammar. But now you need to handle this additional complexity. Therefore, externalize the rules during run-time .
20. Rule Analysis: Add a phase in your development life-cycle to concentrate on sampling a subset of business rules that form a representative picking of the population of your business rules across tiers and business domains. Create a Rule Matrix showing dependencies and start isolating clusters of dependencies that are cohesively as unit. Provide a solution for each cluster and place each cluster in its appropriate place within the architecture using Rule Placement.

Appendix B: Rule Pattern Language Map – Business Process Emphasis

The Rule Object Pattern Language can be briefly described in the following narrative. We will then take a look at the relationships between the individual patterns and how one can potentially follow another or is somehow related to another pattern.

Note that some patterns are tiny pattern languages in disguise: Rule Object consists of Validator, Simple Rule Object, Assessor, Action, ErrorResult, Properties, Compound Rule Object and MediationStrategy (showing what algorithm to apply in sequence when applying rules, conditions or actions. These can be simple round-robin style or have a more complicated algorithm such as the Rete Algorithm to apply rules. The default is the simple round robin; just go through the vector of rules and apply each one sequentially. You may want to have a weighted vector or a Hashtable which gives a priority to the application of each rule of assessor or action, should they be a collection or should they be Composite).

You can start at various parts of the ROPL language and work your way around by applying patterns to resolve forces in the problem domain. So for example, there are many use-cases for starting the journey and resolving forces that arise in the problem space trhough the application of the pattern language's patterns.

Use-case 1: You want to build an insurance application and need to implement existing business rules that are given as part of the requirements specification.

Use-case 2: The organization wants to organize its business rules. You are on a business rules hunt and Document Rules as Patterns, Create a Corporate Rule Repository and store the rules.

Use-case 3: You have a Corporate Rules Repository and want to configure it for general use. You assign Rule Access Rights to managers and developers who have access to the Rules in the Repository. You Log Changes to Rules as management changes the business to meet market and operational demands. Having this repository affects how business is conducted so Rules Changes Business Processes. As changes are made to rules, the architecture is impacted; maybe rules are concentrated in a middle tier rather than being scattered and duplicated in various middle tiers, database triggers, GUIs, etc.

Use-case 4: During development with Rules Objects, Rules Evolve and changes to rules are handled by creating new Assessors and Actions, or reusing existing ones in new combinations to reflect new rules and processes. Properties are created as raw inputs for Assessment and Actions may update the Properties, yielding a Results or ErrorLog which is reported to the user and/or logged in a persistent store.

Use-case 5: As Rules start proliferating, we Handle Rule Proliferation by encapsulating related rules in Rule Clusters and compose Compound Rules from Simple Rules. We start reusing Assessors and Actions. We may Hash and Cache them in memory or in a middle tier for optimal performance. As they proliferate, we may decide to persist our Assessors, Actions and Rules , Properties and ErrorResults and thus Create Persistent Rule Objects.

Use-case 6: During the course of using Rule Objects we may find that we need to Track Rule State by maintaining it as a Momento or some other mechanism, maybe even a simple static AssessorResult would do the trick.

Use-case 7: As we conduct more requirements analysis and create more Rule Objects, we find that there are Clusters of collaborating classes tat work with each other to achieve a business goal. These Business Objects form Clusters. These Clusters tend to have rules governing their interactions within the cluster, we call this their "manners". Thus, Clusters have manners and Business Objects have Manners.

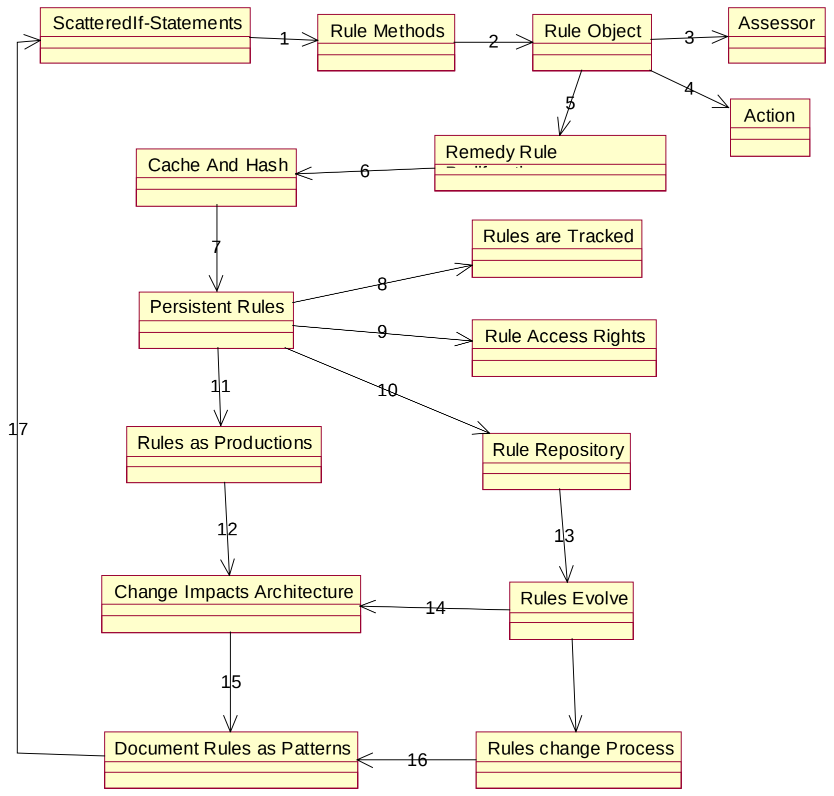


Figure 17: Rule Object Pattern Language Map

The Following Table discusses the above figure 17 and specifies the transition criteria for each edge of the graph.

Number	Transition Criteria/ Forces
1.	Complexity, adaptability, composability, organization
2.	Frequent changes to rules rapid turnaround needed; reuse of designs, rules, conditions and actions; maintenance issues as complexity increases; scalability
3.	Uses; interchangeable conditions needed (again zip code is checked against the service type and features)
4.	Uses ; interchangeable and reusable, recurring actions (e.g., error messages, updates to db, etc.)
5.	Subclasses and objects proliferate
6.	Greater proliferation (object proliferation syndrome); need to have faster access to objects; need to manage and maintain object assets
7.	Even greater proliferation; faster access; treat objects as “data”; databases handle this well
8.	Need to track changes and report
9.	Many people have access, access needs to be controlled; legal and security purposes; stability and control
10.	Common portal for corporate rules; organized and browseable; development teams need frame of reference from which to be updated on new needs for rules changes; dynamic, cross-domain reuse
11.	Multi-domain reuse; product line architectures; create a domain grammar; employ grammar-oriented object design [Arsanjani90]
12.	Architecture and business process driven by rules
13.	Rules and business change; need to stay in business, maintain market share and profit; promotions, new services, new offerings; new legislation; deregulation, etc.

14.	Change of rules impact architecture in terms of functionality and non-functional requirements (availability, security, performance, persistence, scalability, etc.)
15.	Corporate awareness of rationale behind rules, issues the rules are trying to solve need to be documented, what are the reasons and issues; how does this rule solve them; what is the resulting consequence (sound familiar?!)
16.	<same as above>
17.	Implement rules starting with the most appropriate design or implementation mechanism

Appendix C: The Peer Patterns

A Brief Look at Rule Object's Peer Patterns

Peer Patterns[5]

Rule Object acts as a **Mediator** to Assessors and Actions; Properties (Context) and Results. Rule Objects determine which Assessors should be used in the assessment of Properties, with a possible recording of Results in an Assessment result. If the assessment was successful, the appropriate Actions are invoked, possibly updating or changing the State of Properties. Results may be recorded in an Action Result.

Assessor

Assessors encapsulate and evaluate conditions based on a set of input Properties and record the results of the evaluation within their AssessmentResults. Assessor is similar to Command. A Command executes an operation whereas an Assessor assesses a set of conditions based on input State or Properties. The difference is that it has an evaluate() method rather than an execute(); and collaborates with a Property List and a ErrorLog or ResultSet:

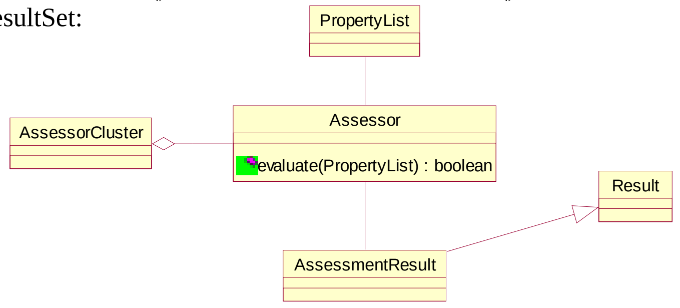
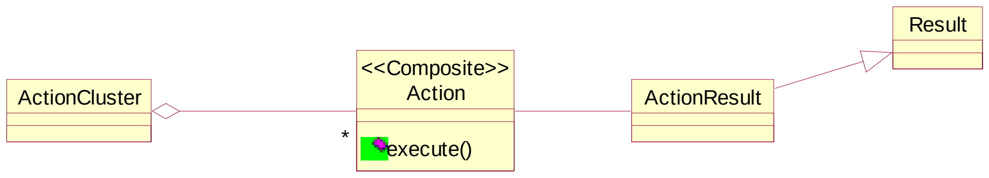


Figure 13: The Assessor (Condition Evaluator)

Action

An Action executes by updating State or invoking behavior in other collaborating objects. It records the result of its actions in an ActionResult which may contain an ErrorResult or ErrorLog. Actions can be implemented as Strategies, Visitors, Interpreters, Commands. Here is the structure of its cluster:



The Action Pattern

Appendix D: Types of Rules

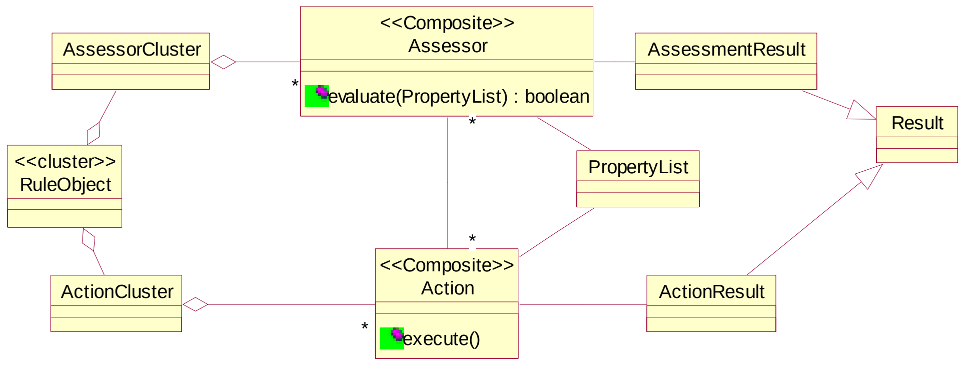
Rules come in various types. Here is a list of scenarios that provides motivation to define a taxonomy of rule types based on complexity, scale, layer and function. Specifically, 1-7 address tier-specific types of rules and 8 contains a list of non-tier specific types of rules.

1. Field on a form needs to be validated for non-blank, correct domain of values, etc. Objects check these; UI sends the app object a message to check this data value.
2. Set of interdependent fields must be validated collectively.

[5] Peer Patterns are patterns in the same pattern language that work together to resolve and balance forces.

3. Compound data spread across multiple forms needs to be cross-validated; data in one form should not violate another data value in another, related form. Implemented using a mediator pattern whom checks to see if all data are consistent without needing the forms to know about one another.
   3.1. Invalid combinations need to be identified and reported or disallowed altogether
   3.2. Valid combinations tested for validity
4. Checking with policies and verifying business rules in the middle tier constitutes the bulk of application (or "business") logic; which may be accessed by many types of clients: such as PDAs, thin-clients, fat clients, etc.
5. Checking rules at the database layer in the form of triggers and stored procedures.
6. Middleware communication rules.
7. A variety of security-related rules; authorization, authentication, non-repudiation, etc.
8. Regardless of tier or layer s in which they reside, rules can pertain to computations (functions, mappings, transformations usually expressed or expressible via mathematical formulae), for example: When a Coverage is selected (added to a policy) it may have causes of loss associated with it. These are fixed, and cannot be changed. If a coverage has causes of loss, when the coverage premium is calculated, it asks each cause of loss to calculate its contribution to the premium and sums them. If there are no causes of loss for a coverage, then the coverage has a premium of $0.00.
   8.1. Some restrict and constrain business processes and data that are input to drive them;
   8.2. Some pertain to the workflow order and "Route", "Rules", "Roles";
   8.3. Some constitute "terms of agreement" or "conditions", such as "service level agreements" which are more legal in nature
   8.4. The enforcement of certain types of rules results in "positive inclusion": a Coverage is valid if and only if …. Sometimes, however, we have "negative inclusion": for example, "If the zip code of the applicant is within <list of high-fraudulent zip codes> then we will have to do a pre-paid calling card; other times we have "invalid combinations" the conjunction of a set of conditions results in an invalid state: "if the loan requester has a second house and the primary house is within a high-risk earthquake zone and they have no insurance, then we will have to reject the loan if credit is below <xxx> amount".
9. The design of rules and their implementation are two separate things: e.g., rules can be designed as business constraints and implemented in the data layer via referential integrity rules that can be modeled as specific types of associations/relationships.

Actions are related to Assessors in the following way:



Actions and Assessors