Domain + Control + Policy ≡ Reusability

Bindu Rama Rao
Integrated Solutions
brr@mcs.com

and

Lizette Velázquez
Lucent Technologies
lizette.velazquez@lucent.com

Introduction

Reusability of objects across features within the same product is possible if the domain objects have no knowledge of the activities in which they are employed. This requires the separation of activity-specific information from objects in the domain. As a partial solution, control objects are recommended [Jacobson, 95] for encapsulating control type information that are associated with any activity. This scheme works fine for modeling a product for a specific customer in terms of reusable domain objects and activity-specific control objects. However, when similar products for multiple customers are to be modeled, the customer specific information are unfortunately captured in control objects or domain objects or both. Hence, the reusability of objects is often compromised. A complete solution would require the use of policy objects along with control objects.

This paper addresses the reusability issue during the domain analysis phase of an OO software development lifecycle (e.g. [Booch, 94], [Jacobson, 95], and [Rumbaugh, 91]). It recommends a set of three patterns that together promote reusability, namely the Plurality of Customers and Features, the Control Information Decoupler, and the Policy Decoupler. The first of these employs the other two.

In this paper, by domain objects, we mean objects that naturally occur in the problem domain under consideration; by underlying engineering technologies, we mean the technologies that are part of the domain. For example, hybrid fiber coax (HFC) and switched digital video (SDV) can be considered to be two possible underlying technologies for broadband telecommunications applications.

Plurality of Customers and Features

Problem

How can reusable assets be created that can be used across several customers, when only one customer at a time is driving the requirements?

Fact: The requirement analysis is performed one customer at a time.This enforces the delivery of a specific product to a specific customer, employing a specific underlying engineering technology.

Fact: To be a profitable turnkey business unit, we need reusable assets - to support

- Several underlying engineering technologies, each with their own engineering policies.
- Multiple product lines.
- Several customers for each product line.

Context

Software development of a turnkey, OO product, custom-tailored to several different customers.

The approach to software development is the creation and usage of reusable assets, made possible by object oriented technology.

Forces

Customers provide information on features and policies that must be incorporated into a product. However, the policies and preferences are seldom distinguishable from the customer's feature requirements as the customer does not have to think in those terms.

We need to achieve reusability while supporting multiple customers. But,

- different customers have different policies for the same features, and
- different customers have different yet similar definitions for the same feature.

Capturing information about too many customers and activities in domain objects, not only causes a dependency but also makes them difficult to reuse and maintain.

Solution

The solution is the pursuance of the following reusability strategy:

- Using the <u>Control Information Decoupler</u> pattern to decouple control information from domain objects. This makes the domain objects reusable across features.
- Using <u>Policy Decoupler</u> pattern to decouple policy information from control and domain objects. This further promotes reusability.

Both the steps described above must be performed for maximum reusability of the domain models. Performing only one of the two will result in a partial solution.

Resulting Context

The resulting context is one where we can achieve reusability of domain objects across features and policies. This benefits are those that are accrued from the *Control Information Decoupler* and *Policy Decoupler* patterns.

Rationale

Explicit models of control information in control objects enables us to reuse domain objects across features. Again, explicit models of the policies of different customers and technologies makes it possible to properly incorporate the policies and related information that the customers provide. Products can then be developed by employing truly reusable assets (i.e. control and domain objects) and associated policy objects.

Control Information Decoupler

Problem

How do you make domain objects reusable while employing them for multiple activities?

How do you limit the impact of a change in a feature on other features, when domain objects are reusable for multiple features?

Domain objects are used in multiple activities. Different activities can manipulate the same objects in different ways. While establishing the behavior of the domain objects using features as a guideline, control type information infiltrate the behavior.

Context

A product is being developed for one customer and the features are provided for the customer. During Requirement Analysis, various features are identified and specified. In Domain Analysis, domain objects that stem from the obvious entities in the domain are analyzed, identified and specified. The product may be developed over multiple releases, with additional features added to each release.

Forces

Reusability of domain objects across features: For new features, need to reuse objects that already support existing features. Forcing the reuse of objects that already incorporate control information from other features results in explosion of methods, dependency of features, and makes objects unusable. In short, the need to reuse objects vs. the inability to reuse what exists due to their corrupted nature (corruption by activity specific control information).

Modality of Development Efforts: While modeling the problem domain for reusability, the behavior of domain objects are established using features as a guideline. This invariably results in the infiltration of control type information in the behavior, thus hampering reusability.

Impact of Feature Changes: Common objects are used to support multiple features. It should be possible to change one feature without impacting others, or with minimal impact or major rework.

Traceability: Objects have inherent behavior and they support features. If they incorporate control information, such information are not distinguishable from the inherent behavior of those objects. It must be possible to trace every feature.

Solution

- In the Domain Analysis, following Requirements Modeling where activities are first encountered and studied, study the activities for participation and sequencing information.

- Create control objects to encapsulate control information.

Information that tend to be feature -specific and not the inherent behavior of the object must be encapsulated into control objects.

Control objects are objects that encapsulate the following:

- participants: identity of the participants in an activity or feature
- sequencing: he sequence of interactions between the participating objects

Resulting Context

Proper decoupling of control information from domain objects.

We can now achieve more reusability across features, i.e. experience the benefit of decoupling using control object approach and the explicit modeling of control information.

Rationale

Impact of Feature Changes: Control objects encapsulate control information of a feature and provide information on participating objects. As a result changes in a feature does not require changes to control objects of other features.

Reusability of domain objects across features: Capturing control information in control objects makes the domain objects cleaner, easier to maintain, and more reusable.

Modality of Development Efforts: It is easier to model newer features using existing domain objects due to the ability to assign control type information to newer control objects while reusing domain objects. Every feature will have an associated control object, making it easier to encapsulate control information.

Traceability: Control objects provide us with a placeholder for feature related control information, thus offering traceability.

Sketch

In the general case, control information may be embedded inconsistently across domain objects. Figure 1 below depicts this scenario.

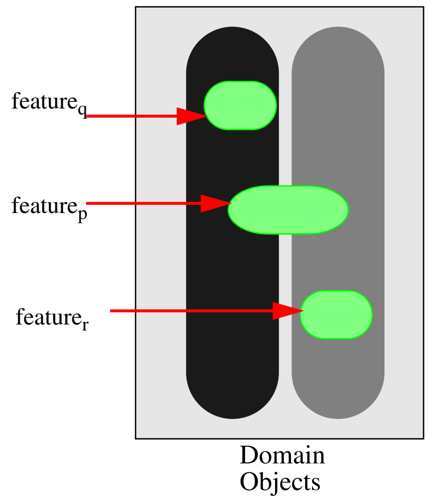

FIGURE 1. Control Information Embedded Inconsistently in Domain Objects

The explicit modeling of control information in terms of control objects provides us with more reusable objects as shown in Figure 2 below.

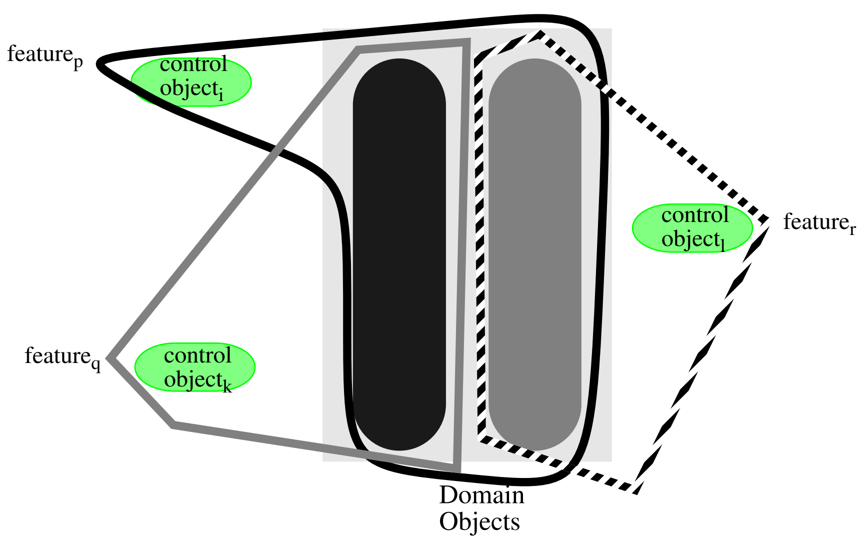

FIGURE 2. Control Information captured in Control Objects

Policy Decoupler

Problem

How can we create software for multiple customers, each with their own policies, when only one customer at a time is driving the requirements?

How do you limit the impact of a change in a policy for one customer, on policies for other customers, when the same domain objects are employed for multiple customers?

How can we provide support for multiple underlying engineering technologies on which a product can be based on, when each underlying technology enforces a different operational policy?

Traceability and maintainability of policy related information is not possible if policy information for a given customer or underlying engineer technology is scattered among several domain objects, and if they are not distinguishable from the inherent behavior of those objects. Reusability, reorganization or modification of domain objects is difficult due to the same reason.

Context

Software development of a turnkey, OO product, custom-tailored to several different customers.

The context is turnkey projects, with multiple customers that have their own requirements. However, the policies and preferences are seldom distinguished from the requirements (especially, by the customer) as the customer does not have to think in those terms.

Forces

Customers and Technologies: There exists a need to incorporate and support specific customer requirements, policies, preferences, and idiosyncrasies while trying to minimize retooling for each customer. There also exists a need to make the software work on multiple underlying technologies, each with their own operational policies. However, this must be achieved while minimizing retooling for different technologies. We would like to have the same product support different customers and engineering technologies and we do not like to create a different yet similar product if policies are not the same.

Modality of Development Efforts: The problem domain is handled for one customer at a time with the imperative of providing for multiple customers subsequently.

Reusability: The goal is to perform product development using reusable assets while ensuring that all customer and technology inputs including policies are modeled and incorporated. Traditional approaches end up embedding policies in domain objects, which is an unsatisfactory compromise that hampers reusability.

Nature of Policies: Policies can originate in customers preferences, operational modes determined by the underlying engineering technologies, and the operational environment. Policies can change, and need to be periodically updated and the relevant policies should be enforced for each customer, in each product line. This must be done without any major rework.

Solution

In the activity Domain Analysis, following Requirements Modeling, study the domain models and requirements across customers.
Create specific objects to encapsulate business and technology policy information.

The solution is the explicit modeling of policies employing policy objects. The consumption of these objects offer enforcement. Changes to policies only effects the associated policy object and allows for quick turn-around time when the policies change. This offers traceability of policy information, which makes it possible to ascertain the proper incorporation of policies in the product.

Separating policies from other information:

The first step is to analyze the deliverable from requirements (delivered from Requirement Analysis Phase) for possible policy information. It is not easy to distinguish policies from other requirements, and it takes multiple iterations to get them right. To the extent that the policies can be identified and modeled explicitly, the domain models will be reusable across customers.

The Requirements Analysis activities typically involve studying the end-user and customer requirements of one individual customer or environment and does not call for the study of similar environments or other customers at the same time. The Domain Analysis activity provides the opportunity to study the domain models and requirements across customers and underlying-technologies for the first time within the OO software development life-cycle. This implies that reusability must be attempted in Domain Analysis if we are to maximize it.

The modeling of policies:

Employ policy objects to encapsulate the customer specific policies. One policy object per customer functional domain is the preferred approach. These policy objects are identified and modeled by analyzing the requirements, possibly captured during a previous requirement analysis phase.

Employ policy objects to encapsulate the underlying engineering issues. Such issues could encompass underlying engineering technologies and operational policies for those technologies.

Resulting Context

We analyze customer-specific and engineering technology-specific policies and explicitly model the policies. The domain objects are now free from policy information and hence more reusable across customers and technologies. Traceability is enhanced.

Rationale

Customer and Technologies: Policy objects provide us with a placeholder for customer-specific and technology-specific policies. We can now model policies, however complex, with proper traceability. Policy objects make it unnecessary to retool the application from scratch to incorporate new customer policies. The same is true for newer technologies and policies for them.

Modality of Development Efforts: Domain Analysis using policy objects is the approach that allows us to model the needs of multiple customers without compromising the needs of each customer. Moreover, this makes it possible to provide for many, while focussing on one customer's product life-cycle.

Reusability: Explicit models of the policies of different customers / technologies makes it possible to properly incorporate the policies and related information that the customers provide. Products can then be developed by employing truly reusable assets (i.e. control and entity objects) and associated policy objects. Modality of Development Efforts: The problem domain is handled for one customer at a time with the imperative of providing for multiple customers subsequently.

Nature of Policies: The enforcement of policies is ensured by the consumption of the policy objects in our models. The policy objects and the design framework for policies help in providing proper support for policies.

Sketch

Figure 3 below depicts the case where customer policies are embedded in domain objects. Two customers ($customer_p$ and $customer_q$) share domain objects. However, because of the inclusion of policy information in the domain object, two different classes of domain objects may need to be supported, *domain object j* for $customer_q$ and *domain object j´* for $customer_p$.

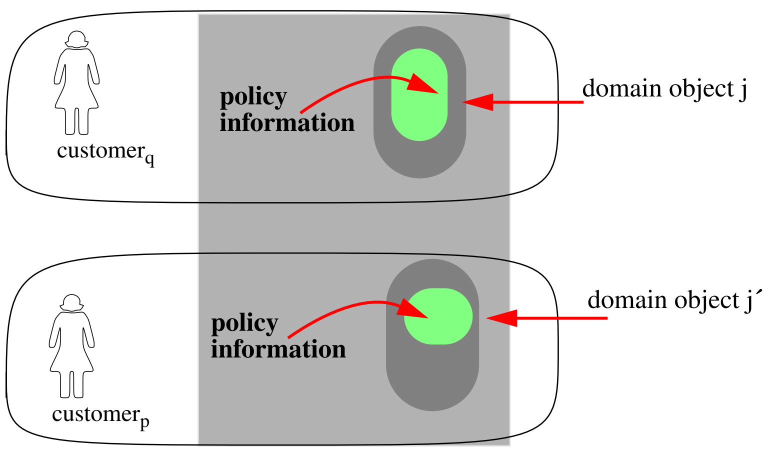

FIGURE 3. **Policy Information Embedded in Domain Objects**

In the general case, policy information may be embedded inconsistently across multiple domain objects. Figure 4 below depicts this scenario.

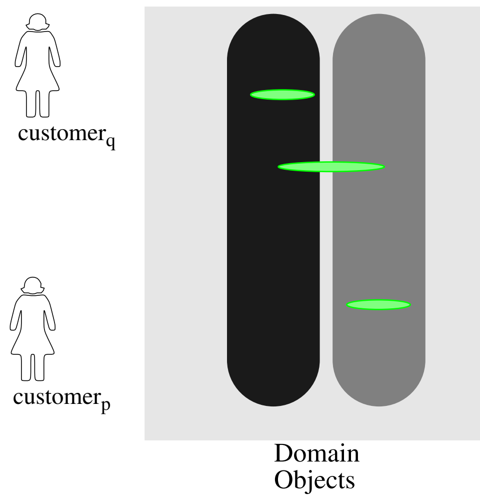

FIGURE 4. Policy Information Embedded Inconsistently in Domain Objects

The explicit modeling of policy information in terms of policy objects provides us with more reusable assets as shown in Figure 5 below.

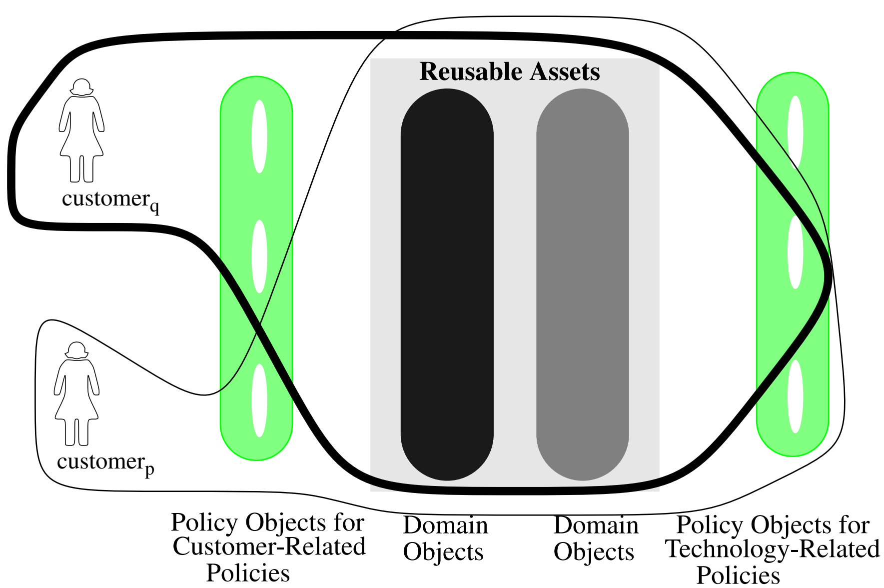

FIGURE 5. Policy Information Captured in Policy Objects

Example

The Example Domain: Audit

Audits are activities that are performed to verify that the information contained in one data resource conforms to information contained in a second (referenced) resource. The activities in this feature domain includes the scheduling of audits, the execution of audits, and the collection of responses from the audits, and the optional restoral of the database.

The Requirements

$customer_q$

The audits need to be performed when the load of the system is low or moderate, with inventory audits taking precedence over other kinds of audits. Audits should not be performed more than twice a day for inventory types of audits. In the first release, only inventory and alarm kinds of audits are permitted. When an audit is being executed, other activities that result in further audits should be limited. Equipment announcements are however allowed. They must later be verified.

$customer_p$

Audits must be performed for inventory and alarms. Cross-connect audits are possible for broadband systems. Alarm audits can be performed twice a day and cross-connect audits only once.

Model the Domain and Policies

Domain Object:

The Audit class provides audit related information. When an audit is to be executed, an object of the Auditor class is employed to coordinate the audit.

The Type_of_Audit class allows the consumer to specify the types of audits to be covered during an auditing activity

Policy Objects:

An exemplary policy class provides the following policy support:

Policy	Description
Types of audits	Audits can be for inventory, identifying available alarms, analyzing cross-connects, or for any combination of these. Other specialized audits may be necessary for some customers.
Restoral after an audit	Updating the value of a data item following an audit. Partial data restoral, or restoral on approval from User Interface.
Maximum number of audits in a day	The number of times an audit of a particular type is allowed to execute in a day is dependent on the scope of the audit, the type of audit, the state of the Access Equipment and the state of the Management application.

TABLE 1. **Sample Policy Class for the Audit Domain**

Policy	Description
Audits resulting in other audits	One audit can result in invoking another audit.
Dependency between audit types	The relationship between audit types - especially the order of precedence and the preemptive nature of audits.

TABLE 1. Sample Policy Class for the Audit Domain

The table bellow clearly shows how policy objects are to be instantiated for run time policy support.

Policy	customer$_q$	customer$_p$
Types of audits	inventory and alarms	inventory, alarms and cross connects
Restoral after an audit	none	none
Maximum number of audits in a day	inventory: 2 alarms: 5	inventory: not specified alarms: 2 cross connects: 1
Audits resulting in other audits	only for equipment announcement during audits	not allowed
Dependency between audit types	inventory is been conducted then alarms audits are disallowed	none

TABLE 2. Instantiating Policy Objects

References

[Booch, 94] Grady Booch, *Object-Oriented Analysis and Design*, The Benjamin/Cummings Publishing Company, Inc., 1995.

[Burdick, 93] Roberta G. Burdick, *Domain Analysis and Information Engineering: Promoting a Combined Attack on Stovepipe Systems*, Paramax Systems Corporation, WISR-6: The Workshop on Software Reuse, 1993.

[Dunn, 93] Michael F. Dunn and John C. Knight, *Creating and Using an Industrial Domain Model*, Industrial Software Technology and University of Virginia, WISR-6: The Workshop on Software Reuse, 1993.

[Guerrieri, 94] Ernesto Guerrieri, *Enhancing the Use of Domain Analysis*, Digital Equipment Corporation

[Jacobson, 95] Jacobson, Ivar, *Object-Oriented Software Engineering*, Addison-Wesley, 1995.

[Mohamed, 95] Walaa-Eldeen Mohamed, *Specification Reuse via Homogeneous Interpretation of Concepts*, University of North London, WISR7 in St. Charles, Illinois, August 28-30, 1995.

[Rumbaugh, 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.