Principles of Advanced Software Engineering: Variation-oriented Analysis, Design and Implementation

Ali Arsanjani, Maharishi University of Management

PRINCIPLES OF ADVANCED SOFTWARE ENGINEERING: VARIATION-ORIENTED ANALYSIS, DESIGN AND IMPLEMENTATION ................................................................. 1

CHAPTER ONE: REUSE LEVELS, CLUSTERS AND PATTERNS ........................................ 4

INTRODUCTION ........................................................................................................... 4

BASIC PRINCIPLES ................................................................................................... 5

Open-closed principle ............................................................................................... 5

Inversion of Control ................................................................................................ 5

VARIATION-ORIENTED DESIGN ............................................................................. 5

REUSE LEVELS .......................................................................................................... 7

PROGRAMMING TO INTERFACES ......................................................................... 9

Unprepared Change brings Entropy ......................................................................... 9

Open/closed Principle ............................................................................................. 9

Inheritance-based Polymorphism ............................................................................. 10

Composable Inheritance .......................................................................................... 11

ENCAPSULATING COLLABORATIONS ..................................................................... 12

PRINCIPLES OF COMPONENT DEVELOPMENT AND INTEGRATION .................... 13

INTRODUCTION AND DEFINITIONS ..................................................................... 13

THE CONCEPTS OF GRANULARITY AND CONTEXT ............................................. 13

RELATION WITH OTHER COMPONENTS: COMPONENTS HAVE MANNERS ......... 13

RELATIONSHIP WITH PATTERNS ......................................................................... 14

Evolving Designs ...................................................................................................... 14

RELATIONSHIP WITH FRAMEWORKS .................................................................. 14

A LIGHT-WEIGHT COMPONENT-BASED METHOD ................................................. 15

A Sample of Patterns .............................................................................................. 15

CHAPTER TWO: PATTERNS, FRAMEWORKS AND COMPONENT-BASED DEVELOPMENT 

CLUSTERS AND COMPONENTS: SMALLER FRAMEWORKS, LARGER-GRAINED OBJECTS ........................................................................................................ 17

COMPOUND PATTERNS .......................................................................................... 18

CHAPTER THREE: APPLYING PATTERNS ................................................................ 19

Introduction ............................................................................................................ 19

The Primordial meta-Pattern ................................................................................... 19

DOMAIN MODELING .................................................................................................. 20

USE CASES ................................................................................................................ 20

ITERATION ONE: SIMPLE CASE ............................................................................. 20

CLUSTER: THE PLAY IN WHICH OBJECTS ARE ACTORS AND THE THREAD OF EXECUTION IS THE SCRIPT .................................................................................. 20

Factories: “Create me an object of this type” ......................................................... 20

Clusters ...................................................................................................................... 20

OBSERVER: TELL ME WHEN THIS CHANGES ......................................................... 21

STRATEGY: COMPUTE A PRICE .............................................................................. 21

COMMAND: “HAVE SOMEONE DO THIS PLEASE” ............................................... 22

SUMMARY ROADMAP: ........................................................................................... 23

RULE OBJECT .......................................................................................................... 23
Chapter One: Reuse Levels, Clusters and Patterns

Introduction

Variation-oriented design was first mentioned in the now famous book entitled “Design Patterns-Elements of Reusable Object-oriented Software” by Gamma, Helm, Johnson and Vlissides.

The essence of this approach is to encapsulate that which varies within an abstract class or interface and to customize that abstract class with actual variation instances. But this is the way it is implemented. The implementation is based on the concept of a pattern.

Here is an excerpt from the Design Patterns book:

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [AIS+77, page x]. Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context.

In general, a pattern has four essential elements:

1. **The pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog.

2. **The problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

3. **The solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

4. **The consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system’s flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

Point of view affects one's interpretation of what is and isn't a pattern. One person's pattern can be another person's primitive building block. For this book we have concentrated on patterns at a certain level of abstraction. Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is. Nor are they complex, domain-specific designs for an entire application or subsystem. The design patterns in this book are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
**Basic Principles**

There are a number of good, sound software engineering principles which, when adhered to, bring about the fruition of the desirable qualities of a software product and process: maintainability, reusability, quality, reliability, performance, scalability, etc. These include high-cohesion/low-coupling between modules, separation of concerns, layering and tiering (such as database, business logic and user-interface tiers or layers), abstraction (using a more general concept than the one that immediately comes to mind for this particular instance), etc.

There are some principles, however, that are particularly apropos to our discussion of designing, building and deploying and maintaining reusable software components and assets in the context of variation-oriented analysis, design and implementation which enables and facilitates framework and component-based development.

**Open-closed principle**

“Software should be open to extension but closed to modification” (Bertrand Meyer). The idea is to enhance functionality by making non-intrusive changes. Intrusive changes are changes that alter code that has been previously written; changing inside of classes. Rather, interfaces should be changed/extended rather than the implementation of classes.

**Inversion of Control**

Also called the “Hollywood” principle, this refers to the thread of control that a framework takes upon itself to sustain, and call your classes that you have extended form the abstract classes in the framework. A Framework is a set of collaborating abstract classes with default behavior that you can accept, customize or over-ride based upon specific application needs.

**Variation-oriented Design**

This accomplished by variation-oriented design. Variation-oriented design is about separating the domain of discourse, the problem domain, the program into its changing and non-changing aspects.

<table>
<thead>
<tr>
<th>Principle 1: Separate the changing from the Changeless.</th>
</tr>
</thead>
</table>

In programming and design, separate interface from implementation (type from class). Program to interfaces rather than implementation.

Encapsulate what changes frequently by reifying it (make it an object; a class) in a class and creating an interface for the entity that is changing. The question is then: “How do I modify or extend the behavior or functionality of the entity once I have encapsulated it?” The answer is that we can then change the attributes and implementation of its behavior (all of its methods) inside the class implementation without changing its class signature (all of its method names, and their parameters).

“Then how can we extend and add or modify behavior if we can only change the implementation, not the interface to the class?” The answer is that we create an abstract class, which captures the key and common characteristics across a number of similar class instances, and derive the known classes from that abstract class. The classes thus inherit from the abstract class and implement it as a concrete class.

Then, whenever we want to add a new set of variations in behavior, we create a new class which is derived from the abstract class and implements its (new, added or modified) behavior in a slightly different way. Thus, you are making the changing, Changeless. This is the next principle.

| Principle 2: Introduce the second element. Introduce the Transcendental aspect of a class; the abstract class. |
Here is what the above will look like for each class that is incorporating the open-closed principle and variation-oriented design:
**Reuse Levels**

It is convenient to identify a number of levels of reuse based on the SCI principle of “life is found in Layers”.

| Principle 3: Life is found in layers; there are different levels of reuse. |

Consider the following levels of reuse or reusability:

<table>
<thead>
<tr>
<th>Base Class</th>
<th>A single class instantiated as an object; the lowest level of fine grained reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregation Hierarchy</td>
<td>A class containing one or more other classes; related with the aggregation/composition relationship. Example:</td>
</tr>
<tr>
<td>Inheritance Hierarchy</td>
<td>A set of classes related by the specialization-generalization relationship</td>
</tr>
<tr>
<td>Cluster</td>
<td>A set of, community of collaborating classes that typically have a façade, at least one mediator, composite which would typically have an abstract factory to produce objects within the cluster.</td>
</tr>
</tbody>
</table>
Figure 3

Framework

A set of classes related by a combination of inheritance and aggregation which have an interface, abstract classes with default implementations, that can be customized for a given context.

```
<<abstract>>
ServiceSubscriber

Agreement

Passenger
(from Logical View)
name

Ticket
(from Logical View)
TotalPrice

Flight
(source, destination, date, class, price)

1..*

<<abstract>>
ServiceProvider

Airline
(from Logical View)
mainHub
routes
```

Figure 4

Component

An encapsulation of a framework or an executable, composite mediating facade

Generic Architecture

The result of implementing a domain model

Pattern

A solution to a problem in a context; a design level construct that can be reused by customizing its participants to balance a set of tradeoffs for a given set of forces.

“Design patterns help you identify less-obvious abstractions and the objects that can capture them. For example, objects that represent a process or algorithm don’t occur in nature, yet they are a crucial part of flexible designs. The Strategy pattern describes how to implement interchangeable families of algorithms. The State pattern represents each state of an entity as an object. These objects are seldom found during analysis or even the early stages of design; they’re discovered later in the course of making a design more flexible and reusable.” [GHJV94]

Environment Interaction

Meta-knowledge

Knowledge about how to interact with a given environmental context

Technology transfer Knowledge

The knowledge of how to transfer technology into an organizational type

Methodology and Process Adoption Meta-knowledge

How to implement and customize a process or a method for a given context.
Programming to Interfaces

Unprepared Change brings Entropy

One of the strongest aspects of object-oriented programming is the ability to program to interfaces rather than implementation. This allows de-coupling and thus more reusability. For example, code has been written to implement an application. As the requirements evolve, we will gradually want to change the code, and we change the design.

Frequently, even if the code is a faithful reflection of the design, changing the code is a difficult and time-consuming task. Therefore, where possible, people rewrite code. If this is not economically feasible (i.e., in terms of person-hours and deadlines) then every effort is made to first understand the code, based, hopefully, on the design, and then to change the code to reflect new requirements. The more changes are applied, the more brittle and prone to “breaking the code”, the system becomes. This is because changes are introduced in places in the code where the original programmer had not envisioned; and had thus made some tacit assumptions that increase the chance of being violated with every intrusive change to the code. So every change makes the system a little bit more brittle, and entropy creeps in. Finally, the entire architecture has to be revamped in order for continued changes to be made based on new requirements.

Many approaches nowadays embrace change by planning for it. One example is Extreme Programming.¹

Open/closed Principle

So what is to be done? Bertrand Meyer proposed, in 1989, that “software should be open to extension but closed to modification.”² This is called the Open/Closed principle.

But how do we go about implementing the open/closed principle? Among other principles, in this book, we take the perspective of Variation-Oriented Analysis, Design and Programming in order to implement the Open/Closed principle. By encapsulating what changes frequently, we can modify it without having ripple effects traverse the breadth and depth of the application’s code. This is where we specifically use patterns (in most cases, design patterns) in order to come up with a good design. This is because we can still have a “bad design” even if we reify our changing code. We may make mistakes in modeling the situation and thus run the risk of making the system brittle and introducing entropy all over again, but in a different way.

Patterns help in selecting the “participants” of a design. In other words, patterns guide us to made better design decisions as to what objects and classes we choose to model our problem with. Then, in the pattern’s implementation section we find guidance on how to make tradeoffs to implement the pattern.

But beware, adding patterns randomly just because it may feel right to use a pattern does not mean that we should include the pattern in the design if it is really not called for. In other words, don’t over-use the notion of patterns. Apply them, like medicine; whenever necessary; but only carefully and in small doses.

One of first constructs that help in implementing the open/closed principle is to hide implementations behind interfaces. This goes back to David Parnas’ notion of information hiding.³ Therefore,

```
Program to interfaces rather than implementations.
```

² He also introduced the all-important notion of design by contract; pre-conditions, post-conditions and invariants. These are actually programming constructs in his Eiffel programming language.
³ “On the Criteria for Partitioning Systems” [Parnas??].
Inheritance-based Polymorphism

Instead of sending a message to an object that we know exists today, we may want to send a message to an interface that the (class) object then implements. This is done so that future additions of new classes find a “plug-point” or “extension point” behind the interface. So, for example, if we need to add a new type of service to our order entry system, which currently takes only long-distance telecommunication services, we can extend the program but not make intrusive changes if we have judiciously applied interfaces and have reified change points. Remember that change points were those areas in the analysis and design of the domain in which variations occurred. These variations are likely to multiply in the future, so we deal with this by reifying the variation into either an abstract class or an interface, as the language allows. We then implement the services we require today as concrete classes implementing interfaces. In the future, we can implement a new type of variation by reifying it as a class that ties into the plug-point we created by defining an interface.

<<Example>>

But there is a traditional set of forces that go in opposite directions: should we use inheritance or should we use composition (aggregation)? Inheritance produces hierarchies of classes at reuse level 2 that are more dependent on one another than an aggregation hierarchy of level 3.

Therefore,

| Favor Composition over inheritance. |

But we must balance forces between composition and inheritance. This is because inheritance gives the benefit of inheritance-based polymorphism.
But there is a pattern to be applied here.

**Composable Inheritance**

**Problem**
When should we use composition, when inheritance?

**Context**
Frameworks heavily use inheritance. Inheritance allows for reuse through extension. Frameworks are a set of abstract classes with a default collaboration between them. How can we make use of this default collaboration, the default implementations in abstract classes without derivation?

On the other hand, composition can be changed at run-time. Composition does not produce the brittle inheritance hierarchy problem which arises when you want to make changes to the parent nodes of an inheritance hierarchy, you will have to propagate changes down to its children and descendents with unknown effects.

**Forces**
- Inheritance –based polymorphism provides a powerful mechanism for unifying code and making it more maintainable and reusable by allowing you to add a new child that does a thing slightly differently rather than having to rewrite things.
- Composition is more powerful than inheritance because it provides runtime configurability, does not have a brittle inheritance hierarchy problem.

**Solution**
Use the following general cluster to combine the best aspects of inheritance with composition. Use only one level of inheritance. Start with an aspect of the client (the cluster, the entity object or whatever other level of reuse and granularity), factor out the Type commonality in an Interface; factor out the Class commonality in an abstract class, allow the Variations to be created as children of the common abstract class. The Abstract class can implement some default interaction sequences through Template Methods that concrete instances can over-ride to accommodate their unique behavioral needs. The Interface provides the Type: the valid operations that can be performed on the CommonType. To enforce the manners governing the way in which these operations are to be used, the Abstract Common Class provides default state, behavior, rules and meta-data are encapsulated as manners of the interface’s valid way of collaboration.

The Abstract Class then provides common behavior in the form of Template methods, Strategies or straight interactions between participants.

The concrete classes may implement the abstract methods to have the framework call them, in its own thread of execution.

The Client will compose an interface and at startup will be instantiated with a ConcreteVariant that will be appropriate for it based on configuration and setup criteria. The Client can then send messages defined in the Interface to the ConcreteVariant and expect correct behavior. The actual behavior and implementations can be customized for each type of variation by implementing a new ConcreteVariant. If the AbstractCommonClass changes or needs to altered in any way, a new AbstractCommonClass can implement the CommonType interface and have its own set of ConcreteVariants.
Another important principle of variation oriented design is:

Reify what changes

This allows you to make changes to the element with minimal ripple effect to the other aspects of the system.

**Encapsulating Collaborations**

There has been a positive movement towards recognizing that objects get as entangled as do procedural programs. But only in a different way; but entangled nonetheless.

Aspects were designed to address “cross-cutting” functionality that really did not belong in any class that was participating in a collaboration. Thus, an aspect would be introduced. Aspects are a step in the evolution of the object paradigm.

Nonetheless, we need to take a broader view. As purported in aspect-oriented programming, cross-cutting functionality should not be sprinkled into objects. They have other responsibilities. But consider for a moment that objects are just at reuse level 1 in the reuse hierarchy. As we go higher in reuse levels, we see a broader pattern of abstraction emerging. We need constructs that are larger grained than objects and classes. As a start, we can use inheritance hierarchies as we have in class libraries. We have also tried aggregation hierarchies and mixed them with inheritance hierarchies and classes, in class libraries and frameworks. But we have missed a step of reification in the midst of other fine-grained object perspective. We have missed the Cluster level. Yes, sub-systems, categories and recently, UML packages are ways of grouping classes into namespaces for the purposes of deployment and design. But the package disappears at run-time, and even in analysis (at very end of the spectrum), packages do not have relationships stronger than a relatively vague notion of “dependency” (see figure below).

<<figure on package dependency>>

Catalysis deals with this issue in a slightly different manner.

---

4 Aspect-oriented Programming, Gregor Kiczales, Xerox Parc.
5 Catalysis, D’Souza and Wills.
Principles of Component Development and Integration

Introduction and Definitions

Components can be developed from scratch, according to an enterprise analysis model. If we want to address the problem of designing, developing and deploying families of applications, Product Line Architectures must be utilized. This implies that Components must be developed after a domain analysis which aims to capture the common and variable aspects of the families.

A Component can mean different things to different people and indeed vary by context. But whatever it is, it carries within it the connotation of a composite “replaceable part” that can be “plugged-into” a context, or assembled with other components to form an application or part of an executable application. The notion of being a composite is in the sense of the design pattern Composite: being composed of finer-grained elements or parts that may themselves be composed of finer-grained components.

Here are some other definitions:

The analogy with hardware is clear. In fact, Brad Cox coined the term “Software IC” to describe the notion of a reusable software component.

Components, thus, have ports (like pins on the hardware IC), connectors that allow them to be connected to other components [cite other references]. But one of the most important concepts that is missing from the conceptual description of a component is that of “context”.

The Concepts of Granularity and Context

A Component Context is the “bread board” (as in electrical circuit testing), the domain (as in business domains), the background (as in a graphics application), the motherboard (as in a hardware context) on which the software IC is to be placed. Thus, if the context is that of a GUI system, a JavaBean or ActiveX component might come to mind. But in the context of an enterprise-wide architecture supporting telecommunication billing, insurance claims processing or banking transactions, a medium-grained component is needed to be “plugged-into” that Component Context.

This gives us an idea of how “coarse-grained” or large the component we would like to implement as an Enterprise Java Bean or a DCOM component (etc.) should be. It sort of defines the scope and size (scale) of the component.

See the notion of Reuse levels below.

Relation with Other Components: Components Have Manners

Next, it is essential to realize that components need not be single objects; in fact, most often they are not. In relation to objects, they encapsulate a cluster of objects (classes) and export a “cluster of types with manners”. Manners are the different ways in which you know how to behave in different circumstances and contexts. It is the external behavior that shows itself by virtue of a trigger, a set of rules governing the internal interaction of the encapsulated classes among themselves (Valid Object Interaction Sequences), along with the set of rules that the cluster displays to the outside world (Collaborations). Meta-level information is included within the notion of manners to account for a component being “queriable”; to so we can understand, dynamically, how the component behaves in a given context. The meta-information carries with it the information regarding the context and the VOIS and collaborations within that context. And remember there may be sets of these behaviors within contexts.
Thus, in order for autonomous clusters to be allowed to interact, they should be able to understand each other’s *rules of behavior*; not just interfaces (class signatures or types) but the ways in which I (as a component) can interact with another component within a given context.

In this sense, we need meta-information in the component and also need introspection capabilities to be able to query the component to identify itself and give us an idea of how it was designed to behave; its manners.

Therefore, we say: component have manners.

**Relationship with Patterns**

Patterns are solutions to a problem in a context. They describe the static and dynamic aspects of a solution, along with its tradeoffs, in a given set of contexts. In this sense, a component’s internal interactions between its constituent objects as well as its external collaborations can be described as patterns.

Frequently, compound patterns are used: patterns are combined to resolve and balance forces inside and outside (as the border between the component and its environment and context) the component.

The compound patterns that is used most often or seen most often in the design of a component can be expressed as: a component is an executable composite cluster that . We use a bridge outside the component to ensure that its interfaces and implementations can be independently varied. We may use a proxy to access certain service within the component.

But inside, the component is a cluster. The cluster is a community of interacting objects (classes); each object may be an “atomic” class or a cluster in its own right, and is thus a Composite. It will typically have a set of facades, which will allow single message-sends to invoke complex collaborations to achieve a result. Inside, there will probably be a mediator, which mediates the interactions of its constituent elements, so they can be varied without having to know about each other (dependency management).

**Evolving Designs**

Take a set of patterns that are seen to apply to the problem space; combine them into a set of compound patterns. Often, you will find some finer grained patterns that are frequently used together. For example, in meta-data and active object modeling there is extensive use of Property Lists with Type Objects. In the development of graphical user interfaces, Mediators mediate between GUI components so buttons don’t have to refer directly to textboxes and combo boxes, etc.; Observers (Listeners) trigger events and Mediators use Commands to implement operations; Commands then work with Strategies to interchange algorithms based on different contexts.

**Relationship with Frameworks**

There are various taxonomies of frameworks [Griss96+]. The one view we are taking is that of utility: there are “grape-vine” frameworks where the user has to use the entire framework (the whole grape-vine, with all the class hierarchies that are dependent (i.e., use) other class hierarchies within the framework) or nothing at all. These are the “all-or-none” type.

Another type of framework is the “small” framework. This is a cluster of interfaces and abstract classes with default behavior that supports inversion of control with various threads of execution within it. Thus, a component can encapsulate this kind of small framework and be utilized within applicable component contexts.

Patterns generate architectures. Architectures with hot-spots, extension-points and plug-points form frameworks (clusters of types (interfaces) with default implementation and control flows supplying hot-spots, plug-points and extension-points for customization within a given application context). Smaller frameworks can be encapsulated as components.
A Light-weight Component-based Method

Many of current object-oriented methods are criticized for not having adequate support for component-based development. The problem seems to stem from the fact that these methods (now having a common notation, the Unified Modeling Language) tend to view the world in terms of fine-grained objects to the exclusion of coarser-grained, larger component structures.

Indeed, the UML has the notion of a package to define a grouping of classes at design time. But the problem is that the proverbial seamlessness of object-oriented development (from analysis to design to implementation) is lost when design-time modeling concepts and constructs are lost at run-time. Thus, in order to solve this problem, let us define a cluster as a UML package that does not disappear at run-time.

Indeed, let us reify it (“make into an object or class”). The Cluster then becomes the controller or mediator between the set of collaborating classes that work together to solve part of a larger problem in the domain.

Step 1: The first step in the methodology is to select what level of reuse you are trying to achieve. Once you define the level of reuse, you can then concentrate on conducting analysis and design at that level of granularity. For example, you might typically want to start at the cluster level. Remember that a cluster is a group of collaborating classes that want to fulfill a common set of business objectives. They form a cohesive set of related functionality as a “community of classes”. This is called “Design by Cluster”.

Step 2: Identify the manners for that level of reuse. We will call the chosen level of reuse a component. “Manners” are the rules by which the component interacts within different contexts. First begin by specifying the rules that the component needs to abide by. Thus, you can use the rules (e.g. business rules) to arrive at the objects, methods and attributes that participate in a cluster. This can be remembered as “Clusters have Manners”.

Step 3: Identify individual, finer-grained objects that are collaborating. Specify the methods on the objects (classes). Identify the attributes.

Step 4: Define Valid Objects Interaction sequences based on the “manners” previously defined.

A Sample of Patterns

Strategy
Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Structure

Applicability

Use the Strategy pattern when

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.

- you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.

- an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.

- a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.
Chapter Two: Patterns, Frameworks and Component-based Development

Clusters and Components: Smaller Frameworks, Larger-grained Objects

Use the Primordial Pattern (Samhitta of Rishi, Devata and Chhandas)\(^6\) to determine relationships and roles between participants in a meta-domain pattern.

We start out with three components or clusters derived from the Primordial Pattern: Service Provider (Rishi), Party (Chhandas), Business Agreement Transaction (Devata). Each Cluster contains a set of collaborating Types. These Types collaborate internally according to a set of rules. These rules are called “Valid (Type) Object Interaction Sequences”.

Types have strongly defined operations (classes have messages); but weakly defined protocols or dependencies:

\[
\text{A Cluster is a Composite, Mediating Façade (see [Gamma+95]). Please refer to the explanation on Reuse Levels in Chapter One}\(^7\).}
\]

Here is an example of a Cluster for Orders. An Order contains OrderLineItems. Each OrderLineItem is associated with a Product that is being ordered. Each Product has a Price associated with it which is specified on the OrderLineItem:

<table>
<thead>
<tr>
<th>Seq</th>
<th>Product Description</th>
<th>Quantity</th>
<th>Unit Price</th>
<th>Unit Discount</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Disk Drive</td>
<td>3</td>
<td>30.00</td>
<td></td>
<td>90.00</td>
</tr>
<tr>
<td>2</td>
<td>Monitor</td>
<td>1</td>
<td>480.00</td>
<td></td>
<td>480.00</td>
</tr>
<tr>
<td>3</td>
<td>Mouse</td>
<td>2</td>
<td>21.00</td>
<td></td>
<td>41.00</td>
</tr>
</tbody>
</table>

\[
\text{Total 511.00 Discount 10\%= 51.10 Amount Due } \$459.99
\]

The notion of a Cluster as described in the previous chapter, as a larger-grained construct above and beyond the notion of a package is important to understand. Here is the corresponding class diagram representing a Cluster for this scenario:

---

\(^6\) The “Primordial Pattern” is an application of Maharishi’s Vedic Science to describe a fundamental pattern in software engineering. Patterns that are in some sense similar or special cases of the Primordial Pattern are Event Logging [Coad96], Association Object or Event Reification.

\(^7\) [Chapter One is document Vod.doc]
The Order Cluster implements the Cluster “Class” in this implementation instance, as the notion of “Cluster” needs to be mapped onto current programming language constructs.

Notice that there are different ways of grouping a set of Types or Classes into a Cluster.

In this particular Order Cluster, Price and Product are Interfaces. Thus, the Cluster itself may be represented as: Order Cluster = {Order, OrderLine, OrderHeader, OrderFooter}, the connectors or ports that OrderCluster has are a Price port and a Product port.

This is analogous to the “pins of an IC”. The Hardware IC, is the Order Cluster (or component). The Order Cluster has two plug-points or ports or, in the hardware analogy, two “pins”: Product and Price.

But Order Cluster can participate in a larger composition and customization of Cluster, as in the Service Provider Meta-Domain Pattern [Arsanjani99]:

Here a Service Provider is entering into a Business Transaction with a Party. The Retailer is a Service Provider (extends Service Provider), and Liable Party is a special kind of Party. The Order is a special kind of Business Agreement/Transaction.

Thus, the domain pattern Service Provider is said to be “customized” by Retailer-Liable Party-Order. Order, being a cluster participates in a composition of components in an assembly oriented paradigm of software integration and assembly via components.

**Compound Patterns**

Compound patterns are the result of combining a number of patterns in consistently repeatable combinations. Examples may include using Mediator, Command and Observer for user-interface controls and “widgets” to wait for an event, notify the appropriate Party (control), have Mediator send a Command to the recipient widget without all the widgets having to know about each other.

Another example is the Observer Party Account compound pattern. [See Article in Communications of the ACM].
Chapter Three: Applying Patterns

Introduction
Inspired by Maharishi’s Vedic Science, we try and discover the Primordial Pattern in each software engineering situation in order to factor out the various participants and thus be able to arrive at better and more flexible design decisions.

Identify Rishi (actor, Knower), Devata (Action, Knowing), Chhandas (Acted on, Known) and their Samhitta (Wholeness value).

To illustrate the application of the Primordial Pattern of Variation Oriented Design as inspired by Maharishi’s Vedic Science, here are some Java programs and their pattern design decisions.

The Primordial meta-Pattern

Context
This is a pattern on discovering and applying patterns, design principles and design decisions, selecting participants in a pattern and for help in domain modeling.

Problem
How do you divvy up the word of objects from a conceptual standpoint? How do you categorize things at the most fundamental level and carry forward the analogy to designing (for example) software systems?

Solution
Use Maharishi’s Vedic Science’s interpretation of Samhitta of Rishi, Devata and Chhandas. Classify software system objects and participants in patterns using this taxonomy.

Forces
You can categorize the world in many ways? Which one should I use? Which one should I use for this problem at hand?

Resulting Context
Now, you have a conceptual super-structure to tack ideas upon, to provide a scaffolding for your thoughts and categorizations. You can start your categorization using this fundamental truth about the nature of consciousness as expounded by Maharishi. These categorizations are dynamic and can change; the pattern is a meta-pattern on how to mine, choose, apply and learn more about patterns that are abstract recurrences of natural law in a small context. They are efficient ways of doing things that master craftsmen have handed down in the software community.
**Domain Modeling**

“…is about knowing about things without having to tie down that knowledge to a particular instance…”

**Use Cases**

We want to be able to:

- Price a product
  - Get its price from a catalog
  - See if discounts or promotions apply
- Select Products
  - See the price, picture and description of products (usually from the catalog);
  - But there may be special offers or promotions or discounts (“New Year Sale”)
- Request/Place an Order
  - Consisting of Items or Order Line Items
    - Each Line Item has Product, a Price, a Quantity associated with it.
    - It also has a subtotal price of what that product will cost before deductions,
    - and the price after all deductions have been applied (discounts, promotions (“this product is $5 off this month only”) and taxes
- Allow Customers to Pay for the Order

**Iteration One: Simple Case**

Let’s start simple and evolve gradually into a more complex example by composing solutions with previous ones. Let’s start with a button. Here are some questions in the mind of a novice developer. We will be using this question/answer format throughout.

“How do I get a button to know I have clicked it? Then how do I get it to do something for me? If I add more useful and realistic GUI widgets like text boxes, lists, labels, etc., how does my application have them interact?”

**Cluster: The Play in which Objects are Actors and the Thread of Execution is the Script**

Factory: “Create me an object of this type”

How do we create buttons? Who creates Buttons? We use a Factory (Abstract Factory or Factory Method or just a plain old Factory) to create objects.

**Clusters**

We use a cluster to aggregate the objects that are collaborating as a community of objects.

Someone in the program (usually, traditionally the “main” program knows how to create things quite magically out of the blue. Let’s reify this as a Cluster of interacting objects (Cluster for short). The Cluster will “know about the participants in all collaborations”. There are many ways in which objects within the

---

8 Ali Arsanjani’s Mentor/Disciple Model of Teaching based on Upanishad’s way of teaching through mentoring.
community need to interact in order to fulfill a step of a use-case, and set the stage for the next step. These sets of collaborations, or how the objects interact changes.

We reify the following knowledge within a Cluster:

- Knowledge about the collaborations,
- The participants of the collaborations and rules governing these collaborations:
- The sequence of steps of the collaboration, their pre- and post-conditions,
- The dependencies on other participants in terms of pre-requisite and co-requisite types of rules and
- General business rules that apply to the domain in general.
- Meta-information: or information about the way sin which this cluster can collaborate that can be made accessible to other components at run-time in a dynamic, self-configurable, reflective fashion, allowing dynamic negotiation and interaction in various business-to-business like scenarios (dealt with later).

**Observer: Tell me when this changes**

Start with a button and text box. When we click on the button, we want the button to be notified we have clicked it and we want it to do something for us. Java provides the direct language implementation of Event Handling via Listeners, a slight variation of the Observer design pattern.

<< Code and GUI>> << pull out difference between Observer and Listener>>

Another example. We have a combo box containing all of Sanata Claus’s reindeers: Dasher, Dancer, Prancer, Vixen, Comet and Cupid, Donner, Blitzen, and Rudolph. When ever Sanata Chhoses a Reindeer to lead his sleigh, we want to notify the reindeer by printing his name of a label:

**SCI:** This is the Samhitta value. It ties together the Actor (User), Action (mouse click) and the Acted on (Button) in order for the Acted on to do something.

**Strategy: Compute a Price:**

Strategy asks: “How should I compute the price? Which pricing algorithm should I use?”

Strategy Computes a price from the different ways you can compute prices, based on promotions, discounts, etc.

Here are questions a user may ask you or confide in you when you are trying to figure out requirements: “Can I change the way I price things very fast?”

The answer is yes if you apply a **Strategy**.

Scenario: You are a e-business owner who sells products over the Internet. You would like to have a shopping cart so people can select products and submit an Order to you. People select Products. The Products description and Price and may be a picture is all they want to see. They can then choose a Product and put them it into a shopping cart. If you had a Computer Hardware store, and would like to have people select components they want inside their computer, you would let then select parts and add them to a larger collection, the computer they are purchasing. For this we will use the **Composite** design pattern that allows us to treat a part and whole (a Product and a Product that contains other products) in a uniform manner.

---

9 See Chapter Eight: Dynamic Business to Business Interactions Using Components and Grammar-Oriented Object Design
So you go through the list of items using a Visitor. Visitor uses an Iterator to traverse the nodes of a Composite pattern, it then performs calculations on the items and returns results.

But let us break the problem into manageable pieces. Let’s start with the issue of computing a price. Let’s say you know what the product is, and can therefore ask for its price. The Order Line Item knows how many (quantity) of the Product the customer wants. Now you would like to compute the price of the Line Item along with any discounts or promotions you may have running at the time.

You would like to input a price of a product and have the discounts, taxes and

Object Manners:

<table>
<thead>
<tr>
<th>If lookup price then apply pricer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pricer applies product price, discounts, taxes,</td>
</tr>
</tbody>
</table>

So before computing anything, we need to check some business rules and see if some conditions apply or not. Then we can go ahead and perform the computation, which is often implemented as a method on a class.

**Command: “Have Someone Do This Please”**

We apply the first principle of variation-oriented design, which reminds us to reify things if they are going to be changing a lot. This time let’s wrap the method or function in an object, since it will change a lot, but may not know what its other forms will be. This we call a Command. (Also called a Functor, a function that has been wrapped in a class with a method).

```java
Interface Computation {
    void performComputation(InputParameters inp, OutputParameters outp); // general case
    float performComputation(float inp, float outp); // simple case
}
```

So we ended up writing an interface instead of just a function or a method. So, what ever class in our problem space would like to “do something” or perform a computation, it will implement this interface. For example:

```java
Class CalculatePrice implements Computation{
    // … other stuff
    void performComputation(InputParameters inp, OutputParameters outp){
    }
}
```

Instead of calling the performance of the action a Computation, the Command Pattern calls it execute():

```java
Interface Command {
    Void execute();
    /** just execute, but what if you wanted to return a value of the execution such as the results of a computation? Use a double dispatch (See section on Double-dispatch), or a call-back (See Implementing call-backs in Java). */
}
```
Or, what if we wanted to give the function execute some parameters to perform the computation with? Can we do that?

Summary Roadmap:

**Observer** (button and text field; button is listening for a “click on it to execute”) ➔

**Command** (Button implements a Command (e.g., compute price input in this textbox)) ➔

**Strategy**: now what if we want to change the computation? Use a Strategy. A given Command may contain a Strategy or the Class implementing Command may have a Strategy within itself, without Command knowing about this.

Questions:

But what if we had to maintain State in the user-interface? The use-case “Allow Customer to Pay” has variant Methods of Payment. One of which is a Credit Card. Another is by Check. Another is through direct debit where your account is debited directly. But these MOPs have different information they need to capture: the expiration date and credit card number will not apply to the check MOP.

So different MOPs need different informational items to be captured for them. So applying the principle of VOD, RWC (Reify What Changes [A Lot, or has many Variations that will/may/are apt to be changed]), we reify these as a set of fields and the rules that apply among them (CC# is checked via an algorithm, date must be after today’s date, must have a CC# and a expiration date, must be valid CC#, etc.) These business rules are encapsulated in a **Rule Object**.

**Rule Object**

**Aka**

Rules as Objects; Objects have Manners

<table>
<thead>
<tr>
<th>Modified</th>
<th>Author(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 1998</td>
<td>AA</td>
<td>First Draft; although its been around a long time, I finally decided to write it.</td>
</tr>
<tr>
<td>July 23, 1998</td>
<td></td>
<td></td>
</tr>
<tr>
<td>May 7, 1999</td>
<td></td>
<td>Minor revisions</td>
</tr>
<tr>
<td>Dec 24, 1999</td>
<td></td>
<td>Putting various versions together</td>
</tr>
</tbody>
</table>

**Intent**

To separate and modularize the creation and alteration of rules from the user-interface, database and application layers.
Business Objects are composed of interfaces plus the rules governing that business object as specified by the corporate context in which the rule can be utilized.

**Problem / Motivation**

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

Business rules are expected to change more frequently than the rest of the business object. Rules are frequently implemented within the context of the methods of a business object. They refer to other business objects from within a another business object’s method, creating a web of dependencies that are typically implicit. Thus, changing one business rule would impact a set of objects dependent upon that rule.

**Rule Object: Objects have Manners**

Typically, an object does not execute a method blindly:

```
<< create head, torso, (forgot to create legs), run() ➔ boom! Error, exception >> << cite Arian crash
[Sidebar: Dangers of Objects]>>
```

Before doing something, executing a method after a message send has been received, an object will typically check to see if its State and possible input parameter or global information (e.g., in a database or a state object, or a Session Object, or some other object’s state) is valid or not. Then the object will execute the method. Otherwise, it will throw an exception or provide some means of notifying the caller that the operation has failed or cannot be done due to an inconsistent State of the object or other objects it knows about (has references or access to).

Let’s start with Strategy; being able to compute something

Here is a simple GUI:

```
<<insert picture>>
```

Explanation: There is a button (Compute), a text box in which we enter a value and the label displays the result of the computation based on the value. This is an example of a simple $y = f(x)$ fountain in mathematics.

This is an example of using an algorithm to compute the total price of a product. But say you wanted to be able to choose which algorithm form a set of related algorithms, you wanted to apply. This would require a pattern called Strategy. Strategy allows you to interchange which algorithm you are using, in a family of algorithms.

**Checking Rules**

Next, imagine you were doing a conversion of Celsius into Farenheit and you wanted to be able to go both ways : $F \rightarrow C$ and $C \rightarrow F$. Now we have two text boxes (one for C and one for F) and one button (to start the computation). You would enter the number into one text box and click on the Compute button. The process would check to see whether there was a value in the other “converted” text box. If there is a value, is it a valid conversion? If not (or if it is blank (no value in it)), reconvert, other wise, give a message saying already converted.
So you need to do checks before doing actions. This is a trivial example of a more important problem in larger software systems: objects usually do not carry out actions (execute methods) “mindlessly”; some condition is usually checked and if those series of conditions apply then you can perform the action. Here is a lengthier example from the telecommunications domain.

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.

**Context**

Rules should be first-class constructs of the object paradigm.

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 held separately.

Instead of rules being expressed as logic within each business object, rules are collated together into their own objects.

Having a central repository of rules 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.

**Forces**

◊ Rules are present at different layers (UI, App, DB)
◊ Business processes have a set of fixed rules for any given business domain; and they have a rapidly expanding and changing set of rules that may change day to day.
◊ Rules need to be created by and visible to management.
◊ Rules and rule changes need to be visible to programmers
◊ Rules should be first-class constructs of the object paradigm.

◊ 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 held separately.

◊ Instead of rules being expressed as logic within each business object, rules are collated together into their own objects.
Solution

Structure
Compose Rules within Business Objects, yet encapsulate the Rules in a Rule Object so that they can be changed through well-defined interfaces, rather than having to make intrusive changes; without affecting the rest of the business object.

The following is a static representation of Rule Object:

Collaborations
<<Insert collaborations here>>

Rule Design
The simplest rule is the “if condition then action” logical construct. But we need to extricate these chains of ifs and cases from within the application layer code; to enable making rapid alterations to their rule structures, since they will be changing so much. So according to the principle of variation-oriented design, whatever tends to change on a frequent basis would be worth encapsulating in a class. Thus, we are insulating the system as a whole from ripple effects of individual objects and sets of objects (clusters); to whatever extent is possible.

Field Validation
A rule is an instance of a policy. So we need to categorize and distinguish these. The simplest variety is the field validator. Frequently this is implemented directly in the client user-interface. In Java 1.1, using the delegation mechanism, the observer pattern is used to determine event notification.

Validators can be implemented using Strategy and in conjunction with Command, if necessary. If you want to decouple the handler of the rule checking mechanism from the class itself, you would use Command. In order to have various methods of checking rules based on external criteria, you would use Strategy to encapsulate the rule checking mechanism and make all the ways in which you can check the rule(s) under given legal constraints or company policies, etc., into a family of algorithms.
Policy

Policies are sets of rules that need to be applied together according to business goal fulfillment strategies. Thus, a policy can be implemented as a Template method sequence of rule checks, and invocations in sequence. Any one, which needs to be changed, is implemented as a hot-spot. This would contain a Composite condition and a Composite action (each action is a Command). The default behavior at the abstract class level would be to iterate through the conditions and if each failed, define an exception for it. If all were satisfied (my Assessor pattern; similar to Command, but only assessing or evaluating something rather than executing an action)

User-Interface Perspective on Rule-related Terminology

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 fields must be validated
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 in the middle tier; business rules and policies
5. Checking with rules in the database layer. Should not be necessary once 3 is done.

Business Perspective on Rule-related Terminology

A validator is a UI-level or app-level strategy that validates a given simple or composite data item

A rule is a class consisting of a Composite condition (Assessor) and Composite action

A CompoundRule has two implementations:
- as a Composite of Rules (simple rules);
- as a Template Method

A policy is a template method or set of composite rules that have a set sequence.

Our sample apps will be written according to the above rules as a test of the efficiency or lack thereof of the implementation mechanism for our rules.

The intent is to be able to store then rules separately in a Rule Repository supplied by our framework. The rule repository will be a set of rules that are added for each system (e.g., MCRS, PhoneBook, etc.) So we will have a web- interface (java) to browse through the rules for each system using the Category Domain Pattern: SuperCategory = “Application System (e.g., MCRS, PhoneBook); Category = “Type of Rule (e.g, UI rule, app rule, db rule, validator, policy, compound rule, customer rule, order rule, services rule, billing info rule, etc.)sub-category= “Rule Name”

Consequences/Resulting Context

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

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.

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 Considerations

First Try

Here is a small example of implementing some simple rule objects. We want to construct a class that can be reused in many contexts.

The following is an example of how people might ordinarily code rules. In the next section, we will give the Rule Object version.

class AlisRules{
    Account acct; // the Liable Account
    Errors errors; // List of errors we encountered when applying rules
    Hashtable data; //
    Properties alisListOfErrors;
    AlisRules(Account a) {
        acct = a;
        this.errors = errors;
        data = acct.data();
        alisListOfErrors = new Properties()   //fieldname-errorMsg
    }
    public boolean checkAlisRules() {
        ... rules...
        ...with each error, add to your Properties a fieldname/msg pair...
    }
    public Properties getAlisErrors() {
        return alisListOfErrors;
    }
    //when you need to read a data value from your hashtable data of data from account,
    use this method
    private String getDataValue(String fieldname) {
        return acct.getDataValue(fieldname);
    }
}

These rules are to be run after submit but before client rules are run and before application data is set by Customer Care (some of these rules depend on this sequential placement)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 1. | If isNewAccount() the following fields should have value "":
|   | "TGActNo", "RepName", "RepCompany", "CCRejectReason", "CCRejectDate", "EstInUsage",
|   | "EstCardUsage", "EstOutUsage"
| 2. | If isNewAccount(), "TaxExempt" should have value "NNN"
| 3. | If isNewAccount(), "Status" should have value "New"
4. If isNewAccount() and "CustType" has value "D" then "MonthNoMin" should have value ""

5. If isNewAccount() and "CustType" has value "I" then "LEC" should have value ""

6. The allowed values for "ResBus" are "Business" and "Residential"


8. The allowed values for "BillOnMedia" are "Disk", "CD", "Internet Access"

9. The allowed values for "BillPayMethod" are "", "Credit Card", "Bank Debit", "Check"

10. The allowed values for "InvoiceDelivery" are "Mail", "Fax and Mail", "Email"

11. The allowed values for "SecurityType" are "", "Cash Deposit", "Credit Card", "Letter of Credit", "Supporting Credit Documents"

12. The allowed values for "BankAcctType" are "", "Checking", "Savings"

13. (cd is an instance of common data)

   The allowed values for "Language" are obtained from Vector languages = cd.languages()
   The allowed values for "CompanyCountry" are obtained from Vector companyCountryName = cd.companyCountries()
   The allowed values for "BillCountry" are obtained from Vector billctrynames = cd.companyCountries()
   The allowed values for "BankCountry" are obtained from Vector bankcountries = cd.bankCountries();
   The allowed values for "PaymtCurrency" are obtained from Vector currency = cd.currencies();

14. If "CompanyState" has any lower case letters, these should be changed to upper case letters.
    Likewise, if "BillState" has any lower case letters, these should be changed to upper case letters.

15. If "LOADate" contains either a "." or a "/", these characters should be changed to "-".

16.

17.

Here is a simple solution:
import java.util.*;

class AlisRules{
    AlisRules(Account a) {
        // acct = a;
        // this.errors = errors;
        // data = acct.data();
        data = new Hashtable();
        alisListOfErrors = new Properties();
        // fieldname-errorMsg
        fillValidValues();
    }

    public boolean checkAlisRules() {
        //  ... rules...
        //  ...with each error, add to your Properties
        a fieldname/msg pair...
        alisListOfErrors.clear();
        checkNewAccountRules();
        checkValidValues();
        setDataValue("CompanyState",getDataValue("CompanyState ").toUpperCase());
        setDataValue("BillState",getDataValue("BillState").toUpperCase());
        setDataValue("LOADate",getDataValue("LOADate").replace('.', '-'));
        setDataValue("LOADate",getDataValue("LOADate").replace('/', '-'));
        if (alisListOfErrors.isEmpty())
            return true;
        else
            return false;
    }

    Properties getAlisErrors() {
        return alisListOfErrors;
    }

    private void fillValidValues() {
        Vector values = null;
        // valid values for <<ResBus>>
        values = new Vector();
        values.addElement("Business");
        validValues.put("ResBus",values);
        // valid values for <<MonthNoMin>>
        values = new Vector();
        values.addElement("Jan");
        values.addElement("Feb");
        values.addElement("Mar");
        values.addElement("Apr");
        values.addElement("May");
        values.addElement("Jun");
        values.addElement("Jul");
        values.addElement("Aug");
        values.addElement("Sep");
        values.addElement("Oct");
        values.addElement("Nov");
        values.addElement("Dec");
        validValues.put("MonthNoMin",values);
        // valid values for <<MonthNoMax>>
        values = new Vector();
        values.addElement("Jan");
        values.addElement("Feb");
        values.addElement("Mar");
        values.addElement("Apr");
        values.addElement("May");
        values.addElement("Jun");
        values.addElement("Jul");
        values.addElement("Aug");
        values.addElement("Sep");
        values.addElement("Oct");
        values.addElement("Nov");
        values.addElement("Dec");
        validValues.put("MonthNoMax",values);
        // valid values for <<BankAcctType>>
        values = new Vector();
        values.addElement("Checking");
        validValues.put("BankAcctType",values);
        // valid values for <<PaymtCurrency>>
        values = new Vector();
        validValues.put("PaymtCurrency",values);
        // valid values for <<SecurityType>>
        validValues.put("SecurityType",cd.languages());
        // valid values for <<CompanyCountry>>
        validValues.put("CompanyCountry",cd.companyCountries());
        // valid values for <<BankCountry>>
        validValues.put("BankCountry",cd.bankCountries());
        // valid values for <<Language>>
        validValues.put("Language",cd.languages());
        // valid values for <<InvoiceDelivery>>
        validValues.put("InvoiceDelivery",values);
        // valid values for <<BillingCountry>>
        validValues.put("BillingCountry",values);
        // valid values for <<BankAcctType>>
        validValues.put("BankAcctType",values);
        // valid values for <<Language>>
        validValues.put("Language",cd.languages());
        // valid values for <<InvoiceDelivery>>
        validValues.put("InvoiceDelivery",values);
        // valid values for <<BillingCountry>>
        validValues.put("BillingCountry",values);
        // valid values for <<CompanyCountry>>
        validValues.put("CompanyCountry",cd.companyCountries());
    }

    private String getErrorMessageFor(String field_name) {
        // return a constant error message
        return "An Error String";
    }

    private void addErrorFor(String field_name) {
        alisListOfErrors.put(field_name,
                getErrorMessageFor(field_name));
    }

    private boolean hasValidValue(String field_name) {
        Vector values = (Vector)
                validValues.get(field_name);
        if (values!=null)
            return values.contains(getDataValue(field_name));
        return false;
    }
}
private void checkNewAccountRules()
{
    if (!isNewAccount())
        return;
    if (!getDataValue("Secured").equals(""))
        addErrorFor("Secured");
    if (!getDataValue("SecurityCCType").equals(""))
        addErrorFor("SecurityCCType");
    if (!getDataValue("CCType").equals(""))
        addErrorFor("CCType");
    if (!getDataValue("CCAuthAmount").equals(""))
        addErrorFor("CCAuthAmount");
    if (!getDataValue("Usage").equals(""))
        addErrorFor("Usage");
    if (!getDataValue("ServiceNums").equals(""))
        addErrorFor("ServiceNums");
    if (!getDataValue("TGAcctNo").equals(""))
        addErrorFor("TGAcctNo");
    if (!getDataValue("RepName").equals(""))
        addErrorFor("RepName");
    if (!getDataValue("RepCompany").equals(""))
        addErrorFor("RepCompany");
    if (!getDataValue("CCRejectReason").equals(""))
        addErrorFor("CCRejectReason");
    if (!getDataValue("CCRejectDate").equals(""))
        addErrorFor("CCRejectDate");
    if (!getDataValue("EstInUsage").equals(""))
        addErrorFor("EstInUsage");
    if (!getDataValue("EstCardUsage").equals(""))
        addErrorFor("EstCardUsage");
    if (!getDataValue("EstOutUsage").equals(""))
        addErrorFor("EstOutUsage");
    if (!getDataValue("TaxExempt").equals(""))
        addErrorFor("TaxExempt");
    if (!getDataValue("Status").equals("New"))
        addErrorFor("Status");
    if (getDataValue("CustType").equals("D"))
    {
        if (!getDataValue("MonthNoMin").equals(""))
            addErrorFor("MonthNoMin");
        if (getDataValue("LEC").equals(""))
            addErrorFor("LEC");
    }
    private void checkValidValues()
    {
        if (!hasValidValue("ResBus"))
            addErrorFor("ResBus");
        if (!hasValidValue("MonthNoMin"))
            addErrorFor("MonthNoMin");
        if (!hasValidValue("BillOnDisk"))
            addErrorFor("BillOnDisk");
        if (!hasValidValue("BillPayMethod"))
            addErrorFor("BillPayMethod");
        if (!hasValidValue("InvoiceDelivery"))
            addErrorFor("InvoiceDelivery");
        if (!hasValidValue("SecurityType"))
            addErrorFor("SecurityType");
        if (!hasValidValue("BankAcctType"))
            addErrorFor("BankAcctType");
        if (!hasValidValue("Language"))
            addErrorFor("Language");
        if (!hasValidValue("CompanyCountry"))
            addErrorFor("CompanyCountry");
        if (!hasValidValue("BillCountry"))
            addErrorFor("BillCountry");
        if (!hasValidValue("BankCountry"))
            addErrorFor("BankCountry");
        if (!hasValidValue("PaymtCurrency"))
            addErrorFor("PaymtCurrency");
    }
    // Attributes:
    private Account acct;
    private Errors errors;
    private Hashtable data;
    private Properties aliasListErrors;
    private Hashtable validValues = new Hashtable();
Second Try: Implementing the Example using Rule Object

Here we will give an example of a possible implementation of the above requirements using some patterns; specifically focusing on showing how one would implement Rule Objects.

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 Assesor 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.

If the Rule Object is for a Field in an Object, specifically a GUI object, you might consider using the lighter version which is a Validator. The Validator essentially is like a Command which may contain Strategies. One kind of Strategy is a “Non-NullValue” Strategy, or a Mandatory Field Strategy. Another kind may be a Rang Strategy implying that the value within the field must be within a given range of values. If these values are non-discrete (like Age 1-120), or date (12/1/1900 and 12/1/2010), then primitive types can be used to implement the Strategy. If the values are discrete, such as in the example above (The allowed values for "BillPayMethod" are ","Credit Card","Electronic Funds Transefr" or "Check"), then these values are either the Validator, or an Assessor keeps these values and thus a CompoundRule must be used instead of a SimpleRule (Validator).

Here is how an assessor encapsulates a discrete range of values:

<<Insert example>>

Here is how the previous program would use Rule Objects in conjunction with other patterns to do its work.

<<insert code here>>
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”. 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.

Known Uses

The Rule Object has been used on several projects by various teams. Domains include Telecommunications (Customer Care and Billing), Healthcare, Automotive, Higher Education industries.

Rule Object has been used in the implementation of the Java Business Frameworks [Arsanjani99] and in various similar forms in other projects; e.g., IBM San Francisco, Policy Common Business Object and Pattern.

References

[Arsanjani89] Grammar-Oriented Programming

[Arsanjani98] The Assessor Design Pattern


[GHJV95] Design Patterns: Elements of Reusable Object-Oriented Software

[Riehle98] Composite Patterns. OOPSLA ’98 Proceedings
A Pattern Language for Dynamic and Flexible Rule Construction

Rule Object, discussed in the previous section sets a natural mental process in motion: “Is this the only way suggested for implementing business rules? Or is this just one way? If so, what are other ways in which I can build Business Rules in a robust, extensible and reliable fashion.”

The following section outlines a set of patterns that can be used together to resolve and balance forces arising from the requirements to design and implement business rules. They form a pattern language for Business Rule Construction. Not all patterns are to be used in every project; nor should any project assume that any one pattern will be adequate: the needs to design and implement business rules evolves over time. As the evolution goes on, new forces appear or gain more importance and have thus to be resolved. A pattern resolves and balances some forces while other forces remain; as it were outside of the scope of the pattern based on the tradeoff the designer made to incorporate and implement the pattern.

Thus, with the evolution of requirements, comes the need for new solutions to the problems arising from the forces that change and shape our problem domain. Select the patterns and path within the pattern language that best matches and suits the evolution-path of your project or problem space. Judicious designers will try to foresee future trends of evolution of requirements and, anticipating this trend, will use other patterns and within the patterns use different tradeoffs to cope with them.

Roadmap and Relations between Patterns

Here is a summary and map of how the patterns that comprise the pattern language are related, how they should be used and what their characteristics are. An example of a possible migration path is given.

<<Insert Map Here>>

Scalable Pattern Language for Business Rule Construction and Maintenance

Background

Business rules are like salt; you can taste them in programs but they are dissolved within the code. When changes to business processes, or new requirements arise, two different classes of problems arise for managers and technical staff. Business managers have the rules “in their heads”. If we want to identify a consistent set of rules for the organization, we might invite the managers to a meeting and record the rules each of them proclaims for their domain. The problem, however, is that they are rarely going to be in 100% of agreement; business domains and organizational departments tend to overlap and depend on each other. Thus, if a rule has changed (such as national legislation on telecommunications) and not all managers have been updated, some will continue to base their policies and work on the previous version of the rule.

Corporate Rule Repository

Therefore to have the sum total of business knowledge (exemplified by the totality of the semantics of the rules, individually and as a whole) we would appear to need some corporate rule repository (CRR). This would be used in the following manner: use-cases for CRR.
Pattern Language for Object Business Modeling

Some ways of handling rules have been found to apply to three domain projects: Telecom, Car Industry, and Registration/Order-Entry systems. Some of these reusable techniques are hereby listed for convenience. They form the core of a fledgling pattern language on Business Rules, which my colleagues and I have been working on for the past two years.

1. Rule Object
2. Rules have states
3. Rules are Tracked (Rule Changes are Tracked)
4. Document Rules as Patterns
5. Centralize Rules in Corporate Repository
6. Give everyone read access; policy-makers write access
7. Changing Process Changes Rules; Changing Rules may not change Business Process
8. Clusters Have Rules
9. Rules as first-class Constructs of the Object Model
10. Rules as Production Rules of the Application Domain Grammar
11. Rule Change Impacts Architecture

Use-cases: Business Rule Usage

Some use-cases for dealing with business rules are shared below:

1. Business Manager (BM) creates a new rule. The role is a policy maker in the organization with the authority to mandate and effect corporate-wide changes to business processes, work-flow, policies and rules.
2. Business Manager updates a rule (previous versions are all tracked; history is kept)
3. Business Manager changes the status of a rule from active to inactive (or not applicable anymore). Previous versions are recorded; the rule is not going to be active anymore and will not be used as a constraint henceforth.
4. Technical personnel (TP) access the Rulebase to get the latest update on the rule
5. When a BM creates a new rule that will affect multiple areas of the business (rule ripple effect); TP must be notified of this impact for immediate planning and subsequent actions to be taken.
   5.1. To insulate the ripple effect due to rule changes, we will have conducted OO analysis with classes having rules within them [Fig 1].
   5.2. We need to have an abstraction (let’s call it a cluster) encapsulate business rules that govern a set of collaborating classes. If multiple classes are affected by changes of state of the business rule, they will frequently be a cluster (set of collaborating classes) who will be the one responsible for seeing that the group of classes abides by the constraints. So create a cluster class, have it assume the responsibility of enforcing rules.

Draft:
Even when consensus is compromised, development staff have no clue where to begin coding from: there is need of a corporate rule repository to be accessible by managers and developers alike. Development staff has a hard time maintaining systems. This is partly due to the fact that rules are scattered within the analysis, design and implementation (code).

Business Rules in Legacy Systems

Business rules are hard-coded into the client in the worst systems, are partially on the client, partially on the server and partially on the middle-tier for the best performance-tuned systems.
1. Client (usually field validation)
2. Middle-tier
3. Database (Server-side rules) (Usually cross-table validation)

The concept behind traditional database server-side (triggers, stored-procedures) rules is to concentrate the rules in one modifiable area, and to ensure that no program can by-pass the rules to modify the rows in the database without first going through the server side rules triggers.

Object-oriented architectures and systems are created in the following manner: domain analysis (e.g., culminating in frameworks) or plain old requirements analysis. We have found that the problem lies in not considering rules to be explicitly separate abstraction entities; on a par with attributes and methods.

**Rules as first-class Constructs of the Object Paradigm**

Therefore, we have been teaching OOA/D with the notion that rules are first-class citizens of the object model. As such they belong to classes and are kept in a fourth class compartment along with class name (and possibly stereotype), attributes and methods.

There are some rules that pertain to a given class, and tend to govern its behavior. There are other types of rules, which govern the behavior of a set of collaborating classes. Much as Behavior in data abstraction governs the operations that may be performed on an abstract data type without exposing the attributes and data types themselves (encapsulation of data or information hiding).

We aim to further extend this notion to methods. We would like to encapsulating methods so that they will be called in a typical sequence (perhaps with A Template Method to customize behavior for child classes), or ensure that, for example, whenever an item is taken from inventory, a check is made to determine whether it has reached its reorder point. If so, we would like to place an order. This constitutes the insertion of an action sequence or the fact that rules cause other actions to be performed.
Chapter Four: Refactoring Code to Use Patterns

During the software development life-cycle, there are choices to be made about how we decide to address issues and problems. This is true whether we are doing traditional waterfall model where analysis, design and implementation are sequential, or in a spiral model where each iteration has its own bit of analysis, design and implementation and refactoring.

In whichever process model we choose to delve, we need to make design decisions that will impact the risks associated with our project, its success and its ability to meet functional, non-functional and political demands of a modern software development organization or the strong winds of e-business on the web.

Balancing Current Deadlines with Future Extensibility

There are two general ways in which we may go about implementing and using patterns. Either we already have code that we would like to make more maintainable; in order to extend it with new functionality or attach functionality from another program or legacy system.

The second approach is to use Variation-Oriented analysis and Design from the very outset to factor in the participants of the patterns into your original design. This will create natural plug-points for further extensions in the future.

However, we must beware of overdoing either approach. Let’s assume we have a deadline to meet, and instead of implementing code directly to the problem at hand, without any hesitation (after the analysis and design has been communicated and understood, and use-cases selected and prioritized to be implemented), we spend too much time planning for the future. We try to conceive of and incorporate the most general and reusable implementation of a design that we can possibly create; in order to “have reusable components or code for future use”. Based on such a short-sighted decision, we then run the risk of not finishing the project on time. This risk increases with every generalization we attempt to make. One of the reasons is that this creates new assumptions and dependencies on other developers and portions of the design that were not originally communicated or planned for.

On the other hand, we would not like to program in a “quick and dirty” fashion, just to “get the code out of the door”; where it will probably be flung back at us after quality assurance has had its tests conducted. Or worse, flung in our faces when users in the field are complaining of errors and crashes.

Therefore, a balance needs to be struck between too much unnecessary generalization and too little generalization.

Kent Beck suggest doing the absolutely simplest thing that will get the job done, nothing more. Then, he suggests that the code with then be refactored while doing pair programming, either immediately, or at a later time.\(^{10}\)

In this chapter we will be looking at small examples of programs that we will then refactor to use patterns. This is particularly judicious approach to take in incorporating patterns if you already have the code.

Using variation-oriented design, you would figure this into your original design and thus would not require inexoritant refactoring.

\(^{10}\) [Beck99], Extreme Programming.
Examples of Patterns usage: Before and After

We will be giving examples of how code is written without the application of patterns, and then how it can be refactored to use patterns and demonstrate how that will help future extensibility and ultimate packaging in a framework or a component.

The Model-View-Controller Architecture

Originating with SmallTalk, MVC has become a proverbial “good” in the ethics of programmers. Let us explore an example. The Model is the application data or state. The View (can have multiple, nested Views) is a way of representing the data or state of the Model. The Controller handles input from the user such as a keyboard handler or a mouse handler. If changing the model changes a view, we expect that other views will be notified of that change using a subscribe/notify protocol.

<<insert example here>>

Java AWT and Swing: Differences in Architecture

Java Abstract Windowing Toolkit first used the notion of having to handle events by inheriting from a handler. This proved to be exceedingly inconvenient. A delegation-based event model was designed and implemented as the Swing class library. In Swing, a Delegate is used to encapsulate the View and the controller in one module. The Model is separated out.

If we were to do a list box in AWT it would look this:

```java
Choice myChoice = new Choice();
// Load it up.
myChoice.addItem("Item 1");
myChoice.addItem("Item 2");
```

If we were to do it in Swing, we have three choices: to let the JList create a DefaultListModel, to create a DefaultListModel and assign it to the JList. The latter way looks like this:

```java
JList myList = new JList();
// Create the list model.
DefaultListModel model = new DefaultListModel();
// Load it up.
model.addElement("Item 1");
model.addElement("Item 2");
model.addElement("Item 3");
// Make it so this class listens for changes in the model.
model.addListDataListener( this );
// Tell the list to use the model to store its data.
myList.setModel( model );
```

This represents a separation between the layers of model and view-controller.
The MVC architecture consists of three primary patterns: Observer (to notify Views of changes in Models), Composite (to support multiple hierarchies of views) and Strategy. A View uses a Controller subclass to implement a particular way of responding (a strategy). To implement a different strategy all we have to do is to replace the instance of the controller with another instance of a different kind of controller. “For example, a view can be disabled so that it doesn’t accept any input simply by giving it a controller that ignores input events.”

An object that “does nothing” in response to a message is a special kind of object. It is an example of another pattern called Null Object.

Another way of extending system behavior is through the Extension Object design pattern.

Example 1: Operations on a List Box

Requirements
Our customer has a number of reindeers that he wants to keep track of. We would like to do this by using a list to hold the names of his reindeers.

GUI
Here is what he wants it to look like:

---

12 See [PLOP3], Null Object.
13 See [PloP3], Extension Object by Erich Gamma.
First Implementation

Here is the code for the first implementation:

```java
import com.sun.java.swing.*;
import com.sun.java.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class MyFrame extends JFrame implements ActionListener, ListDataListener
{
    private JList myList;
    private JTextField mField;
    private JButton insert;
    private JButton insertAfter;
    private JButton delete;
    private JButton replace;

    public MyFrame()
    {
        setTitle( "List Manipulation" );

        // Store the content pane in a variable for easier access.
        JPanel contentPane = (JPanel)getCContentPane();

        // Components will all be added to this panel.
        contentPane.setLayout( new FlowLayout( ) );

        // Create the empty list.
        myList = new JList();

        // Restrict to single selection.
        myList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION );

        // Create the list model.
        DefaultListModel model = new DefaultListModel();

        // Load it up.
        model.addElement( "Dasher " );
        model.addElement( "Dancer " );
        model.addElement( "Prancer " );
        model.addElement( "Vixen " );
        model.addElement( "Comet " );
        model.addElement( "Cupid " );
        model.addElement( "Donner " );
        model.addElement( "Blitzen " );
        model.addElement( "Rudolph " );
    }
}
```
// Make it so this class listens for changes in the model.
model.addListDataListener( this );

// Tell the list to use the model to store its data.
myList.setModel( model );

// Create a scroll pane to put the list into in case it gets too big.
JScrollPane scroller = new JScrollPane( myList );

// Make the scroll pane take up the upper part of the content pane.
contentPane.add( scroller );

// Create a panel to hold the various controls.
Box changeBox = Box.createVerticalBox();

// Create a text field with a label.  The contents of
// the text field will be applied when you insert or
// replace nodes.
JLabel newText = new JLabel( "Santa's List of Reindeers" );
mField = new JTextField( "New Reindeer" );

// Make the label and text field line up.
mField.setAlignmentX( 0.0f );
newText.setAlignmentX( mField.getAlignmentX() );

// Add the label and text field to the panel.
changeBox.add( newText );
changeBox.add( mField );

// Leave some space.
changeBox.add( Box.createVerticalGlue() );

// Create four buttons the user can press to insert a
// new item either before the selected item or in the
// list's first position, insert a item after the
// selected item, delete the selected item, or
// replace the selected item.
insert = new JButton( "Insert        " );
s
insertAfter = new JButton( "Insert After" );
s
delete = new JButton( "Delete       " );
s
replace = new JButton( "Replace     " );
s
// Add the controls panel to the lower part of the
// content pane.
contentPane.add( changeBox );

// This is called when one of the buttons is pressed.
public void actionPerformed( ActionEvent e )
{
    String command = e.getActionCommand();
// Get the index of the currently selected item. It
// may be -1 if none is selected, or beyond the end
// of the list if the last item was deleted.
int index = myList.getSelectedIndex();

// Get the list data model.
DefaultListModel model =
        (DefaultListModel)myList.getModel();

// If the selected index is beyond the end of the
// list, treat it like there is no selection.
if ( index >= myList.getModel().getSize() )
{
    index = -1;
}
if ( command.equals( "insert" ) )
{
    // If the insert button was pressed - if no item
    // is selected, create a new item and add it as
    // the list's first item. Otherwise, insert the
    // new item before the selected one. The new item
    // is created using the text from the text field.
    if ( index == -1 )
        index = 0;
    model.add( index, mField.getText() );
}
else if ( index != -1 )
{
    // If an item is currently selected...
    if ( command.equals( "insertAfter" ) )
    {
        // If the insert after button was pressed, add
        // a new item after the selected one.
        model.add( index + 1, mField.getText() );
    }
    else if ( command.equals( "delete" ) )
    {
        // If the delete button was pressed, delete the
        // selected item.
        model.removeElementAt( index );
    }
    else if ( command.equals( "replace" ) )
    {
        // If the replace button was pressed, replace
        // the selected item.
        model.setElementAt( mField.getText(), index );
    }
}
else
{
    // Print out a message saying that the button the
    // user pressed requires that an item be selected,
    // and none was.
    System.out.println( "No item is selected" );
}

// This is called when the replace button was pressed
// and we replace the selected item in the list.
public void contentsChanged( ListDataEvent e )
{
    System.out.println( "Changed" );

    // Call repaint so the change is reflected in the
    // list.
    myList.repaint();
}

public void intervalAdded( ListDataEvent e )
Lab2: Refactor the Implementation

Directions

◊ Refactor the above implementation to use Command, Mediator and Factory Method.
◊ Extend the widgets to implement a Command.
◊ Create a Mediator to handle all the interactions between the widgets.

Lab 3: Add New Functionality
We would like to add some new functionality to our List. Our customer does not want to have duplicates in the list and he also does not want to inadvertently delete an item.

◊ Add a new rule that the Mediator must implement as a Rule Object to check for duplicates
◊ Add another rule object to display a dialog box asking the user if they are sure they want to delete the item that has been selected, when they have clicked on the remove button.

A Portion of a Customer Care System: Method of Payment

Lab 4: A Method of Payment Component

Requirements for the Method of Payment Lab

1. Valid methods of payment (MOP) are The allowed values for "Method Of Payment" are 
   "<none>", "Credit Card", "Electronic Funds Transfer (EFT)" or "Check".

2. Rule: if MOP == Credit Card then the following fields have to be filled out:
   2.1. CreditCardType (== "Visa, MasterCard, Discover, Amex, Diners Club"),
   2.2. ExpirationDate, (must be sometime in the future and not more than 5 years from today’s date)
   2.3. CreditCardNumber (also needs an algorithm to compute the valid number based on 
       the type of card selected. We don’t have to actually implement the algorithm, but 
       merely have a strategy that can implement the algorithm and print a message and 
       return a true or false. Thus the Assessor will use a Strategy to determine 
       whether a creditcard number is valid or not.)

3. Rule: If the method is EFT then the following fields have to be specified:
   3.1. DestinationType (== USA or Other)
   3.2. If DestinationType is USA, then the following field must be filled out:
       3.2.1. American Bank Association Number (ABA Number) (must be a valid one; needs to be 
            Assessed)
   3.3. If DestinationType is not USA then a BankAddress must be provided
       3.3.1. BankAddress : Country, City, Street
   3.4. Bank Name (name of the bank)
   3.5. DebitAccount Number (the account number that is to be debited)
Hints
1. Use a Mediator to coordinate collaborations between GUI elements
2. Use a Command to handle action performed on widgets (GUI Elements)
3. Use a Factory Method to create the widgets
4. Use Rule Objects to model the Rules. The Mediator will use the Rule Object (be its client)
   4.1. Assessors
   4.2. Commands
   4.3. Strategies
Chapter Five: Component-based Development

Background
Developing Product Line Architectures as a capital-intensive endeavor appears to require more maturity of experience across the industry. In an effort to contribute to the pool of knowledge that brings experiences to the critical mass of maturity, we describe some of the successful practices in developing Product Line Architectures in a component-based development scenario. The results of these projects have been the introduction of a few activities to create a viable and successful component-based development approach that is amenable to the development of Product Line Architectures.

As noted in the SEI PLA workshop report ([Northrop99]), different organizations have opted for creating a single product line architecture rather than several, more focused ones. Creating a single product line architecture requires the coming together of several disciplines and requires the introduction of software engineering practices that far outweigh the usual application development suite of skills, practices and processes. These include: applying the concept and practices associated with Reuse Levels during initial Domain Engineering, discovering and evolving components through their “manners”, the mining of domain patterns for lines of business, using meta-domain patterns as an conceptual architectural framework for unifying commonalities across lines of business and finally, defining Domain Languages that serve as Architectural Description and Interaction Languages for the assembly and integration of the components of a Software Product Line.

The remainder of this paper is a distillation of experiences gained in attempting to apply principles of Variation-oriented Analysis and Design (VOA/VOD) to define and produce product line architectures for four clients in telecommunications, insurance and banking within the context of component-based development.

Introduction
Organizations embark on the Product Line Architecture voyage for a number of reasons: cost reduction (skilled human resources, budget, other costs), time-to-market, lower cost of maintenance and mass production based on the crystallization of the notion of reuse. Lastly, another reason we have encountered is, that in order to avoid duplication of software for the purposes of customization for new clients and markets, organizations want to be able to manage the inordinate array of duplicate systems with slight variations in a tractable fashion.

Instead of engaging in a piecemeal single-software product paradigm, organizations are tending towards a capital-intensive software factory type of approach where product lines are created to facilitate assembly of prefabricated components that can be customized for given usage instances and component contexts.

In embarking upon such a quest, two problems or issues must be resolved in order to ensure success or to mitigate risk at the very least.

Background
The Evolution of the notions of Components and Product Line Architectures in an Organizational Infrastructure
The typical scenario is that an organization starts growing and after growing beyond a single initial company, acquires other companies, attempts to integrate their information systems and organizational structures into its own. The usual outcome is a hybrid between the two organizations: the attempt to form a unified whole often fails. Although there are many factors affecting this, two primary ones are the differences in organizational structure and the variances in information systems and infrastructure.
In terms of organization structure, this is a spectrum ranging from informal culture to more formalized business processes, strategic goals and plans, to operational implementation of those strategies and plans.

In terms of the fundamental variance between the information systems, the infrastructure (hardware, software middleware combinations), architecture (combinations of legacy and newer client/server, n-tier and web-based e-business systems), environments (development, testing, staging and production) and processes to produce the software vary considerably. The process is often dependent upon the level of organizational maturity. This maturity level, measured by the SEI Capability Maturity Model, tends to vary, not only from company to company, but most often, from development group to development group.

Difficult as it is the next merger or acquisition makes things even more complex. Often, there are two or more parallel efforts and products that have overlapping target domains: they essentially cover the same vertical domain. This could be (varies by industry) insurance claims systems, web-banking or telecommunications customer care, rating or billing systems.

Thus, the attempt is made to somehow merge these duplicate efforts and products into the amalgamated fabric of the assimilated new organization.

These efforts often have impacts on how the organization does business with its clients. Supporting a new client who uses one of the “old” information systems leads to a huge project involving many scarce resources in an attempt to breed a new system with slightly different features, functionality and business rules.

Having not originated from a common set of core assets that share commonality and have – at least theoretical – plug-points for extension, the cost and time-to-market to accomplish this becomes a high risk factor with every new potential opportunity for growth and expansion of market. Indeed, in most cases, e-business is a recurring theme: extending the legacy applications to the World Wide Web requires a transformation of business processes and information systems to accomplish this goal.

Software Product Line Architectures for e-Business
One of the most challenging and widespread themes during the past five years has been that of extending an organization’s systems already in production, especially legacy applications, to the World Wide Web. This has been seen to require a transformation of the way the organization does business (business processes and activities) to accommodate new functionality and gain new richness and reach. The value proposition of reach: the possibility of extending the business processes of an organization to new markets and indeed to individual consumers (e.g., based on analysis of customer profile and demographic information analysis in data warehouses) has been an over-whelming motivating factor.

The other, perhaps equally motivating factor has been gaining competitive advantage over competitors by extending the reach to the Web in a timely and powerful fashion; carving out a market territory in virgin ground.

**The Gap and the Common Meta-Model Problem**

**The Gap**
The process of component-based development geared towards a Product Line commences with the investigation of business processes. Subsequently, domain engineering is conducted (abstract to concrete) in order to discover commonality and variability among the various “line of business”. In parallel with the latter effort, investigation of candidate Product Line Technical Target Architecture (more concrete to more abstract) can take place. Upon the completion of the Domain Engineering activity, a given target component architecture is selected. The selection criteria includes the requirements and results of the domain engineering effort which can be most appropriately satisfied by the Target Component Architecture. A corporate transition plan is produced in conjunction with various stakeholders within the
MUM Software Engineering Course

organization. This organizational transition plan helps educate the divisions of the organization in the new paradigm and instills a new cultural infrastructure that is conductive to product line development and maintenance.

During this process, we have encountered several recurring technical obstacles and common organizational hurdles. From a technical point of view, there is a large conceptual “chasm” between the domain component architecture and the technical target architecture. This gap has been a source of project risk.

In order to fill this gap, a “middle-out” approach is necessary to complement the domain engineering’s top-down approach and the target component architecture’s compositional inductive or bottom-up perspective.

The filling of this gap is in part facilitated by the constraints that are defined as architectural styles [Garlan94]. To help close the gap even further, we have found that the remaining key factor that helps cross the chasm between domain analysis and target architecture is the use of a meta-domain pattern to serve as a common ground of unification. The use of meta-domain and domain patterns, along with their critical role in reducing risk, increases traceability and helps create a blueprint context in which later iterations of the Product Line Architecture efforts within the organization can be staged. Meta-domain patterns such as Service Provider [Arsanjani99] help serve as conceptual scaffolding on which individual domain patterns for each line of business can be more easily structured and placed.

The Role of Architecture and Domain Patterns

Reference architectures and architectural patterns have helped make this transition a smoother and more cost effective one. However, the lack of a reuse program that capitalizes an organization’s already-existing assets and investments, and channels new development efforts in a common and compatible direction is seen.

Three levels of Architecture

- Business, Application and Target Technology Architectures
- Risks associated with having them not in line.
- Risks associated with not having an end-to-end system to prototype the components and start the product line.

The Common Meta-Model Problem

During the process of conducting domain engineering for component-based product line architectures, we have experienced several cases of the “common meta-model problem”. Here is a summary.

In performing domain engineering, there are two general approaches that we may take from the point of view of capturing commonality and variability. Firstly, we may construct a common meta-model for all lines of business and then attempt to perform domain engineering on each one with reference to the meta-model.

The second approach is to take more or less, one line of business (usually the one that is specified in greater detail, has better documentation, including further practical and theoretical considerations) and use that as a baseline. This baseline is then used to produce common themes and a frame of reference for variations across lines of business.

Any of the two methods described above can ultimately lead to the creation of a common meta-model. This meta-model outlines the criteria by which future extensions to the model itself can be made. Examples of this problem are the creation of the UML meta-model subsequent to its creation. This is an example of the second alternative. An example of the first alternative is the creation of the OPEN modeling language where a meta-model was first created through explicit generalization [Bulthuis95].

Although both roads lead in the same general direction, the adoption of one has its shortcomings. Whichever approach we take, we may risk a large degree of traceability to actual systems and business
needs if we first do not mine out the domain patterns for each line of business. Each line of business (LOB) has a set of commonly accepted practices and design decisions that work for that particular LOB and has proven successful for them. Whereas the principles, tradeoffs and design decisions that define the [often implicit] architectural style associated with a LOB may be unique to that LOB, they none the less provide a great deal of value in bringing that business to meet its objectives. Thus, we cannot afford to lose this set of design decisions that may, at first sight, be unique to a particular LOB; or else we run the risk of not satisfying the practical critical success factors that are salient to a given line of business.

Mining out these design decisions, tradeoffs, architectures and even non-functional requirements in terms of domain patterns decreases this risk by making it an explicit artifact of the Product Line Architecture development process. The results of this pattern mining and reference architecture creation will itself be treated as a “core asset”.

**Solutions**

In order to balance the forces involved in the above two general categories of issues and problems, we have introduced a few convenient constructs, notions, activities and a process that we shall outline below. In brief, these consist of:

1. Incorporating and implementing the notion of Levels of Reuse,
2. The **manners** of components and reusable assets for component discovery and use-case analysis,
3. domain patterns as starting points for domain analysis and
4. The use of meta-domain patterns (e.g., Service provider as in [Arsanjani99]) as a conceptual infrastructure for unifying domain analysis results across lines of business.
5. Creating the Component Context, in addition to focusing on Component Definition, Connectors, Ports and Interfaces.
6. Lastly, a life-cycle for component-based development of Product Line Architectures is outlined.

**Reuse Levels and Components**

**Background and Taxonomy**

It is a prevalent practice to conduct analysis and design leading to the creation of a corporate domain model or enterprise model with the use of objects. It is shortsighted to assume that this is the only method of doing so. On another level, if we are using the object paradigm to conduct such analyses, we believe that the process of creating PLA’s will be greatly enhanced by evolving the current paradigm of using fine-grained objects to one of using what we call Reuse Levels and Components.

Objects are too fine grained to be of large-scale reuse. We need larger grained entities for reuse. In order to do so, it is convenient to define Levels of Reuse and of components being or belonging to one of these Reuse Levels:
12. Technology transfer knowledge,
13. Method implementation/customization knowledge (mentoring meta-knowledge).
14. Process Improvement Knowledge

### Asset Reuse Levels

The first six levels of reuse have code manifestations. Components have meta-data and descriptors in addition to standard code. Levels seven (patterns and beyond) through ten are of a technical nature. Levels 11-14 are of an organizational nature.

### An Assembly-based Paradigm

In an attempt to create and render components that are amenable to integration and assembly using a variety of component interconnection protocols (CIP’s), such as plug and play adaptive components (e.g., [Mirenzi98] ) we need to have a common metric of a component’s size and reusability. For this purpose, it is convenient to use a taxonomy of Reuse Levels. Thus, components of the same reuse level are more amenable to integration and assembly than those of different reuse levels.

### Reuse Levels

Over the past five years, we have found the notion of a reuse level to be a convenient conceptual taxonomy that not only helps to categorize but to serve as a starting point for component discovery and identification. Thus, in the requirements analysis phase, we have been advocating Analysis-by-Cluster (and later in the life cycle, Design-by-Cluster) rather than analyzing the domain using individual objects only. Traditional object-oriented analysis starts object discovery with identifying nouns, or eliciting requirements using use-cases or CRC cards and then identifying nouns, etc. This approach has been seen to be too fine-grained from a reuse engineering\(^{14}\) perspective.

Here is a brief example. Often, the valuable information is not about an OrderLineItem or an Order, but the fact that the Order is composed of a header, a set of OrderLineItems and a Footer. Each Order Line Item has a Price and a Product associated with it. Thus, the Cluster of objects \{Order, Order Line Item, Header, Footer, Price and Product\} form a unit of collaboration that runs through the entire domain of order entry (for example).

A component (the definition of which is a highly controversial subject these days) can range from fine-grained to medium-grained to large-grained. Within this spectrum of variability, the common theme is that the component is an assembly-conscious software asset. This asset may be placed in a repository facilitating search and retrieval, usage and bugs, version history, etc. The asset may be associated with a set of non-software core assets. This may include a number of meta-level information describing the core component/asset, such as: use-cases and test cases, usage instances and history, certification process, patterns and architecture it uses, extension points, plug-points, customizations, non-functional requirements, etc.

There are various ways in which larger object graphs are composed: through inheritance, through composition, association or a combination of the above. Each of these produces an object graph that may be treated as an asset in its own right. Each object graph thus constructed is said to be at a Reuse Level of base class, aggregation hierarchy, inheritance hierarchy or cluster. These constitute the first four Levels of Reuse.

Component Analysis starts with the selection of the Level of Reuse at which we will be “chunking” the analysis; defining the domain or requirements analysis’s granularity in terms of its end products.

We will later see the application of Reuse Levels and Components in the Component-Based Analysis and Design process.

---

\(^{14}\)The Reuse Engineering life-cycle is outlined in diagram 1.
Thus, instead of “underlying nouns” for object discovery we identify component manners, first. Through the knowledge acquisition associated with discovering the business rules, usage rules, validations, valid object or domain interaction sequences and collaborations between participants that are at the selected level of reuse, we begin to discover more detail regarding the state, behavior and further manners of the component. Note that components at the cluster level may be business process level components.

How Much Should We Build?
In attempting to build a Product Line Architecture, there are many design decisions to be made and documented, so the next iteration or group will have the benefit of understanding the constraints under which we operated to produce deliverables. Among these constraints, one of the more important ones pertains to whether we should build a common denominator that every line of business can use in all projects across the organization or have various groups product their own core assets and reusable components, hoping to have some cross-reuse. Note that the latter approach stance runs a higher risk of failure from a reuse perspective, since the inter-organizational politics disallows the reuse of assets except under explicitly motivated circumstances [Griss95].

Object and Component Discovery
Instead of starting medium- to large-grained component discovery and development with the discovery of fine-grained objects, we propose an alternative scheme. Let us first identify the reuse level that we are trying to target; this is the granularity of components we will be building. Granularity may change at some point – and this is fine. But at each stage, an explicit level of granularity – in terms of a component at a given Reuse Level will be within context. Upon selection of the reuse level of choice, its components along with their implied level of detail will be used during domain and requirements analysis.

For example, we propose that component-based development efforts should not be commenced with the identification of object-level components that are fine-grained. Instead, a higher Reuse Level should be chosen, such as the Cluster. The Cluster is a replaceable, executable, Composite Mediating Façade. It is a medium to large-grained component. Replaceable to be interchangeable with components of like functionality providing and adhering to the same set of interfaces and component Manners. Manners are the interaction and collaboration rules that a component obeys relative to the Component Context into which it is placed, in addition to the meta-data that make the component reflective and self-describing – and thus dynamically (self-) configurable. A Component Context is the environment in which the component will be used. This environment is described through the use of architectural styles.

Thus, the way in which we conduct analysis for Product Line Architectures and for component-based systems is an enhancement to the way we build application systems using the standard object paradigm (e.g., as described by UML 1.3).

A Three Pronged Approach to Developing Product Line Architectures
The current systems and policies did not grow overnight and cannot be changed overnight without obvious adverse ripple effects within the organization. Therefore, an evolutionary approach is needed to dovetail new development requirements, the merging of redundant vertical systems, creating a common set of core components between vertical markets with legacy systems and business processes. For this, a three – pronged approach has been seen to be necessary.

Domain analysis is needed to discover and document commonality and variability between vertical lines of business within the organization. This helps produce a “road-map” for defining a target Component-based

---

15 These refer to the standard design patterns in [GHJV94].
16 See the discussion on Component Manners, below.
Product Line Architecture that will support the creation of Product Lines. This is the top-down aspect of the process.

A parallel effort, the second prong, is to perform knowledge mining of existing assets for an inventory of existing potentially reusable assets. This may result in gluing together large grained sub-systems with messaging protocols (message oriented middleware) with newer client-server or n-tier architectures currently under development.

A corollary of the second prong is to transform rather than re-write legacy systems subsequent to the knowledge mining process. In this way, potentially reusable components are identified, extracted and harvested. They will soon join the pool of core assets we have started to build.

The third prong is to use a bottom-up approach to assemble and aggregate smaller grained objects or units of functionality into medium- and large-grained components. This “sub-system refactoring” (as distinct from code refactoring) elevates these smaller grained and often intractable objects that often are used as “grape-vines” (to use one object you have to use the entire web of objects across reuse levels).

**Build, Buy or Transform?**

So when the time comes to make the decision as to whether to build or buy, managers should be made aware of at least two alternatives: outsourcing (in an offshore development context) and transformation or mining and harvesting of legacy assets using automated tools. Among these four options transformation provides one of the most attractive and powerful alternatives. Rather than rewriting systems that have already been heavily invested in, use a knowledge mining strategy and toolset to identify potentially reusable assets that are in accord with the newer architecture. Mine, extract and transform the legacy component in order to accommodate it into the new architecture.

**The Component-based Development Life-cycle**

On one axis, there is great controversy as to whether the CBD life-cycle should be predicated or based on object technology. Components can be at any reuse level. As such, they may be a module that communicates with other components through some form of middleware: CORBA, DCOM, EJB, MOM, etc.

On another axis, components may be built, bought or transformed from legacy assets. The notion of a PLA adds the concept of a common pool of key/core reusable assets that have been identified by domain engineering to be of utility to the entire breadth and depth of the way the organization conducts its business processes.
Component-based development for Product Line Architectures: Life-cycle

The following list of stages outlines the experiences gained during four component-based development projects that were geared towards product line architectures. These steps may be used as guidelines of when and how to incorporate the five major concepts that were presented in this paper.

Briefly:

1. Conduct Business Process Identification and Analysis (Re-engineering if applicable)
2. Conduct Domain Engineering
3. [Identify which meta-Domain Pattern is applicable to these sets of domains/lines of business, if any]
   3.1. Use the meta-domain pattern as a template to organize and categorize ideas, requirements, participants, commonality and variability
4. Identify/Mine Domain Patterns to provide a conceptual framework on which the place the findings,
5. Identify Participants and Roles mapping to participants from system instances
6. Identify Domain Collaborations and Dependencies
7. Perform Variation-oriented Analysis and Design
8. Plan for a number of iterations for mapping to target component architecture
   8.1. Resolve and mitigate Risks; use domain patterns to fill the gap between domain analysis and target component architecture
   8.2. Start with a first thin-slice end-to-end component assembly map:
       8.2.1. List of dependent components that can be built in a “short period of time” with a target component architecture that is understood within the organization.
       8.2.2. Potentially plan to migrate to other (better?) target component architectures
           ❖ Watch for industry stability
           ❖ Use traditional technologies if you have to; due to:
               ❖ Industry instability
               ❖ Development team lack of expertise in the area; bring one a few experts
9. Define the Component Context
10. Design and Build small frameworks for core assets
11. Encapsulate frameworks within components
12. Assemble components
   12.1. Component Integration with any sort of middleware
       ❖ Message-oriented middleware
       ❖ CORBA
       ❖ DCOM
       ❖ Enterprise Java Beans
   12.2. Assemble using scripting languages; architectural “glue” languages based on the definition of the Domain Language.
13. [ongoing throughout] Configuration Management through creation of a Corporate Reuseware Repository by Product Line
14. Transition to Field
15. Maintenance of assets
   15.1. Mine out more domain and meta-domain patterns to serve as rapid starting points for next iteration
16. Process Improvement for the next iteration

How to Improve the Process [Daftari91]

The SEI Framework for Product Line Practice identifies the key practices in developing Product Lines. The above life-cycle presents of series of steps and stages that attempt to cover the product line architecture lifecycle from inception through completion and fielding. As with all over processes, there needs to be a process improvement strategy for Product Line Architecture Development Life-cycles. This can be found in [Daftari91].
There are two complementary aspects to developing Product Line Architectures using a Component-based development paradigm: technical and managerial. The following diagram attempts to capture the managerial, project management aspect.

Diagram 1: Reuseware Development Life-cycle – Project Management View

- **Conduct Feasibility Study for Product Line to determine**
  - Conduct Domain Engineering, Requirements Analysis, and Architectural Design and Reuse Engineering

- **Determine granularities of decomposed hierarchy of abstractions corresponding to development-oriented use-case**
  - *(i.e., iterate)* For each level of granularity of Reuse Level and each use-case/feature

- **Partition into 80/20 subsets based on risk analysis**
  - Remaining 20% of planned reuseware
  - Develop (Produce) and QA Reuseware for the first 80%
  - Remaining 20% of planned functionality
  - Develop customized/app-specific non-reuse software

- **Record all knowledge gained in project reuseware repository (PRR), preferably in object paradigm terminology**

- **Apply abstraction, decomposition, hierarchy, and concurrency**

- **80% of functionality usually takes 20% of the effort; 20% of functionality takes 80% of effort**

- **Develop and QA Application for the first 80%**
- **Consume Reuseware, Relative to supplied test cases**
Summary
So, to summarize the propositions that have been laid out in this paper so far, we have added two essential ingredients to the recipe for the creation and evolution of Product Lines and product Line Architectures.

Asset Reuse Levels
The first is the introduction of the notion of a Reuse Level, to be applied during the initial stages of domain and requirements engineering. A Reuse Level is chosen based on a Reuse Engineering Assessment (REA). The result of this assessment tells us which level of reuse is more suitable for being incorporated into the domain or requirements analysis process. For example, to build components and conduct component-based development, by choosing the reuse level of a fine-grained object is likely to increase the chances of failure by producing an explosion of objects.

The exponential proliferation of interconnections and coupling (associations, inheritance, dependency) between these candidate objects of fine-grain leads to solving the wrong problem: the software engineer is engrossed in the process of overcoming the entropy that has avalanched upon the project by attempting to organize and classify sets of collaborating types. But this is a battle that is best not fought. It detracts from the very goal of the effort to build a set of common, reusable components; an effort which has at its heart, the objective to identify and build highly cohesive units of reuse that have explicit plug-points, extension points and hot-spots.

Thus, the selection of a Cluster reuse level would be the appropriate choice in this case. The Cluster embodies and reifies the collaborative nature of the quest for pluggable components. As a Composite, Mediating Façade, encapsulating the manners of the component, it provides and conforms to a set of interfaces that can be dynamically (reactively) re-configured through negotiation (agents). Alternatively, in the more non-reactive system role of a Negotiatory Business System -- for example a business-to-business price quote transaction over an extranet -- to interface with meta-data belonging to the Business Party and utilizing its internal manners to interact and negotiate business agreements (e.g., provide price quotes) through the grammar that its manners contains.

Components have Manners: Manners Drive Collaboration Discovery
The second element of importance is the utilization of a component’s “manners” to discover its other characteristics that aim to satisfy a set of features. These characteristics include constituent components, more finer-grained levels of reuse such as inheritance hierarchies, aggregation hierarchies and, ultimately, individual objects.

Therefore, instead of performing component discovery through “underlining the nouns” as is the case for object discovery, business processes engineering is conducted to discover a component’s manners. This includes discovering the business rules governing the internal state of the cluster (component), its externally visible protocols, and finally, its meta-data expressed as a Domain Language in the form of a context-free grammar that describes its generatively infinite number of valid interaction sequences. Note that the externally visible protocols are in the form of rules governing its collaborations with other potential implementers of a Type (other components).

Domain Patterns: Capturing the Language of a Line of Business
Thirdly, the focal point of domain analysis, in addition to more standard practices of identifying commonality and variability, features, etc., will be the mining out of domain specific patterns. These domain patterns reflect the nature of the line of business (domain) under investigation. It includes the specification of a higher level of abstraction that analysis patterns [Fowler96]. Domain Patterns [Arsanjani99] include generalizations of functional and non-functional requirements within the line of business; within all its variations. They also include business processes, activities as discovered through business process re-engineering. In addition, standard reference architectures that support the line of business are identified and included within the domain pattern.
Therefore, domain patterns guarantee that the peculiarities of a line of business is not overshadowed in the attempt to unify all lines of business with a single (often, least) common denominator. It ensures the integrity of the domain engineering results by providing traceability back to the generalizations of the business processes, activities, features and architectures that have successfully passed the test of time in a given line of business.

**Meta-Domain Patterns: Conceptual Framework for Generic Architectures**

Fourth, the domain patterns may reflect an underlying meta-domain pattern such as Service Provider [Arsanjani99], which can be used to unify various lines of business and apparently unrelated domains. Meta-domain patterns provide the conceptual groundwork and foundation upon which the effort to produce a common set of reusable assets can be built for a Product Line. Meta-domain patterns add further value by lessening the conceptual gap between the results of domain analysis and the target component architecture that will be used as a standard platform for implementing the Software Product Line within the organization or sub-set thereof.

**Architectural and Component Contexts**

Much has been written about architecture and components. One of the few topics that has not been adequately covered – if at all – is that of a component or architectural context. The very notion of a component presupposes an environment into which it can be plugged into and subsequently connected to other components that were designed to live in the same component context or environment. A component has an interface, plug-points (conceptual), connectors and ports (physical). But it also needs to have an explicit Component Context.

This is especially true when we are dealing with the creation and evolution of Software product Line Architectures. As is well known to PLA practitioners, the process of creating and maintaining a PL is more involved and complex than a single product.
References


