A Pattern-based Approach for Building Reusable, Technology-neutral Component Integration Architectures

Ali Arsanjani
IBM,
National E-business Application Development,
Center of Competency,
Arsanjani@us.ibm.com

Abstract

In this paper we present Enterprise Component, one of the more significant patterns of a pattern language-based approach for building technology-neutral, reusable component integration architectures. It addresses the issues and problems of how software architects should model and construct component-based architectures in a technology-neutral specification context for medium- to large-grained enterprise-level components in the face of shifting and often conflicting decisions points, based on the notions of subsystem analysis and variation-oriented design. Components of composite granularity can be designed and implemented; workflow and business rules are appropriately partitioned in this component framework.

1 Introduction

The prospect of building software applications out of pre-fabricated parts represents a fundamental departure from the development-centric view of current software engineering practices. This is an assembly-based and composition-centric view of software engineering (Component-based Software Engineering).

Many software project organizations are struggling with a combination of new development efforts including n-tier architectures and Enterprise Application web-integration. This is the e-business enablement of legacy applications that must also integrate and work hand and glove, with newer, n-tier development efforts. The solution to the integration of legacy and web-based technologies rests on a spectrum starting with “screen scraping” approaches, using (e.g.) Java to connect directly to host-based systems (using tools and APIs such as Enterprise Connector Technology from IBM) to wrapping calls to backend systems and front these calls with a services API that essentially is an adaptor to allow back-end functionality to be accessed from a middle-tier. The end of the spectrum is to use a transformational approach to partition and componentize current legacy through actually reading in their code and partitioning functionality and extracting business rules using automated tools.

Yet even this latter approach requires a methodology for defining the “chunks” of functionality, what portions to keep and which to phase out, how to integrate these transformed parts of a monolithic system with current web-based middle-tier business logic.

We have mined out best-practices and experiences in this area into a pattern language for component-based development and integration. We will cover one of the major patterns in this pattern language, called Enterprise Component.
Issues in Component-based Development

The current industry focus is primarily on the implementation and deployment aspect of components. The Enterprise Java Beans server-side component model[26], for example defines roles and standards for the implementation, delivery, deployment, assembly of server side components as EJBs. Other component models (CORBA Component Model and Microsoft COM+) provided functionally similar perspectives.

One shortcoming of these standards, or perhaps the area they were not meant to address is in the modelling and design of the component; other views of the 4+1 View of Architecture [Krutchen91] other than the Deployment View.

Typically, object-oriented methods (e.g., Unified Process of Software Development) do not provide full life-cycle support for crafting component-ware. Although there are icons in the UML that represent components, the semantics of how to model, design and architect these components have been largely absent from the Unified Process and similar methodologies. Although other methods such as Catalysis [24], Syntropy[25], etc., address the designing of components via interface definition and bringing precision by introducing OCL [4] into the mix, there is still a gaping hole where business modelling should be conducted. The Eriksson-Penker extensions to the UML and their business modelling techniques [5] are more of what is needed to conduct business modelling and map the business architecture onto a software architecture that is intrinsically amenable to component-based development and integration.

We have seen the emergence of the CORBA Component Model, Microsoft COM+ and Enterprise Java Beans (EJB) server-side component models. These tend to be run-time and technology focused.

First, is the problem of providing full-life-cycle support for component-based development. This starts with modelling a component and refine this model in a consistent manner down to implementation; mapping it onto one of the above technologies. One of the major issues in implementing a component-based approach is how to conduct analysis and design using object methods to arrive at larger-grained components.

Second is the issue of providing software architects with a uniform set of best-practices and patterns that give them an operational definition of how to design technology-neutral components that can be mapped onto implementation standards.

1.1 The Five Domains of Component-based Development and Integration

Addressing component-based development and integration in an enterprise context requires support from five domains of concern:

- Organizational Domain
- Methodology/Process Domain
- Architecture Domain
- Reusable Technology Implementation Domain
- Infrastructure Domain

The description of these domains is out of scope of this paper. We will, however, concentrate on demonstrating some aspects of the methodology domain.
2 Component-based Development and Integration Pattern Language: The Architecture Domain

Through the course of retrospective analysis of multiple client engagements across several industry domains it was clear that clients needed an operational definition of how to build a component; not only the definition of a component.

There are 18 patterns in this pattern language. We will be talking about one of the most important ones, the Enterprise Component, which shows how to handle the creation of robust enterprise-scale components and provide capabilities to add rules and types to the black-box component without making intrusive changes in accordance with Meyer’s Open-Closed Principle [15].

We define a component as being specified through the application of four basic design patterns: Façade, Composite, Mediator and Rule Object. In order to handle increasing complexity the following patterns can be combined to resolve more forces should they arise in the context of designing a medium- to large-grained enterprise component.

The application of multiple design patterns in a repeated sequence of application to solve a common set of problems is called a compound pattern. Enterprise Component is the first (compound) pattern of a pattern language for Component-based Development and Integration (CBDI).

This operational definition of an Enterprise Component has been taught to architects and team leads in a very short duration to provide guidance on how to craft uniform and predictable component structures that can be placed within the context of a component architecture.

Let us take a small example and evolve it into an Enterprise Component. Let’s start with a simple banking scenario where we have a Business Party (who can be an individual customer or an organization) open a set of Accounts and later perform Transactions on the Account(s) to record the activities of the Business; which affect the Entries on the Account. After conducting Business Modelling we often find this set of business objects often working together:

![Business Object Model for Party-Account-Entry](image)

Of course, the design level class diagram would be much more detailed than the business object model in Figure 1. Refining the business object model into a design object model in a proper object-oriented analysis and design method, we then see that this type of “business pattern” or more precisely, meta-domain pattern [1] has applications in Banking, Inventory, Payroll and indeed in an immense variety of business domains.

So we decide to first use this in a Banking application. Pretty soon, we will find ourselves writing new, additional code to fit it into a Payroll application context. This is where we would build a framework around this cluster of classes.

2.1 The Enterprise Component Pattern

This pattern is one of the essential patterns in the CBDI Pattern Language. Creation of large frameworks are error-prone with low reusability. Therefore, build smaller, more light-weight frameworks to capitalize on reuse for an application domain or family of applications. After creating a lightweight framework, we would apply this pattern by encapsulating the framework using the Enterprise Component pattern, which shows how to componentize subsystems or clusters. Figure 2 shows the result of applying the Enterprise Component pattern by
encapsulating the framework with a Façade, defining Mediator(s) for the Façade with pluggable workflow and exception handling rules.

![Figure 2: The Enterprise Component Compound Pattern](image)

Although this is a technically correct implementation, it does not conform to the Open-Closed principle, which disallows intrusive changes, for example, to be made when adding a new type of Account. Therefore, to preserve the black-box nature of a component, we introduce a Type Object for every Interface Type, thus

![Figure 3: Applying the Enterprise Component compound pattern; encapsulating the Framework to get a Component](image)
Figure 3 shows the Party-Account-Transaction Analysis Pattern that is implemented as a framework. First the business objects (classes) are determined (Figure 1). Then a design object model is created which refines the first model (Figure 1). The forces driving the design object model in figure 3 are the need to program to interfaces rather than implementations. This explains the introduction of interfaces with method signatures using only interface type arguments. Next, we would like to reuse some typical collaborations; without having to rewrite them every time we wanted to build a slightly similar application. For example, consider the following collaboration diagram that exemplifies one of many potential reified collaborations that would go into an abstract class that realizes each interface.

In this example collaboration, a Business Partner sends a debit message to an Account. The Account checks its rules (preconditions), creates a debit transaction, validates the transaction, submits the Transaction, in which case the Transaction actually debits the Accounts it must debit. Another scenarios is when business objects have rules that they must check. A business rule may be triggered in a variety of ways, outlined in the Rule Object Pattern Language [11]. There are several situations that rigger the checking of a Rule: upon occurrence of an event, or the presence of a condition, before the invocation of a method (pre-condition), after the termination of a method (post-condition), during the execution of a method (invariant).

Without introducing too much variety and choices and thus avoiding needless complexity in the resulting implementation, typically, business objects and components should know their own rules of behaviour and the rules governing the behaviour of subordinate classes. Class B is
subordinate to class A when B is aggregated by A, or is on the many-side cardinality of an association with A:

Thus, A contains a reference or collection of type B, or if there is an abstract method that B implements, then A can have business rules that govern the behaviour of B. For example, a Business Partner controls its own Accounts. The AccountType determines whether there is a minimum balance constraint on its Account Instances. For example the sum of balances on all Accounts held by a Business Partner (Customer, in this case) cannot be less than $500 or a penalty percentage will be applied to the account.

In a more complex instance, it is a Rule that hangs off of the relation between a BusinessPartnerType and an AccountType: when there is a government agency business partner who has these types of Accounts, then these are the rules governing the behaviour of accounts.

2.2 “Pay as You Go”: Adding Solution Elements On Demand

The approach taken in the CBDI pattern language is to start with a very simple, agile and lightweight design. When new issues and constraints bring new forces into the picture, new solutions are required to address them. Therefore, we choose new patterns in the pattern language to add as solution elements based on the demand created by introduction the of new forces into the problem domain. Thus, the solutions (corresponding to patterns) cover a spectrum beginning with three simple patterns: Façade, Mediator and Rule Object. As the need arises for the Mediator to control and direct complex components and fine-grained business objects in a uniform fashion, the Composite pattern is added into the mix. The application of this pattern signifies that Mediators can view other Enterprise Components (medium- to large-grained reuse elements) and fine-grained business objects in a uniform fashion.

If the Mediator’s workflow tends to change or if there is a need for a Configurable Workflow [17], then we could externalise the workflow of the Mediator into a Rule Object, which would control the sequence of tasks and message-sends to objects that it has decoupled; yet is coordinating to fulfill a use-case.

In some cases we may need to maintain the State of the Workflow and this may impact the way objects behave. I.e., the behaviour of objects become dependent on the State of the Workflow. In that case, a State design pattern can be added to the Mediator. This would adjust the behaviour of the Mediator and messages it sends based on the state of the workflow.

The Mediator decouples business objects so they can be reused in other contexts. The business objects are thus Subordinate Objects to the Mediator and as such, the Mediator will be responsible for enforcing rules, constraints and policies across its business objects colleagues. If the number of these rules increases, then the firing rules in (random) sequence may not be acceptable; a Strategy of what the next rule to check may be warranted. In this case, the Mediator will have a Strategy deciding how to apply its Rules across the business objects (perhaps based on the State of the Workflow or Exception that the Mediator is Handling).

The more complexity and configurability or adaptability is required, the more forces are introduced and thus additional patterns might need to be employed to balance the conflicting constraints and forces. For example, if you want to have a configurable profile, you need a factory owned by the mediator to read in the current configurable profile on system load and instantiate the necessary Commands or Rule Objects and their Conditions and Actions.
(frequently implemented as Commands). Or the configurable Profile is used to create an Externalized Workflow which are a set of rules governing the manners of a subsystem or class.

Here is a representation of an Enterprise Component in a block diagram:

Figure 8: An Enterprise Component and Extension Points

The diagram depicts the meta-model of the internal elements of an Enterprise Component. A salient feature of this technology-neutral component architecture for building medium- to large-grained components is that the internal behaviour, business rules and business logic of the component can be altered non-intrusively by adding new Rule Objects, Condition and Actions to any business object or component on the one hand, and the ability to pluggably and adaptively alter the workflow of the component’s mediator.

3 Methodology Domain: Extending Current Methods for Component-based Development

In this section we will explore some of the key aspects of the method extensions that support design-time component-based development and integration practices throughout the software engineering life-cycle.

3.1 Subsystem Analysis

After performing Business Modelling, we partition the key business domains and their processes into a set of loosely coupled and ideally disjoint set of collaborating subsystems.

Subsystem analysis starts with the domain as expressed by a business domain model. This includes a set of business rules that show how the forces within the domain interact. It identifies the key subsystems based on the rules of the business domain. A use-case grammar is written to describe the general functionality of the domain in terms of the scope of the subsystem. The use-cases grammar defines the extensions for component-based development. They show how the use-case behaves with respect to the partitioning of the domain into functional subsystems.

It is key to identify interfaces to their subsystems. These interfaces to the subsystems will later be defined during Architectural Design, described below.

Subsystem Abstractions

An experienced architect uses patterns that are distillations of past experience that are known to work and provide effective solutions to clients across multiple projects.
Thus, the definition of architectural mechanisms bring tried and true best practices into the project. The introduction of architectural mechanisms occurs here at this point in the UP workflow.

Consider what the world would look like to an experienced software architect; instead of a large number of tiny (fine-grained objects), the world of the problem domain would be partitioned into a set of subsystems that can be typically found in all business systems. In each “incarnation” (instantiation) each subsystem may be implemented in a slightly different way that depends on the target problem domain. However, the set of types (interfaces) that define the cluster comprising the subsystem remain unchanged.

Even though we have not shown the complete Subsystem Analysis (using the alternative flow in the Unified Process workflow) for the E-Bazaar example, we can look at what some of the results of Subsystem Analysis would look like, in order to better describe the subsystem context diagrams and their rationale.

For example, consider the following subsystems that may have been identified during Subsystem Analysis after an initial Architectural Analysis:

<table>
<thead>
<tr>
<th>Customer Relationship Management = {Account Management, Contact Management (Addresses), Security, Customer Profile and Preferences}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order Management = {Order Entry (Shopping Cart Management), Billing, Fulfillment}</td>
</tr>
<tr>
<td>Product Management = {Products, Catalogs, Pricing}</td>
</tr>
<tr>
<td>Inventory Management = {Fulfillment (Picking and Shipping), Vendor Management}</td>
</tr>
<tr>
<td>Financial Management = {Billing, Accounting and Bookkeeping, Accounts Payable, Accounts Receivable}</td>
</tr>
</tbody>
</table>

Note that Shopping Cart Management can be thought of as part Order Management subsystem. However, it is architecturally significant so that in Subsystem Analysis we consider it to be a separate subsystem abstraction. The use-case grammar showing their relationship with regard to the larger-grained enterprise-level subsystem we are calling Online Purchase Subsystem can be written as a combination of other subsystems that together form a domain language defined by the following domain grammar:

| Online Purchase Subsystem = {Customer Management, Order Management ([Shopping Cart Management (Order-Entry)), Order-Processing, Fulfillment, Billing], Product management} |

The above domain grammar tells us that in order to build reusable components for a larger Online Purchase Component we need to build at least four other components relating to handling of Customers, Products, Shopping Carts and Orders. Orders pertain to the back-office activities that occur once an online purchase has been made. The Shopping Cart metaphor is used to set up the order. Once the order has been confirmed, an Order is created that is sent to Accounting (to generate an invoice or shipping list), to Inventory in order to Pick and Ship the product. Inventory sends this on to Shipping which actually does the physical shipment.

Next, we would like to model a subsystem discovered during subsystem analysis. For, say and model it:
As we identify the responsibilities of the subsystem in terms of its business rules, we place each business use-case on the IAccountManagement interface. Based on the functionality required to fulfil a use-case, we can define the operations on the interface. We would typically choose to have each use-case as an operation. In order to define a component, it is necessary to define the interface methods, which fulfil the use-cases it provides. However, this component definition is not complete: we require necessary and sufficient criteria to fully define and model a component specification. We have defined the necessary criteria for a subsystem to component mapping through selection of Reuse Levels (the subsystem level) and the use-cases (on the interface) the medium- or large-grained component is designed to fulfil. The next step is to define the component’s manners: the rules governing its behaviour in a given execution context. For this, we define a small (almost trivial) domain-specific language that models the typical interactions and contextual behaviour of the subsystem.

Let’s illustrate the need for an additional element: the component’s “manners.” Manners are the rules governing the behaviour of a reuse element such as an Interface, a component or a subsystem.

As we zoom into the subsystem, we find the following structure:

---

\[ \text{Subsystem: AccountManagement} \]

---

As we identify the responsibilities of the subsystem in terms of its business rules, we place each business use-case on the IAccountManagement interface. Based on the functionality required to fulfil a use-case, we can define the operations on the interface. We would typically choose to have each use-case as an operation. In order to define a component, it is necessary to define the interface methods, which fulfil the use-cases it provides. However, this component definition is not complete: we require necessary and sufficient criteria to fully define and model a component specification. We have defined the necessary criteria for a subsystem to component mapping through selection of Reuse Levels (the subsystem level) and the use-cases (on the interface) the medium- or large-grained component is designed to fulfil. The next step is to define the component’s manners: the rules governing its behaviour in a given execution context. For this, we define a small (almost trivial) domain-specific language that models the typical interactions and contextual behaviour of the subsystem.

Let’s illustrate the need for an additional element: the component’s “manners.” Manners are the rules governing the behaviour of a reuse element such as an Interface, a component or a subsystem.

As we zoom into the subsystem, we find the following structure:

---

\[ \text{Subsystem: AccountManagement} \]

---

The subsystem identified as the Account Management subsystem will be implemented through the interface IAccountManagement and the Façade AccountManagement, which implements the interface and encapsulates the subsystem in order to hide its complexity.

Probably the next design decision step is to find a place to inject the business logic; the functionality that drives the component’s behaviour. Initially, the business logic can of course be placed in methods on the Façade; with the Façade dispatching method calls to appropriate business objects in the domain, yet managing the overall interactions; through Template Methods or regular method invocations.

---

1 See section 3.2 “Components have Manners”.

---

9
As the number of business objects and their dependencies increase, the code within each of the methods on the Façade will become less easy to maintain or alter in accordance with the Open-Closed Principle [16].

Not only do we need to make regular adjustments to the collaborations managed by the Façade, but the need to reuse the individual business objects and components (inside the Façade) often tends to increase: between projects, product lines and business lines within a software development organization. If the objects directly reference one another, we are confronted with the Grape-vine Syndrome: a “grape-vine” of objects with interdependent references to one another. These types of object graphs are hard to reuse since their dependencies reflect a particular application context.

Thus, to decouple dependencies between business objects and components, we use the Mediator design pattern [22]. In this case, we might choose to delegate the execution of each major business use-case (which might contain sub-use-cases) to a Mediator, who, implementing the Reified Collaboration would decouple the dependencies between many of the business objects encapsulated in the subsystem. This is shown in Figure 11 below.

![Diagram showing the relationship between an Interface, Facade, Mediator, Business Object, and Reuse Element](image)

Figure 11

### 3.2 Components have Manners

In this section we discuss the rationale for the addition of rules to objects and manners to components and agents.

#### 3.2.1 Business Rules as First-class Constructs of the Object Paradigm

The core of our experiences across multi-domain projects has been that if business rules were first-class constructs of the object paradigm, then many projects would be delivered on-time and to budget, with lower maintenance and support costs due to adaptive and extensible architecture.

An object is traditionally thought to have identity, state and behaviour. We suggest that business rules and constraints be included as first-class constructs captured during each phase of the object life-cycle. Business rules are the core of the business and define and refine the structure and function of the business enterprise. However, to maintain competitive edge, a business must continuously explore new products and service offerings as well alter its tactical and sometimes strategic operations and workflow to handle competition, legislation, deregulation, mergers and acquisitions as well as expand on new market opportunities.

Changing these rules has been a development and project timeline nightmare: changes tend to be intrusive with unwanted and undetected ripple-effects and side-effects that are rarely found until regression and filed testing; both costly activities that does not enhance the struggle between software quality and time-to-market.
By applying variation-oriented design and separating the changing from the non-changing aspects of a business object, we can factor out its business rules from the business object, so that the entire object will change less frequently than the rules and business logic governing it [16].

Applying the Rule Object design pattern to our current configuration inside the subsystem, we have figure 12.

3.2.2 Component and Manners

Graduating from the world of fine-grained objects to the world of medium- to large-grained recursively composed business components, we find that a component is typically governed by a set of business rules which constrain its behaviour relative to a given application context. Thus, the rules governing the behaviour of a component, along with the necessary meta-data to implement non-intrusive type changes and additions, query the component for services at run-time and to define and externalise the rules governing the operation of the component as a domain-specific grammar governing the behaviour of the component is called a component’s “manners”.

Thus, a business component has manners: it recognizes how to act within a given context. Grammar-oriented object design is a way of defining component manners from the beginning of the life-cycle: starting with domain partitioning into subsystems whose domain-specific manners are captured as a grammar. This language of the domain is a form of its workflow and can be executed by parsing the grammar.

Use-case grammars frequently help in this process. They define the business level language that describes (components are self-describing, configurable, dynamically rule-changing executable entities) a component’s workflow. This is an externalisation of a component workflow in a set of production rules of a domain-specific language for each subsystem and the application as a whole.

Here is an example of a use-case grammar for one use-case in the E-bazaar example:

**Make an Online Purchase**

This use case describes the process of making an Online Purchase using the E-Bazaar System, a hypothetical online e-business system that presents catalogs of items for customers to order over the World Wide Web.

**Actors.** Customer (Online Customer via Web access channel), Shipping Vendor, Product Catalog, Credit Verification System, Address Sanitizer.
**Flow of Events: Basic Flow**

This use case starts when the actor initiates a Purchase for an Online Product in the E-Bazaar System after having browsed and selected items for purchase.

1. The system displays a product item that the user had selected. The user adds the selected product item to their shopping cart. This process (selecting and adding an item to the shopping cart) is repeated until the user is satisfied with the items in his shopping cart. In order to initiate the online purchase, the user selects Checkout.

2. The system displays an Order Summary (an itemized list of selected items and their subtotal). The user reviews the items and clicks “Continue”.

3. The system brings a billing and shipping address screen with the user’s default billing and shipping information. The user either selects the default or selects an alternative shipping or billing address and clicks continue.

4. The system displays the default payment method with the last four digits of the credit card number for user verification. The user accepts the payment method or selects an alternative payment method and selects “continue”.

5. The system displays an itemized Order. This includes items, subtotals, taxes, shipping charges and total. The user then reviews the Terms of Agreement and checks the “I agree” check box and clicks on “Submit Order”.

6. The system submits the order to the E-Bazaar and displays a confirmation of the purchase to the user, along with a thank you note and a confirmation number.

**Use-Case Grammar**

Online Purchase = {Identification, Presentation, Selection, Purchase, Confirmation, Order Fulfillment}

Identification = {Challenge User with Login, Verify UserID and Password}

Presentation = {Display Menu}

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

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

Checkout = {Complete Order Info}

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

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

Confirmation = {Send confirmation number to user}

Order Fulfillment = {Pick and Ship Order}

The use-case grammar is a new artefact, which combines the notion of a structured use-case with one of subsystem partitioning and domain-specific languages. Once a domain analysis is conducted and business language analysis is done, the key abstractions of the domain are partitioned in terms of interacting subsystems that may eventually be realized as software enterprise components which are “Composite Mediating Facades with pluggable Rule Objects.” Manners are assigned to each subsystem based on the Business Rules governing its behaviour.

Subsequent Variation-oriented Analysis is conducted to separate changing from non-changing aspects and features; verify what changes; handle changing aspects using patterns; partition the domain into subsystems and define manners for each subsystem and their interactions; and, use three layers of interface,
abstraction and concrete realization in the aggregate inheritance pattern. There have been numerous examples of highly successful implementations of software based on Domain-specific Languages (DSLs) [28] [29] [31]. A Business Language is an industry or business domain-specific language that characterizes the key manners or rules governing the behaviour in the domain’s partitioned set of subsystems. Grammar-oriented Object Design is concerned with identifying the Business Language for a given business domain, partitioning the domain into subsystems based on Subsystem Analysis, identifying variations within the subsystem manners and applying necessary patterns [Error! Bookmark not defined.] through Variation-oriented Analysis and Design, writing use-case grammars that define the Manners for the subsystem and the context in which it will interact with other component interfaces, once deployed and executing the control flow in a component framework through pluggable micro-workflows that implement the manners.

3.3 Variation-oriented Analysis and Design
After business modelling the business architecture and determining subsystems in subsystem analysis, we next conduct variation-oriented analysis and design to separate changing from non-changing aspects of the domain. These variations can be captured as text or more precisely as a domain-specific language which describes the variations of the domain.

The basic principles of variation-oriented design (VOD) are described briefly below.

**VOD1: Separate changing from non-changing aspects of the domain:**

![Figure 13](Fig13.png)

An example of this is when you split a business object up into its core and rule portions, each reified as their own classes.

**VOD2: Separate interface from implementation; program to interfaces:**
This principle of VOD is a best-practice that loosely couples the definition of an interface with this actual implementation. In the above example, a Person can act as a Teacher role. Instead of directly implementing the behavior for a Teacher (or other interfaces it might implement, it delegates out the actual logic to a TeacherImpl class.

**VOD3: Reify what changes**

This principle of VOD states that if some element of the domain is in constant flux, then it may be warranted to reify that element into a class (or higher layer of reuse). An example is the reification of collaborations, or the reification of business rules that tend to change more rapidly than the rest of the business object. This reification will usually be done in conjunction with the aggregate inheritance meta-pattern which factors the solution into three vertical levels of interface, abstract class and concrete classes.

**VOD4: Implement three-levels; use aggregate inheritance meta-pattern:**
This principle separates the interface that defines behaviour, from an abstract class that defines the abstract collaboration or reified collaboration between business objects in the framework; or can serve as such a home in the event that the cluster should become a framework. The concrete classes provide example implementation-level behavior.

**VOD5: Build Assets at each reuse Level**

Start component discovery rather than object discovery. Instead of identifying “nouns” or using merely CRC Card analysis, start with deciding on a Reuse Level and start identifying business rules for all Reuse Elements within that Reuse Level.

The Reuse Levels are: base class, inheritance hierarchy, aggregation hierarchy, cluster, framework, component, pattern, generic architecture.

**VOD6: Each Reuse Element has its own Manners**

For every Reuse Element we may identify a set of rules that govern its behavior within a given context. The manners of a Reuse Element include these rules of behavior in addition to the meta-data necessary to reflectively and adaptively self-describe the Reuse Element for run-time queries for service capabilities.

For each Reuse Element, such as a class, a cluster or a component, there can be business rules or manners (rules governing behaviour plus meta-data necessary for self-description) associated with it.

Applying these rules of VOD help partition the system into changing and non-changing aspects, making the application of design pattern easier and enforcing the open-closed principle over the design and implementation.

**4. Conclusion**

The process of building component-based architectures that are adaptive and resilient to change requires attention to five domains; not just deployment or implementation-time but also design- or build-time support from organization, methodology, architecture, technology implementation and infrastructure domains.

The Enterprise Component compound pattern can be used to encapsulate a framework to produce a technology neutral component architecture that can be mapped to any desired technology implementation platform or component model. The application of Rule Objects and Type Objects make the component specification and design resilient to change.
References


