Java Component and Sub-System Development

James Yao and Andrew Okimi

Aug 30, 2006

Table of Contents

Chapter 1 Component Development Architecture 4
Section A Component Packaging 9
Section B Object Oriented Design for Components 12
Section C Checkpoints at Component Review 17
Section D Component Management and Evolution 20
Section E Component Management for Managers 35
Section F Identifying Reusable Components for Grant Applications 46
Section G Construction of Reusable Components 49
Section H Examples of Reusable Components 54
Chapter 2 Sub-System Development 62
Appendix A 67
Appendix B 69
Appendix C 76

Chapter 1 Component Development Architecture

Overview

Introduction

This paper provides information and guidelines on component and subsystem development.

Components

Definition

The term "component" refers to an encapsulated part of a system. Ideally, it is a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture.

Standard

Reusable components must be assembly components that only couple with least other components. This applies the Object Oriented design (OOD) principle: low coupling, high cohesion.

Rationale

The application is built from discrete executable components that are developed relatively independently of one another, potentially by different teams. These are referred to in RUP (Rational Unified Process) as "assembly components".

Benefit

The application may be upgraded in smaller increments by upgrading only some of the assembly components that comprise the application.

Assembly components may be shared between applications, creating opportunities for reuse, but also creating inter-project dependencies.

Component Specification and Component Realization

Standard

Unified Modeling Language (UML) is used for component specification and component realization in the design model and other activities. A number of UML standard stereotypes exist that apply to the component:

- For example, **specification** and **realization** components can exist with distinct specification and realization definitions, where one specification may have multiple realizations.
- A component stereotyped by **specification** indicates a domain of objects without defining the physical implementation of those objects. It will only have provided and required interfaces, and is not intended to have any realizing classes and sub-components as part of its definition.
- A component stereotyped by **realization** indicates a domain of objects and also defines the physical implementation of those objects. For example, a **realization** component only has realizing classes and sub-components that implement behavior specified by a separate **specification** component.

Guideline

The separation of specification and realization allows for two separate descriptions of the component.

Specification

The specification serves as a contract that defines everything a client needs to know when using the component.

Realization

The realization is the detailed internal design intended to guide the implementer. To support multiple realizations, it is necessary to create separate "realization" components, and draw a realization from each realization component to the specification component.

Component Specification

Standard

Component specification should be designed as Java interfaces.

Dependencies between components, and between components and packages should be drawn directly in a component diagram at the component specification package.

Rationale

It should be possible to substitute any component realization for another as long as these components offer the same behaviour. We specify the required behaviour in terms of interfaces, so any behavioral requirements one model element has on another should be expressed in terms of interfaces

Checkpoint

Check that there are no circular dependencies between components and interfaces; a component cannot both realize an interface and be dependent on it as well.

Guideline

Component specification can be divided into different sets of interfaces for different users' views.

Relation Between Specification and Implementation

Standard

The component's dependency on implementation should not be exposed as the interface dependency.

Rationale

Dependencies between enclosing components are created when an element contained by one component uses some behavior of an element contained by another component.

We want to express improving reuse and reducing maintenance dependencies, in terms of a dependency on a particular interface of the component, not upon the component itself or on the element contained in the component.

Guideline

The designer should be total freedom in designing the internal behaviour of the component so long as it provides the correct external behaviour.

Rationale

If a model element in one component references a model element in another component, the designer is no longer free to remove that model element or redistribute the behaviour of that model element to other elements. As a result, the so-heavy-coupling system is more brittle.

Rationale

The component itself does not relate to the implementation; it serves as a design that an implementation must follow. The set of realizing classes, subcomponents or parts must cover the entire set of operations specified in the provided interface of the component specification. The manner of implementing the component is the responsibility of the implementer.

Component Usage

Guideline

A component can:

- be independently ordered, configured, or delivered
- be independently developed, as long as the interfaces remain unchanged
- be independently deployed across a set of distributed computational nodes
- be independently changed without breaking other parts of the systems
- partition the system into units which can provide restricted security over key resources
- represent existing products or external systems in the design.

www.ProcaseConsulting.com
Phone: (905) 856-7479
Fax: (905) 265-1930

Section A Component Packaging

Overview

Introduction

This section provides information on component packaging..

Contents

This section contains the following topics.

Topic	See Page
Overview	9
Component Sets	10

Component Sets

Definition

Packaging a component consists of two sets of components, a component specification set and a component realization set

Specification sets

A component specification set consists of:
- Component Specification, fully documented for the interfaces.
- Sample of the usage of this component is suggested.
- Design model to show the collaboration and dependencies. * It may be referenced into application design model.

Realization sets

A component realization set consists of:
- Codes and Unit Tests for most of classes
- Sample of the initialization/configuration and usage of this component realization
- Design model for this implementation (e.g. Class diagram) * It may be referenced into application design model.

Guideline

[Separate specification set and implementation set]. It is not uncommon that the specification set and the implementation set are packed together as a delivery. It is recommended that two sets be provided separately even when they are physically packed together in the same package.

Guideline

[Component specification]. The only way a component specification type can interact with another is through a provided interface. The specifications of the component must explicitly describe all interfaces that a client can expect. As well, the environment into which the component is assembled must provide the interfaces. The preferred component specification is one that is self-contained and symmetrical.

It is always worthwhile to write a specification for a component as perceived and used. Such a document greatly simplifies testing and reduces the time to evaluate the suitability of new and alternative versions of the component.

Continued on next page

Component Sets, Continued

Guideline

[Delay component binding]. When designing and building a component, the designer indicates to the others that this component will be composed only with their contractual interfaces. Actual implementations are selected at composition time or runtime (recommended). It is why factories and component containers are highly recommended while the "new ClassA" operation is forbidden for component binding.

Guideline

[Component implementation – Document Class Responsibility and Collaboration (CRC)]. CRC should be documented as a responsibility statement of each class and interface and the collated list of actions and collaboration for each class and interface.

[The specification of each action]. Action can be explained as "key method" or "use case of a class action". E.g. loan (money) is an action for a credit class while **setAccountNumber** may be not an action. The refinement of each action is recommended.

We suggest using pseudo-code as the comment before coding the implementation. It is a good practice in the coding (implementation) phrase to verify if the design is proper.

When this code is too much for a method, it is time to review the design.

Section B Object Oriented Design for Components

Overview

Introduction

This section provides information on reusable components and the use of Object Oriented Design (OOD)

Contents

This section contains the following topics.

Topic	See Page
Overview	12
Standards and Guidelines	13

Standards and Guidelines

Guideline

(Component Modeling): [based on framework]. (**Note:** Although framework is out of scope in this document, the component design has to be constrained under a certain framework/architectural constraints. Construct the specification for a particular problem by applying the generic framework and plug- ins in details for the problem at hand. On the implementation side, an implementation for the generic specification should be correspondingly customized for the specialized problem specification.

Guideline

(Component Implementation): [least inheritance more composition]. Composition (Polymorph coupling) is far more important as a design principle than Inheritance. It makes the plug-in point possible. As a result an assembly-friendly component is possible.

Guideline

(Component Modeling): [Specify component]. Write a specification for each component no matter what size. It is usual to write some experimental code and then go back to modify the specification to make sense of what has been done. The objective is to identify conceptual problems and obstacles early, reduce the misunderstanding between stakeholders, and reduce confusion in each developer's mind.

Guideline

(Component Modeling): [Specify component Views]. When specifying any component, there are usually several interfaces. The different interfaces are aware of only part of the component's state and behavior. Define partial views of a component, specifying behavior as seen from each interface.

This applies to the role coupling design. Each role of a component (even for each user of a component) has an interface as a view of the component. So that it is necessary to create a spec of a component/sub-system that has multiple interfaces, or many different users, without being forced into a single description.

When composing, the component model is the refinement of each of these views. Construct abstraction functions from the component model to attributes of views. If done fully, the abstraction function provides a way in which the assertions of the views can be checked against the implementation.

Continued on next page

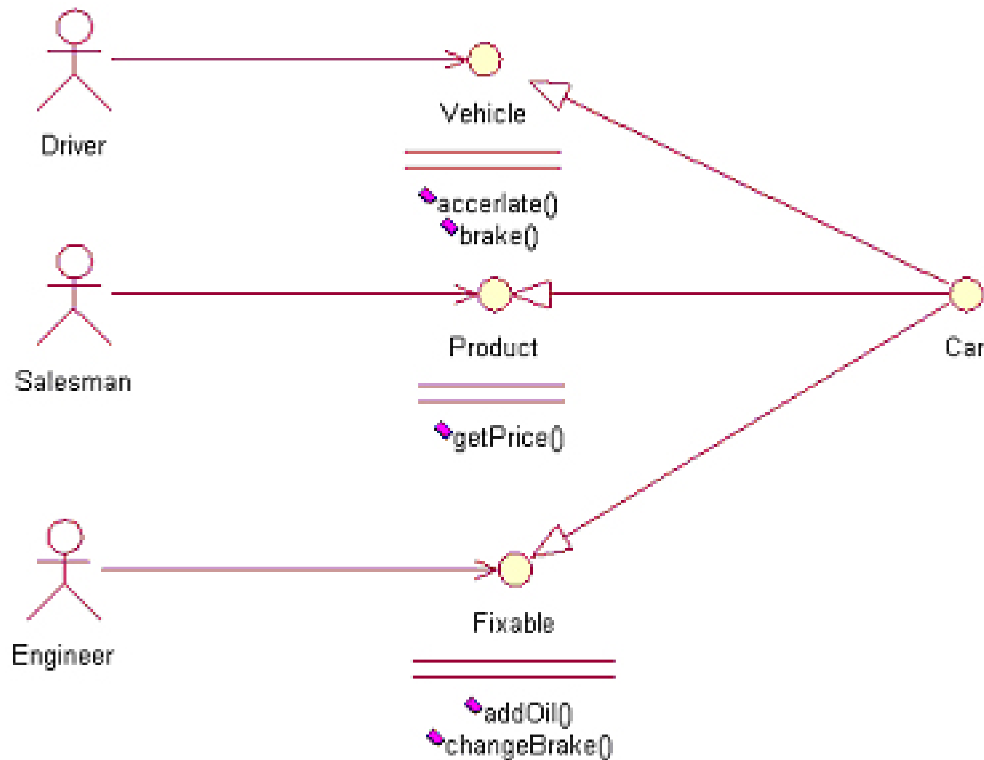
Standards and Guidelines, Continued

Guideline

(Component Modeling): [Role Coupling]: Design the interface (of the component in the component specification) that specifies only the operations used by that role and no more.

It is based on the fact that few clients will need all functions that a component provides or, few objects need to use all the functions understood by those with which it collaborates.

Example. In the figure below, instead of using Car as the only specification, it is encouraged to create more interfaces to satisfy each user. The driver likes to drive the car so that the designer had better keep "price" information from this role.



Continued on next page

Standards and Guidelines, Continued

Guideline

(Framework): [Selection of framework]: Reduce feasibility and scalability risks of technical architecture (framework) by determining/implementing the selection of all infrastructure components as early as possible. The technical architecture covers presentation, component technology, utility classes, data storage and access.

Guideline

(Component Implementation): [Connector]: Use the connector pattern for complex collaborations required by the components.

[Background reading material]. Assembly components need a set of standards. Connector standards represent the complex collaboration between components. It defines a variety of interactive mechanisms for various kinds of components and compositions.

Example. Event notifications, explicitly call and return, and so on. Rather than describe this complex collaboration from scratch for each interface, the pattern connector is invented to the pattern of collaboration that can be invoked wherever components are plugged together. Being able to separately and explicitly define connector types allows for further decoupling of inter-component dependencies from the component implementations. It also abstracts the structure of components better when they are composed later.

Continued on next page

Standards and Guidelines, Continued

Guideline

(Component Modeling): [Business model]: Making a business model is always a good starting point. Clarifying and building a clear vocabulary of the problem domain can prevent a lot of problems later on. (**Note:** we have highlighted that the **Glossary** – see Appendix B –is a mandatory artifact.) To develop the reuse-possible business component, the same business model has to be used for many projects within the same business. As a result, more of the component generalization and modeling can be well designed within a better vision. These business components have higher possibility to be well generalized and reused.

Section C Checkpoints at Component Review

Overview

Introduction

This section provides information when components are required to undergo a review process.

Contents

This section contains the following topics.

Topic	See Page
Overview	17
Design Checkpoints	18
Documentation Checkpoints	19

Design Checkpoints

Checklist

The following list provides checkpoints to be applied when reviewing the design of the component.

- No class dependency for component specification. Component interfaces only allows other components (interfaces), Value Object (immutable preferred).

Note: If a class is introduced, this class must be immutable Value Object only, e.g. String, Date, Point (int X, int Y) etc;

- If there is a dependency on another component, this dependency is based on component specification. It is not allowed to couple with any other component implementation.
- Check if interfaces have satisfied all callers, no less and not too much.
- Review CRC (Class Responsibility and Collaborations)
- Review ownership of each link between component classes.

Design questions

The following questions should be kept in mind when reviewing the design of the component.

- Is it seamless and continuous from the business model to implementation?
- Is the design a valid refinement of the specification, including documenting conformance?
- Can the designer explain all architecture decisions, including patterns and rules applied?

Documentation Checkpoints

Checklist

The following list provides checkpoints to be applied when reviewing the documentation of the component.

- Interfaces are fully documented for its caller, responsibility, collaboration, and dependency
- Sample/User manual is provided along with the specification. The configuration/initialization/factory interfaces are clearly listed.
- Component diagrams show clear layers, collaborations, dependencies information.

Section D Component Management and Evolution

Overview

Introduction

This section provides information and scenarios on component management.

Contents

This section contains the following topics.

Topic	See Page
Overview	20
Best Practices for Component Development Process	21
Non-Business and Business Components	23
Scenario: Reusing a Non-Business Component	24
Scenario: Reusing a Business Component	27
Integrating with Reusable Components	30
Version Control and Release Control	31
Tailoring to Meet the Deadline	34

Best Practices for Component Development Process

Introduction

Good design and careful implementation is the key to success.

The standards and guidelines in this document provide information to the designer that will help in building better components. This results in components that are robust, easy-to-use, and flexible as plug-ins, reusable, and so on. An experienced designer is also a key to the success.

Design focus

It is important not to focus too much on reuse when designing components. Efforts should be concentrated on basic component design and specification, decoupling the component, and controlling the balance between the ideal solution and the budget.

If the component is well designed, it should be very easy to reuse in the future.

Broad vision

Good decision is based on broad vision simply because there are many possibilities and trade-offs in component design. Vision includes the domain expert's knowledge and the designer's experience.

For the non-business component, design patterns, present technology, and experiences contribute greatly, and the decision is a rather simple one to make.

For a business component, the decision is not so easy to make because the domain expert often makes the wrong decision.

Continued on next page

Best Practices for Component Development Process, Continued

Business model

A business model is the key to reusing business components. However, a business model to cover the full business domain is very expensive, if not impossible, to build.

A business domain expert may help on decisions, but it is truly up to the designer to create solutions, before too much time and money is spent. The designer has to make the decision based on the requirements of the project and his experience. Focusing on the current project, up to the resource (budget, schedule etc), the designer has to find the trade off and, as always, design it at the best effort following the original vision.

Components to subsystem

The subsystem is ready to use as a product. Subsystems can be as big as a product, which may span from backend tables to representation layer screens. A subsystem is built/composed of components that includes some non-Object Oriented components, for example: HTML pages, a table, etc.

The bigger a subsystem is, the more difficult it is to fit a new requirement. If a large sub-system is to be reused, some features will need to be sacrificed. For instance, if the client wants to use a security subsystem for project A (and he does not mind that the login page has a totally different look and feel, and he agrees to use the database user-password check instead of the LDAP one), the client can save time and money.

Subsystem design should still follow the component development process. When the subsystem is made up of the components that are easy to reuse, the designer finds it much more flexible to reuse components from the subsystem for a new project. In most cases, the modification is required to the subsystem for the project. The black box reusability is ideal but often it is too expensive on the business decision. Most businessmen will not trade any business requirement for reusability. The practical case is to reuse the subsystem with some modification. During such modifications, the designer often takes a chance to generate some “ready-to-reuse” components.

Non-Business and Business Components

Non-business components

Non-business components refer to those components that are independent of the business model. The typical examples are framework components (e.g. JMS, JTA), utility libraries (e.g. logging, Map, List), pattern components (e.g. Pool, Observer), etc.

Non-business components are built to comply with general system requirements, are fairly standard, and have been used in some capacity in many projects. The requirements are so similar and general that usually the designer can either find the free component or specification, or the designer can build them easily with this knowledge. Non-business components are ready to reuse if designed by an experienced designer.

Business components

Business components are difficult to design as ready-to-reuse. The reason is simple. The business model is changing and the business requirement varies from project to project. What's worse is even the business experts sometimes cannot predict the change, or they usually provide too many out-of-scope models that are not based in reality (to the budget, usually).

The strategy for business components is to follow the project with the regular component development process. The reuse often is found, and clearer in another project. A scenario is provided below as an introduction.

Scenario: Reusing a Non-Business Component

Assumptions

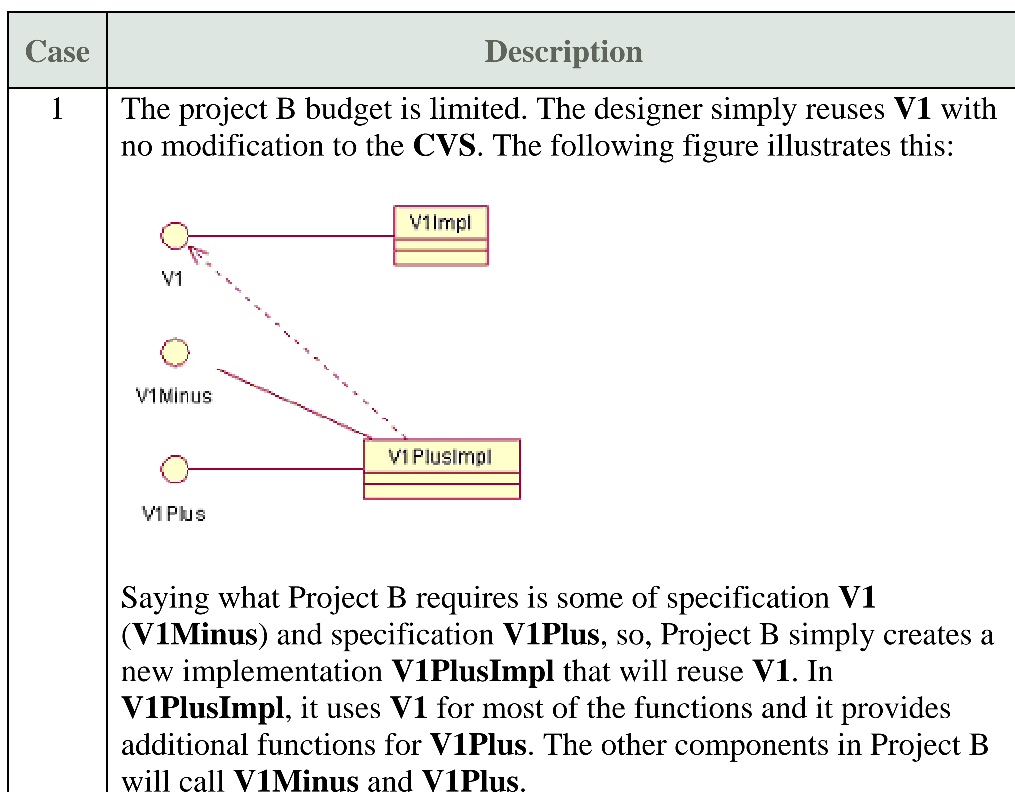
Assume that the terms Project A, Project B, and Project C represent different projects with time lines. Project A is built first, and then Project B is built next. The components **V1**, **V2**, represent different versions of the same component or a set of components.

Overview of Project A

In Project A, we built a non-business component **V1**. The designer knows **V1** will be reused in another project. In Project A, the budget and schedule are good enough to put **V1** into an archive of reusable components. As a result, the designer puts **V1** into a Control Version from Source (CVS) and writes a build for it. In Project A, **V1** is used as an external component.

Overview of Project B

In Project B, the designer finds V1 good to reuse, but project B needs more. To view the process of reuse, review the following cases of use:

Case	Description
1	The project B budget is limited. The designer simply reuses **V1** with no modification to the **CVS**. The following figure illustrates this:  Saying what Project B requires is some of specification **V1** (**V1Minus**) and specification **V1Plus**, so, Project B simply creates a new implementation **V1PlusImpl** that will reuse **V1**. In **V1PlusImpl**, it uses **V1** for most of the functions and it provides additional functions for **V1Plus**. The other components in Project B will call **V1Minus** and **V1Plus**.

Continued on next page

Scenario: Reusing a Non-Business Component, Continued

Overview of Project B (continued)

Case	Description
2	This Project B budget allows for some investment in **V1**. The designer can spend some time to upgrade **V1** to **V2** in the library (**CVS**): In this case, Project B will use **V2**. The version of component **V** in the library has been upgraded to **V2**. In this case, Project A may not consider doing the upgrade for it because Project A does not benefit from the **V2** interface.

Continued on next page

www.ProcaseConsulting.com
Phone: (905) 856-7479
Fax: (905) 265-1930

PROCASE

Scenario: Reusing a Non-Business Component, Continued

Overview of Project B (continued)

Case	Description
3	Project C is using **V2** and likes to refactor it for performance concerns. Possibly, the Project C team goes directly to **V1Impl** to do the refactoring. It may be a risk, but because **V1** is a "ready-to-use" component, all documents and auto-tests are there so that it saves time to prove **V1Impl** is robust and functionally correct for the **V1** specification. After upgrading, Version 3 for Component, **V, V3** is released with performance improvement. Project A may decide to import this version so that it can take advantage of the performance improvement. **Note**: The above figures show one of many possible designs for reuse.

Scenario: Reusing a Business Component

Assumptions

Assume in Project A that the library wants to build a component to put a book on hold. The book has a title, an author, ISBN number, and so on.

Overview of Project A

In this project, even the librarian has no vision about its future. So, let's focus on the business model and make "building this project" a high priority. If the design is good, it will create components that can be reused at its lowest cost.

Case	Description
1	The first model of Project A is illustrated as follows: HoldingMgrImpl HoldingManager holds BookImpl BookTable Book setHoldState() getName() getISN() **Note**: *BookTable is << Table>> in the database. In this design, this model makes perfect sense and it works perfectly. The project is on target and is working well. Some experienced designers may argue that **HoldingManager** is supposed to hold anything that is "holdable" instead of just **Book**. Let's assume this designer did the design in this way. We cannot argue he did not follow the component design guidelines. He, at least, followed all standards.

Continued on next page

Scenario: Reusing a Business Component, Continued

Overview of Project A (continued)

Case	Description
2	After a few years, the library starts to lend out videos, software, or even toys. After reviewing the Project A component, we find there are two differences. 1. In Project A, ISDN number is all that identifies an item in the library. (Because in those days, only book was stored in a library.) 2. In Project A, **HoldingManager** is designed to manage books only. To go about changing the component for reuse (Project B), the Book specification should be changed. Because the library is using a **scanCode** instead of ISDN number. Secondly, HoldingManager should be able to hold any item, not only a book. The new design is:

Continued on next page

Scenario: Reusing a Business Component, Continued

Overview of Project A (continued)

Case	Description
2 (cont'd)	From the above diagram, the **HoldingManager** specification is slightly changed. It should not couple with the **Book**; instead, it couples with **Item** and **Holdable**. Holding Manager's business logic is merely changed. Some code change is required to decouple it. And the testing should be smooth. This change could be very easy because the IDE usually did the most work for the developer. The book table has to be changed, at least one more field should be added: **ScanCode**. As a result, **BookImpl**, **Book** interface (**Book2** is shown on the diagram to explain that this **Book** is modified, and is not the same as **Book** in Project A) requires to extend from item to have scan code.
3	For DVDs, Videos, and Toys, a similar design could be introduced. In such a design, although we cannot reuse the Project A component as it is, the work is light because: 1. Most of Holding Manager's logic is reused. 2. Book Implementation (e.g. persistence, value object etc.) is reused. 3. **BookFactImpl** is introduced. But it is a wrap to the persistence components to **BookImpl** in Project A. From this point of view, Project B looks like an enhancement to Project A instead of a completely rewriting to Project A. It is the point we highlight: "All components are promised to be possible to reuse." It follows the investment rule that you try to achieve the best gain/cost radio.

Integrating with Reusable Components

Overview

Usually, the specification is stable when integrating with a non-business component. The new project may use inheritance on interfaces or even build a new component to use with the existing component. The design varies and there are many choices.

When integrating with the business component, it is often hard to avoid changing the specification. When the designer re-engineers the component, he will refactor the existing design during comparison between the old requirement and new requirement. As a result, the designer should not hesitate to do the reengineering and separate the known changing business logics.

Version Control and Release Control

Introduction

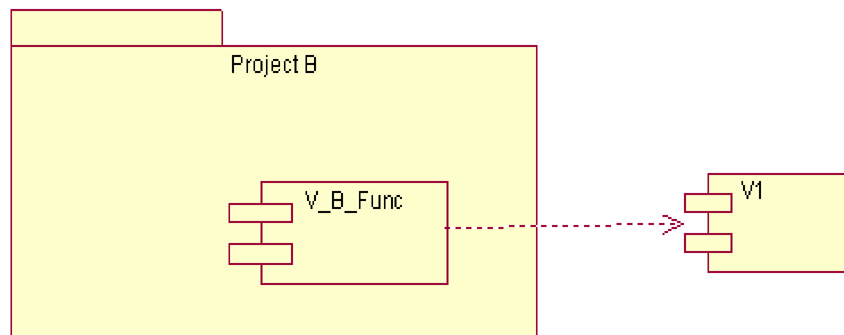
Ready-to-use component version control is necessary.

We recommend grouping them into projects in the CVS and releasing them as projects. Take **V 1.0**, **V 2.0** as the version of component group **V**. Project A and Project B stands for different projects with different time lines. (B is later than A.) CVS is one example of the repository (archive).

Versioning

Project A creates **V1.0** and **V1.0** is installed at CVS.

Project B finds **V1.0** useful, but Project B has a tight budget so it is decided to use **V1.0** with modification. See diagram:



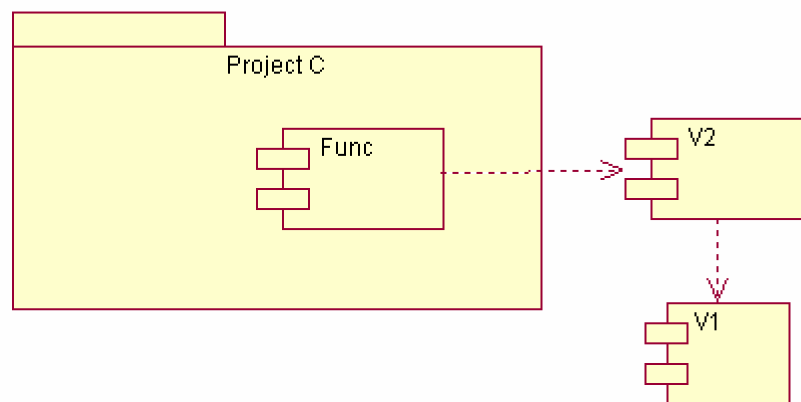
Continued on next page

Version Control and Release Control, Continued

Versioning (continued)

Project C finds **V1.0** and Project C heard about what happened to project B. Project C decides to invest in a build of **V2.0**. **V2.0** is **V1.0** plus some of **V_B_Func** and **V_C_Func**. But the **V2.0** back-supports **V1.0** fully.

Some specs may be marked as depreciated. Some new specifications may be added. Project C should prepare the release notes, change notes, add samples for new function, etc. See the following diagram.



In this design, **V2 depends on **V1**. In most cases, **V2** will replace **V1**.

Note:

1. From this case, Project B may be not able to import **V2.0** without changing code because Project C develops **V2.0** with concern on **V1.0** instead of **V_B_Func** + **V1.0**
2. How can Project C obtain the **V_B_Func**? It is copy and paste of whole component package on Project B, usually. This includes all documents, specs, codes, testing codes, and design models. The common alternative is that Project C develops **V2.0** based on Project B's code trunk (if permitted) with **V_C_Func** concern. In this way, after **V2.0** is released, Project B will import **V2.0** with Project C.

Continued on next page

Version Control and Release Control, Continued

Versioning
(continued)

Recommendations:

1. Project A does not have to import **V2.0** unless there is a clear benefit.
2. When Project X creates **V3.0**, if it is too much trouble to be back compatible to **V1.0**, Project X should create a new component X1.0 instead of **V3.0**. But, it depends. Sometimes, it proves to be good practice to create a **V3.0** that dumps some the **V1.0** features.

Tailoring to Meet the Deadline

Overview

Depending on the size of the project and the type of component, the designer should limit the resource to match the schedule and cost. It usually means tailoring something.

For example, if a component is not selected as "ready-to-reuse", it is unnecessary to package it for a separate release, and some testing and release-version-control can be omitted.

Another example is the framework components. Either some framework components are new to a developer (they need time to learn to use them) or framework components are a required extra resource to develop. For example, when the application needs a logging component **Log4j** can implement, for example: logging. In such a case, the designer can create quick simple log component specs and a quick implementation, which will simply print out the log information to the screen instead of spending time to train the team about **Log4J**. And later the designer can spare a developer who is familiar with **Log4j** to develop a **Log4J** proxy component following the Log Component spec.

But some things are not negotiable. For instance, do not try to save time on design and documentation, which can be treated as a one-time investment and a long-term benefit. Some designers try not to decouple the component too much because they think it will save some design time and development time. Unfortunately, it usually ends up with longer development and reengineering time.

Section E Component Management for Managers

Overview

Introduction

This section provides information about component management for managers

Contents

This section contains the following topics.

Topic	See Page
Overview	35
Introduction	36
Reusable Assets	37
Process	38
Management Concerns	42
Other Management Factors	44

Introduction

Success

Success in being able to reuse components does not depend on the good design only. The management of component reuse is a key factor as well.

What can be reused

There are many candidates for reuse, such as: a small function, a class, or a component to a sub-system, or a whole application. What's more, design models, domain models, requirements, and other artifacts can be reused as well. Once they are ready for reuse, they are an asset, and an investment. The term "reusable asset" covers all these areas.

Focus on: Business component

As described, reusing the business component is very difficult. In the following section, we will focus more on the business components and the business artifacts. The term "reusable business asset" is to cover these artifacts.

To be or not to be

Reuse is a decision. The goal for reuse should be set so that the commitment to reuse would be gained. Means of reviewing and measuring the success of reuse should be defined. Reuse is an investment, which means to give up some benefit now so as to gain more in the future.

www.ProcaseConsulting.com
Phone: (905) 856-7479
Fax: (905) 265-1930

Reusable Assets

Business model and Domain knowledge

The business model and domain knowledge is a valuable artifact. Such knowledge is specific and is key to reusable business components, which often depends on the domain model. This knowledge is recorded for the purpose of being placed in the repository.

To reuse a business component, or a subsystem, or an application, the business model has to be provided as the baseline.

Testing

The testing artifacts, that is, test cases, test reports and programs, are surprisingly ready for reuse.

Hints for business asset

The following can be considered as hints:

- Manage a reduced number of critical business assets instead of a lot of small assets.
- Use granular business assets (e.g. components, frameworks, domain models) rather than fine-grained ones (classes, procedures).
- Reuse artifacts that contain more lifecycle work products. Coding is only 25% development cost and reuse should not be limited to it especially when reusing business assets.

Process

Building a repository

The following suggested assets may be stored in a repository:

Part	Description
1. Meta Information	The reusable artifacts should be packed with all information to use. We recommend that it at least provide a "meta-information" or "detailed description" for search purposes. Also, a catalog is recommended when the repository grows.
2. Packaging	The packaging provides the following information: • Description for "what it does" • Description for "how to use it" • Qualification, testing, historic information.

Repository functions

The function of the repository is to define and maintain a catalog to help the reuser to:

- know where to find the reuse asset
- know what to do with a potentially reusable asset
- know if there is a reusable asset for their use

It also functions to provide central management, including configuration management, searching, enforcement reuse process, etc.

Repository features

A repository may provide the following features:

- Reuse asset identification and description
- Searching and browsing
- Insertion and configuration management
- History, change control, change notification, access control, reviews, feedbacks, interested party, etc.

Continued on next page

Process, Continued

Review of reuse

Each attempt to reuse should be recorded in the repository. A review of reuse is helpful. (e.g. success or failure, context, feedback, modifications, etc). Also, it would be an interested party for the future release. Furthermore, usage history will give confidence to potential reusers.

Reuse method

The following table shows three typical, suggested methods for managing reuse.

Types	Action
1	Integrate projects. Project produces application and reusable assets at the same time. This is recommended to avoid short-term pressure for project completion.
2	Focus on the production of reusable assets.
3	Focus on re-engineering. Application is developed with no reuse in mind. This is not recommended unless the development team is very strong or the budget is very tight.

Domain engineering

Domain engineering consists of analysis and development for a family application (or a bigger domain).

Domain knowledge

Domain knowledge is a good asset and is easily lost if it remains only in the minds of experts.

Domain analysis

The big domain analysis makes it possible to maximize reuse within this big domain (usually there will be dozens of following projects being built on this domain.) This is proven to be a good practice for common business components and reusable business assets.

But it is expensive (time consuming) and it requires a good understanding of the domain. What's more, it also requires the participation of all domain experts together with software engineer.

Continued on next page

Process, Continued

Recommend

Studying existing applications is a better way of studying a domain. It applies as application reengineering. It is recommended to study all existing application domain models along with the new application domain. It will lead to more high quality reusable business assets.

Testing

The following suggestions can be used for testing and qualifying reusable components.

- The reusable asset should be validated before putting it into repository.
- The reuser should check that the asset meets all requirements before reusing.
- The application quality review is often omitted, but the component quality review is a mandatory step. Sometimes we encourage executing the separate review for reusable asset for its reuse objectives.

Component maintenance

The asset owner should maintain the reusable asset but it is also up to management and the work force.

Often, the evolution of the component is left to reusers, who have no support for evolving the component. As a result, the consolidation of a component modification is rarely made and assets are consequently duplicated and rapidly diverge. If this happens, only a small benefit from reusing at development time is obtained. No benefit is derived from maintenance and other phrases. This is why there needs to be a “component owner” who can provide maintenance.

A trade off between limited resources and component maintenance must clearly be made during the design phrase. If a component is going to be modified for reuse, the application team needs to store the knowledge of this component and take the responsibility to maintain it.

Continued on next page

Process, Continued

Guidelines

The following guidelines can be used for maintenance of reuse components.

- The component owner in accordance with company strategy and domain evolution determines component evolution. The reuse support team is better qualified because it has better common vision. For the company that has limited resources, the application team should follow the design standard and should keep in mind that they are dealing with a common component instead of application component. A good practice is to provide an additional "design decisions" document to trace the component decisions as knowledge transfer for the future maintenance team.
- The Application Lead (AL) will be notified of the new release version, but it is up to the AL to make the decision if the new version should be applied.

Management Concerns

Business decisions

Reuse is a business decision rather than a software decision. The following questions should be posed when considering component reuse.

- What are the business objectives? How can reuse contribute to them?
- Is reuse the right solution? Is there a strong potential for reuse in the domain?
- Is my team ready?
- What are the risks and what is the plan?

Reuse risks

The following risks need to be considered when proposing a plan to reuse components.

- It takes too much time to understand and evaluate the reusable asset
- The reusable asset has low quality
- The component does not meet business (functional) requirements.
- The component does not meet technical requirements.
- The company may not be ready or the business is not appropriate.
- The team cannot determine if reuse is worth or if the direction of the reuse is correct.

Support functions

The manager should consider the following reuse support functions that are required.

- Operating a repository
- Reviewing the reusable assets
- Quality assurance and acceptance testing of reusable components
- Maintenance of reusable components
- Checking / reporting reuse performance
- Technical advice to producers and reusers
- Selection and procurement of reuse-related methods, specifications, tools, and processes
- Total involvement for reuse

Continued on next page

Management Concerns, Continued

Decision sequence

The following decision sequence for managers is proposed:

- Know the reuse potential
- Examine the reuse capability
- Change process as reuse implementation changes

www.ProcaseConsulting.com
Phone: (905) 856-7479
Fax: (905) 265-1930

Other Management Factors

Causes of failure

Some failures encountered during component reuse result from:
- Misconception of reuse. For example, management takes reuse as repository … the designer thinks reuse is equivalent to object oriented design.
- Ignorance of human factors
- Lack of commitment by top management

Total involvement (culture)

Total involvement is a cultural approach to process change that is people driven. It will be successful only when it is understood and welcomed by the people who have to use it.

Reuse suffers more than most from being wrapped in excessive technical mystique, preventing it from making sense to the majority of straightforward developers.

The experience indicates it is easier to gain total involvement in a small unit than a large one.

Also, it is the case that the reuse initiative starts from middle management without any awareness or commitment from top management.

Activities

The following activities can be used as a management strategy.
- Set business goal for reuse
- Establish change strategy to achieve the goal
- Determine performance indicators to track the goal
- Decide resource allocation

Continued on next page

Other Management Factors, Continued

Tactical management

The following tactics are suggested when working with a plan for component reuse.

- Designing changes to existing software development practice
- Planning the change process
- Tracking implementation
- Securing commitment to change
- Budgetary management
- Establishing and monitoring detailed reuse metrics

Work force

It is important to keep organizational issues in proper perspective. Without an appropriate work structure – one that 'goes with the grain' of existing work structures and existing incentives and thus minimizes the risk of generating resistance – reuse will be disadvantaged from the start.

But placing too much emphasis and making the design of work structures the process of reuse introduction and can signal the early demise of the project.

Final suggestions

The following two suggestions are provided to managers as the final word on component management:

- Focus on a single business domain and on a restricted team as a starting point.
- Establish domain modeling by studying all existing applications in GRANT

Section F Identifying Reusable Components for Grant Applications

Overview

Introduction

This section provides information that can be used when identifying components specifically fro the Grant Applications process.

Contents

This section contains the following topics.

Topic	See Page
Overview	46
Suggestions	47

Suggestions

Introduction

Suggestion of the following common components is based on limited analysis. The following suggestions are based on a quick analysis to high level of the GRANT project.

Without enough analysis, these listed components may not end with a truly reusable common component. The result depends on a lot of factors: budget, business model, clear requirement, decent analysis, business decision / design decision and so on. So, we do not want the potential designers to take it as a designing a result by mistake

Components list

The purpose of this components list is to provide a quick reference for potentially common components. The list may be considered a quick reference aid for the designer.

Note: Although this is a high level analysis of component reusability, it would be a misunderstanding for the reader to consider it to be inaccurate so that it is useless.

Exposed Service Level

- Some existing information system should be good candidates.
- IFIS Interfaces
- Plant Information

Sub-system Level

- Authorization System
- Code Table / Simple Table Maintenance
- Log Management system
- Tracking System
- Plant Management System
- Payment System
- Accounting System

Continued on next page

Suggestions, Continued

Components list (continued)

Common components

- Components exist in the sub-system. Any existing component is a potentially reusable component once built correctly.

The following special components require some attention:

- Side Bar Builder Package
- Auditing support

Library and Framework

Note: Out of scope to this document. Some examples are given:

- Component Management Framework
- System Management tool
- Logging

PROCASE

Section G Construction of Reusable Components

Overview

Introduction

This section contains information that describes an approach to constructing reusable components for grant applications. It also provides high-level estimates of cost and time for this construction.

Contents

This section contains the following topics.

Topic	See Page
Overview	49
High-level Time Estimates	50

High-level Time Estimates

Introduction

The following examples describe three scenarios of varying degree.

Example 1

A project requires 100 days of analysis and coding with straightforward development. Few documents are delivered. For such a project, the 100 days can be divided as follows:

- 20 days — Requirement gathering
- 10 days — Design HTML Page
- 40 days — Coding
- 10 days — Functional QA
- 10 days — Limited documents (Production support manual)
- 10 days — Deployment and implementation

This project will usually need around another 100 days for maintenance and small enhancements within a three-year period, if the same development group maintains it.

So, in this example, the project's cost is 200 days for 3 years. But it should be noted that such system is a "functional" system, fragile and hard to change. If, during the maintenance phrase, this project requires business improvement, bug fixes and so on, then the cost of maintenance will be much higher.

Example 2

If this project is built as a component-based development and all Object Oriented standards are enforced:

(Assume no reusable components are available)

- 40 days — Requirement gathering including HTML design → Generate UC, Domain model, etc.
- 60 days — Design → Generate Design model

Continued on next page

High-level Time Estimates, Continued

Example 2 (continued)

- 50 days — Coding → Generate Implementation model including unit test
- 30 days — Management → Plans, Reusable asset management
- 30 days — Testing → Test plan, test reports and etc.
- 30 days — Documents → User manuals, Production support manuals
- 10 days — Reviews
- 10 days — Functional QA
- 10 days — Deployment and implementation
- 20 days — Padding for unexpected cost for reusable components (e.g. learning curve)

Total: 290 days.

3-year maintenance program,
Maintained by the same development team: 30 days
Maintained by the production support team: 50 days

Example 3

(Assume some reusable component is available. Say coding time can be reduced by 30per cent.)

- 40 days — Requirement gathering including HTML design → Generate UC, Domain model, etc.
- 60 days — Design → Generate Design model
- 40 days — Coding → Generate Implementation model including unit test
- 20 days — Management → Plans, Reusable asset management
- 20 days — Testing → Test plan, test reports and etc.
- 20 days — Documents → User manuals, Production support manuals
- 10 days — days Reviews
- 10 days — Functional QA
- 10 days — Deployment and implementation

Total: 230 days.

3-year maintenance program,
Maintained by the same development team: 30 days
Maintained by the production support team: 40 days

High-level Time Estimates, Continued

Comparison

The following table compares various component-based design and development scenarios.

	Development Period (days)	Maintained by same development team for three years (light business enhancement)	Maintained by production support team	New enhancement and business logics
Straightforward Development	100	100	150	Assume it requires 100 days as a base for calculation
Component based design and development (No reusable component repository exist yet)	290	30	50	60*
Component-based design and development (Light reusable component repository is available)	230	30	40	40
Component-based design and development (Heavily reusable assets, including models, documents and etc, exist in the repository)	200	30	40	30

Note: * The estimate of 60 days does not seem large when compared to the 100 days. However, considering 100 days is spent mainly on coding and debugging, an estimate of 60 days is a good quantity to spend on the enhancement, reusable components engineering, reusable asset management, change and updating documents.

High-level Time Estimates, Continued

Conclusions

From the above table, the high-level estimate and effect of component-based design and development shows that it does not save money. If the budget is tight and resources are limited, the straightforward development is an option, apparently. Especially it works for some small projects --- mature requirement, no following enhancements and changes required.

The component-based development process provides the apparent benefit in the much higher quality of the system.

- Documents
- Robust application
- Low maintenance
- Low cost for the enhancement. This is very direct benefit. It is not uncommon that the bad-designed project has to be re-built from “zero” or “scratch” for some new features.

Lower cost for the future project. If the reusable assets are well managed, e.g. design artifacts, documents, etc, the future project’s cost will be much lower.

Section H Examples of Reusable Components

Overview

Introduction

This section provides information on examples of component reuse.
Note: The following are examples for reference only. The reuse of components in the real project will not be limited to the following examples. Also not all of following reuse examples makes sense if applies to the real project.

Contents

This section contains the following topics.

Topic	See Page
Overview	54
Exposed Service Level	55
Subsystem Level	56
Special Components Requiring Attention	58
Library	59
Framework	60

Exposed Service Level

IFIS interfaces

IFIS Interface Self Serve has exposed its service to OPS. This is a typical usage. It is not an Object Oriented system. It exposes the service via Oracle remote DB link and it defines business tables (views, actually) for any interested party to get the data. This is a good example for "exposed service" reuse. At such a level of reuse, via remote DB, RPC, CORBA, Web-service and etc, the system can be reused directly by any platform.

Plant Information

[Plant information management system (or other similar data sharing)]. This system, (if there is one), can expose some of its services for a searching function. If the system is good for the whole domain model, this system can even expose other business functions. This pattern fits to any other existing systems.

Subsystem Level

Authorization system

An authorization system can be a sub-system or a framework. It depends on the enterprise architecture. Usually J2EE defines some framework for this purpose. We suggest using it at the component level with some reusable presentation layer packages.

Code table

[Code Table / Simple Table Maintenance]. Any system has a lot of code tables (known as configuration table: e.g. XXXX_Type, city, province) and some simple table (e.g. person contact, email list, etc). These tables can be maintained in a generic way (few business logic). Some successful code table/simple table maintenance systems provide a full set from the front-end package to the backend DB operation.

Log management

[Log management system]. Log framework/library has been widely used. Most J2EE container also provides some monitor/management component for log. Log management system can be a candidate of reuse on the system level.

Tracking system

Tracking requirement is similar, usually, with few special business logic. Tracking system is usually a good candidate. The common tracking system works with monitor component to trace all the change to a table.

Plant management

[Plant management system]. GRANT projects have a common business entity, plant. Plant management system can be deployed as an independent sub-system for reuse.

Payment system

Payment system is another sub-system candidate.

Accounting system

Accounting system is another sub-system candidate.

Continued on next page

Subsystem Level, Continued

Reporting system	Reporting system can be integrated to any business application to provide report function.

Note: Components exist in the subsystem. Any existing component is a potentially reusable component once built correctly.

Special Components Requiring Attention

Sidebar	**[Sidebar Builder Package].** The sidebar is a representation level component to reuse. In the design, we suggest to provide a OO component to use this representation level component. Direct usage of Sidebar scripts/JSP codes is not efficient and friendly.
Auditing support	The auditing support is a special component. It is often designed as a non-OO component. Auditing support varied in the different projects. But it is usually a good candidate for reuse because the auditing rule is similar in a branch level. Duplicate record for auditing is one of solution, which will use DB function to auto-trigger the auditing records.
Business engine	Business engine e.g. business rule plug-in and validation. Some successful ideas can be found on the web site about business engine design. These ideas can be use to develop the reusable business component.

Library

Note: Out of scope with respect to this document. However, some examples are given:

Dropdown	[**Dropdown for code table fields**]. Most fields require the data from the code table or reference table. It is a good library or common component level reuse.
Email	Email has been provided as a library in J2EE. In an enterprise context, usually a better component can be developed to provide a reusable email component for business component users.
Auto-notification	JMS and Java Event have been provided as a library in J2EE. But it is better to develop a component for a business component. If necessary, the design should allow some business logic plug-ins to enhance the component or the design could create some servo-components for different users.

Framework

Note: Out of scope with respect to this document. However, some examples are given:

Workflow	Workflow is a common framework component provided by WAS. In the GRANT system, this should be designed as a framework component or common component for the different application.
Monitor	The server container usually provides it. Some business requires monitor framework as well.
Component	**[Component Container (Component Management Framework)].** It is not recommended developing a self-component container. But it is necessary to create enterprise architecture level component container framework so that the underlying solution is transparent to all components. The underlying solution is suggested as Spring framework and J2EE EJB framework.
System Management	**[System Management tool].** It is usually provided in the J2EE specification.
Logging	Logging is recommended as a standard or specification so that the log from all applications can be easily managed.

This page intentionally left blank

Chapter 2 Sub-System Development

Sub-system Specification and Realization

Definition

The subsystem is a way of building a system with a "divide and conquer" methodology. A subsystem provides functions and usually a subsystem contains some components on these functions.

A subsystem can be as large as an application. That is to say, an application can be a subsystem at some granularity. It is not uncommon that a subsystem covers a full set of functions from the front-end GUI/Web Page to the back-end table trigger.

When and how to use

A subsystem is widely used in the analysis phrase of a project. It can range from something as compact as a component to something as large as a sub-application and anything in-between.

Dependencies

When an element contained by a subsystem uses some behavior of an element contained by another subsystem, a dependency is created between the subsystems. To improve reuse, and reduce maintenance dependencies, we want to express this in terms of a dependency on a particular **interface** of the subsystem, not upon the subsystem itself, or on the element contained in the subsystem.

Exposed Service Reuse

Introduction

[Expose Services for Reuse]. As described on the reusable granularity, an existing system can be reused with the pattern: exposing services. The typical design is to use proxy components for each service. A subsystem can be used to package all proxy components for this existing product.

Exposed service reuse is a common technique employed to reuse an existing application. The current technology such as web service, Oracle Remote Query, etc., makes it easy to expose service to potential applications. But it often limits to the experienced business model.

Proxy strategy

Such a proxy strategy has proved to be good practice. A subsystem often represents an existing product, and the existing product exposes some interfaces to provide services.

GRANT application

In the GRANT, we can develop a plant management application, which will serve all family products for the purposes of plant information management. And after that, a subsystem is developed to proxy the exposed service, such as, Plant Name Search. Another example can be IFIS services.

Modeling a Subsystem

Introduction

All design standards and guidelines applied to component development should be applied to the subsystem as well.

Using a subsystem is a common architecture pattern. But overusing a subsystem to decompose the system will hide the overall problem for all the details of it.

It is common that the designer uses physically separate subsystems so heavily that it ends with a rigid stovepipe system, which has no form of reuse.

Restrictions

The subsystem provides a black box reuse scenario sometimes, and is a good example that shows all components inside the subsystem.

A subsystem is inevitably coupled tightly inside this black box with certain business logic and functions. It limits its direct black box reuse. Some attempts are made to reuse the subsystem by extending or plug-in replaceable components, but we suggest either reusing the subsystem as a black box or simply reuse the component inside it.

Examples of subsystems

The following are examples of subsystems:

- Authorization,
- Code table maintenance,
- Reporting,
- Format conversion (e.g. download as CSV, PDF, WORD, EXCEL …), Plant information management,
- Work flowing,
- Notification,
- Management Tools,
- Cron Controller,
- IFIS interface,
- Payment,
- Mail service

Continued on next page

Modeling a Subsystem, Continued

Example 1

A plant management subsystem and reusable component in this subsystem.

This example shows how reusing a subsystem as a black box will save a lot of effort. At the same time, the designer should never overlook the reusability of components in this subsystem. The guideline is “Trying to use a subsystem along with its components to achieve the most benefit from reuse”.

An existing plant management application is built as a subsystem. It provides front-end to backend functions to maintain all plants in the Ontario. FundsX Project is the new system, which will trace the plant’s application for a government loan. FundsX directly reuses the plant management application as a subsystem to add/update plant information.

This reuse includes reusing all web pages, components, and tables.
There is a use case in FundsX project: “Create a loan application”. This use case will create a loan application for a plant.

Description

To find how the business describes this use case, follow these steps.

Step	Action
1	An operator enters the **Create the loan application** page.
2	The operator selects province, city, and the name of the plant, and then clicks the **Search** button. **Result:** The Search returns an object.

Continued on next page

Modeling a Subsystem, Continued

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>3</td><td>The following table illustrates how the object may be displayed:</td></tr>
<tr><td></td><td>
<table>
<tr><th>IF there is</th><th>THEN</th></tr>
<tr><td>one plant showing</td><td>select it</td></tr>
<tr><td>more than one plant showing</td><td>select the dropdown list</td></tr>
</table>

Note: To implement this solution, the designer can use the “SearchPlant” component provided in the plant management subsystem to execute the search. Using this, the designer does not have to study too much on the Plant data model to design a Search component for this use case.
</td></tr>
</table>

Example 2

To reuse a non-Object Oriented component.

The plant management subsystem has the exact requirement to show the result of a plant, that is: one plant. It can be shown as a text string. More than one plant can be shown as a drop-down list.

Such a component may not be an Object Oriented component. In Java, it usually contains a Java Object Oriented component working with a tag library. Sometimes even a JSP fragment is a part of such a component. This special reuse scenario is presented here to try to highlight the guideline: “Anything can be reused if it is designed properly and packaged as a component”.

The designer studies the “description of usage” and “the example of usage”, which is provided in the package of this component. The designer can then reuse this non-Object Oriented component.

Appendix A

Naming operations

Prefixes

The following table provides examples of the naming convention using prefixes. The table also indicated their category, and adds descriptions.

Prefixes	Category	Description
General Prefixes		
is, can, has, will	Testing	Return a Boolean and test the state of the object
new	Creating	Create and return a new object from a factory that creates only a single type of object
init, setup	Initializing	These methods are called before you can use an object. Only a single init function should be called which can then be followed by whatever setup methods you need to change the default configuration of the object.
Type Specific Prefixes		
Find	Searching	Retrieve a single object or null if unsuccessful
select	Searching	Retrieve multiple objects or an empty collection
add	N/A	Add an object to a collection

Continued on next page

PRO CASE

Naming operations, Continued

Non-prefix

The following table provides examples of non-prefix naming convention

Operation Name	Category	Description
Any	N/A	Return any object that satisfies the request (findAny)
All	N/A	Return all objects that satisfy the request (selectAll)

Categories for method

The following list provides categories for method:

- **Constructing** a section and category. The constructors for the class.
- **Initializing** an additional method that should be applied directly after constructing the object.
- **Setup Methods** that can optionally be applied to an object but must be done immediately after construction and initialization and before using the object normally.
- **Validating Check** whether the current object is in an acceptable state (could also be under asking if this is possible after construction is finishing).
- **Asking.** Asking the state of the current object without causing any (visible) side effects. A pure function. ISE Eiffel 'Query'.
- **Testing.** An asking method that returns a Boolean value.

Appendix B

Glossary

Terms

The following definitions are ChiMu's distilling and reconciliation of the many concepts and work that have been contributed to Object Oriented Design. Many of the sources for these terms are mentioned in of this document. Some other notable sources are:

- The Dictionary of Object Technology [Firesmith+E 95].
- UML: The Unified Modeling Language [Rational 98]
- Design Patterns, especially [Gamma+HJV 96]

Term	Description
Adapter	An object that can convert an Interface of one Class to the interface that another Object expects
Architecture	A concepts, structures, and interactions of a system. Also, the description of a system's desired architecture before construction.
$Association_1$	A relationship between two Types that allow or require Objects of those Types to be linked.
$Association_2$	An $Association_1$, but must be between Types which have Objects with Identity. See Attribute and ValueObject.
Attribute1	A public property of an object that shows an aspect of the state of the object. Frequently there is a minimal collection of attributes that uniquely determine the state of the object. See also Property.
Attribute2	See BasicAttribute.
Attribute3	See Instance Variable.
BasicAttribute	An Attribute1 that takes its value from ValueObjects. This is as opposed to associations, which connect two or more objects with identity. A BasicAttribute is traversable only from the Object to the ValueObject.
Bean	An Object that knows about its own properties (can introspect) and has several other capabilities. Any Java Object can be a Bean but some Objects have more "Bean" functionality.

Continued on next page

Glossary, Continued

Terms (continued)

Term	Description
Behavior	The response of an object to a stimulus. An object's behavior is the answers it gives to messages both now and in the future. (See State).
Binding	Associating a client object to a database object, which turns the client object into a Proxy
Class	Provides a common implementation for a set of objects.
Container	A Registry but without the implication of a primary registration property (e.g. a "key").
DomainModel	All the static rules, constraints, and operations that apply to DomainObjects. The DomainModel can either be conceptual to help understand the behavior of DomainObjects or it can be implemented as DomainClasses
DomainObject	An object, which captures knowledge about a domain. DomainObjects allow a computer to inspect, imply, modify, and "reason" about that information in either very simple ways (the facts) or more complex ways (the rules and implications)
Extend	To define a new Type (called a Subtype) in terms of an existing type (called a Supertype). The new Subtype will have the same contract (operations) as the Supertype but can add new functionality: as either new operations or enhancement in capability to existing operations.
Extent	The collection of all instances of a Type or Class.
ExtentRegistry	A Registry that contains all the objects of a Type (the Extent of the Type). An ExtentRegistry is a close equivalent to a RelVar or Table in the context of Objects.
Factory	An object that can create other objects.
Feature	The Eiffel term for Operation where Operation includes both methods and attributes[1].
Forwarder	A proxy, which immediately forwards messages, over process and machine boundaries, to the RealSubject.

Continued on next page

www.ProcaseConsulting.com
Phone: (905) 856-7479
Fax: (905) 265-1930

PROCASE

Glossary, Continued

Terms (continued)

Term	Description
Framework	A strong partition of generalized functionality common to many parts of an application. Also, more formally, a collection of interacting classes that describes most of the behavior a client requires and can be subclassed and parameterized to customize and complete the functionality.
$Function_f$	A functor that returns an Object
Functor	An object that models an operation.
Functor	"An object that models an operation" [Firesmith+E 95]. For a Java implementation, a basic functor is an Interface with a single, generic, operation.
$Getter_f$	A functor that is designed to retrieve a value from an object (the first parameter)
Identity	The ability to tell one object from another object independently of whether their appearance (behavior) is identical
IdentityKey	A value that defines the RealIdentity of a Proxy
Immutable	Can not be changed after being created. Immutable objects can not be changed after they are created and fully initialized.
Inherit	To define a new Class in terms of an Existing Class (the Superclass) by starting with the Superclass's implementation and i-directi or adding to it
Instance	An object is an Instance of a Type if an object supports all the exterior requirements of that type (see "Is-A"). An object is an instance of a Class if it is implemented by that class
Instance Variable	A way to store encapsulated state information for a particular object. Instance variables are completely hidden within the Object, but they enable two objects of the same Class to have different external Behavior.
Interface	A description of a Type focused on the Operations that the objects can respond to.

Continued on next page

Glossary, Continued

Terms (continued)

Term	Description
Is-A	An object is a Type if an Object supports all the exterior requirements of that Type
Layer	A logical, horizontal division of a system that provides a particular system abstraction to the client above the layer.
Link	A connection between two objects which allows one or both to know about the other object. By default links are assumed to be i-directional in analysis, but they can be defined to be only traversable in one direction
Message	A stimulus sent to an object with a name and any parameters (as Objects) that the message requires. A message will cause the receiver Object to return an answer or nothing. In most Object Languages, the sender has to wait for the answer before continuing
Method	An implementation of an operation for a particular object/class.
Module	A base level subsystem: one which does not contain any other subsystems
Object	An identifiable, encapsulated entity that can only be interacted with by sending messages
ObjectBase	An ObjectBase captures the knowledge, operations, and rules required to usefully represent a particular part of the world in a computer. An ObjectBase contains all the objects that represent a particular state of your DomainModel and all the knowledge contained therein. Also called an ObjectSpace
ObjectShadow	The information needed to see that an object exists without any true representation of the real object. Relational databases could be considered to work with ObjectShadows: they record the information about an object but never have a real object to interact with.

Continued on next page

www.ProcaseConsulting.com
Phone: (905) 856-7479
Fax: (905) 265-1930

Glossary, Continued

Terms (continued)

Term	Description
Observable	An object that can be Observed. See Observer
Observer	An object that "looks at" another object (the Observable) and can respond to events in the Observable without the Observable being knowledgeable about the Observer
Operation	A description of the ability for an object to respond to a particular message and the contract/requirement for that message.
Partition	A vertical division of a system into areas of related functionality
$Predicate_f$	A functor that returns a Boolean.
$Procedure_f$	A functor that does not return a value.
Property	Synonym for $Attribute_1$ and sometimes for $Attribute_2$
Prototype	An object that is used as a template for creating other Objects
Proxy	An object that stands in for another object (the RealObject) and manages the client interaction with the RealObject
RealIdentity	The identity of the RealObject that a proxy represents instead of the proxy's independent identity. For proxies we are rarely interested in their own identity, we just want to know the identity of the RealObject on the server.
Registry	An object that remembers other objects and can search through and retrieve them through one or more properties. Usually the objects within a Registry are all of the same type
Replicate	A proxy which holds local state and performs local operations which are later propagated to the RealSubject
Role	The name of the "position" within a relationship an Object or Type holds. For example, a binary association has two roles that distinguish the two participants in the relationship

Continued on next page

www.ProcaseConsulting.com
Phone: (905) 856-7479
Fax: (905) 265-1930

PROCASE

Glossary, Continued

Terms (continued)

Term	Description
Setter$_f$	A functor that stores into an object (the first parameter) a value (the second parameter)
Singleton	An object that is the only instance of a Class
State	An abstraction to describe and simplify understanding an object's behavior. An object's behavior can be described as the answers it gives to current messages (which are determined by the current state) and the changes to its state caused by these messages
Strategy	An object that encapsulates an algorithm to be used with an Object
Stub	A proxy which acts as a placeholder for the RealObject and must become another type of proxy (for example, forwarder or replicate) when interacted with by a client
Subsystem	A division of a system into a cohesive unit of functionality (tightly related classes and internal subsystems) with a public interface and a private implementation
Subtype	A Type that extends another Type (called the Supertype)
Supertype	A Type that has been extended by another Type
Tier	A level on a hierarchy of processes over which a system is divided
Traverse	To move from one object to another by a Link. If a Link is traversable from an Object than that Object can get to (knows about) the other Object
Type	Describes a common exterior (public behavior) of a set of Objects. Can also be used to conceptually group and understand objects by their similar behavior

Continued on next page

Glossary, Continued

Terms (continued)

Term	Description
ValueObject	An object that does not have identity independent of its value. A ValueObject is immutable and should be considered identical to anything that it is equal to. Primitive data types in Smalltalk (most numbers, Symbols) are ValueObjects. Java Strings are very close to ValueObjects except they are not guaranteed to be identical for the same value (they would be if they did an automatic "intern()"). Java primitive types are not Objects.
Visitor	An object representing an operation that can be performed on the elements of an Object structure (frequently a hierarchy or sequence).

Appendix C

Java Reference list

List

The following is a list of manufacturers and their URLs where more information can be found on various Java coding standards and guidelines.

Manufacturer	URL
ChiMu	http://www.chimu.com/publications/javaStandards/index.html
The AmbySoft Inc.	http://www.ambysoft.com/downloads/javaCodingStandards.pdf
MOST	"The Most Important Design Guideline" by Scott Meyer
IEEE	http://www.aristeia.com/Papers/IEEE_Software_JulAug_2004.pdf
SUN Microsystems	http://java.sun.com/j2se/javadoc/writingdoccomments/index.html
SUN Microsystems	http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
Hammurapi	Open source software developed by Pavel Vlasov, Hammurapi is a tool that can analyze a code base against a set of design guidelines. There is a plug in available for Eclipse (WSAD). See: http://www.hammurapi.org
Bruce Eckel	*Thinking in Java* is a book written by Bruce Eckel. For more information, go to: http://www.BruceEckel.com