ObjectiveView

Objects, Components and e-Business Development for Software Professionals

Published by

OO consultancy – training – tools – recruitment

See http://www.ratio.co.uk for back copies

Sponsored by

ComponentSource™ The Definitive Source of Software Components

http://www.componentsource.com

ObjectiveView

Objects, Components and e-Business Development for Software Professionals

CONTENTS

UML Components 4

By John Daniels & John Cheesman

Ebiz Components 12

By Paul Allen

Let's Get Layered 21

By Mark Collins-Cope & Hubert Matthews

Component Distribution Patterns 30

By Philip Eskelin,with Kyle Brown & Nat Pryce

CONTACTS

Editor
Mark Collins-Cope
markcc@ratio.co.uk

Production editor
Mei Wong
mei@ratio.co.uk

Free subscription
Email delivery:
objective.view@ratio.co.uk
(subject: subscribe)

Hardcopy delivery:
objective.view.hardcopy@ratio.co.uk
(include full contact details)

Feedback/Comments/Article submission
objective.view.editorial@ratio.co.uk
Or join ObjectiveView @ Yahoo! Groups

Circulation/Sponsorship Enquiries
objectivew.view.editorial@ratio.co.uk

Editorial Introduction

Welcome to issue 6 of ObjectiveView. In this issue we have a major in-depth focus on component based development issues. As you'll see from the articles, component based development is a logical extension of object oriented development - components being implemented using object technology.

Two of our articles focus on component development process. John Daniels (of Syntropy fame) and John Cheesman give an overview of their component development process as described in their recently published book: UML Components. Paul Allen (widely recognized as a thought leader in CBD) discusses an e-business component process framework. Paul's process differs from the two Johns' in that its focus is on the management process as opposed to the development process. Taken together, the two articles provide a thorough insight in the CBD development process.

Moving onto component design and structuring issues, Hubert Matthews and myself discuss an architectural reference model to help in consideration of exactly which classes should be in which component. The ideas presented are a synthesis of many existing software paradigms, patterns and principles.

Last but certainly not least, Philip Eskelin presents a small pattern language (group of related patterns) based around designing components in a distributed system environment.

If you'd like to question of discuss any of the ideas presented in these articles - join the ObjectiveView discussion group. See below for details.

Happy Reading
Mark Collins-Cope

YAHOO! Groups

We would like to invite all ObjectiveView readers to join the ObjectiveView discussion group at Yahoo! Groups.

This discussion forum was created as a tool to encourage communication between ObjectiveView readers and authors, as well as between readers themselves.

Feel free to ask questions about articles, as well as about object & component technical issues of general interests.

TWO EASY WAYS:

1. Send an email to **objectiveview-subscribe@yahoogroups.com**

2. Go to **http://groups.yahoo.com/group/objectiveview** and click on the 'Join this Group!' button.

UML Components

DESPITE rapid growth in the use of component technologies such as EJB and COM+ there are few published practical processes for designing large-scale component systems. **John Daniels & John Cheesman** set out a development process to take you from a statement of requirements to a detailed specifications of the components you will need. Alongside the process, they'll explain how to support it using standard UML diagrams.

All software development projects follow two distinct processes at the same time. The **management process** schedules work, plans deliveries, allocates resources and monitors progress. The **development process** creates working software from requirements. Assuming you have these processes written down, you would consult your management process guide if you wanted help with setting milestones and your development process guide if you wanted help with allocating operations to interfaces. This article is concerned with the development process.

Although it hasn't always been this way, today the development process has to be subservient to the management process. This is because the management process controls project risk, and risk control is rightly viewed as paramount, even if the development process is compromised as a result. The favored management process nowadays is one based on evolution, where the software is delivered over a number of development iterations, each refining and building on the one before. The development process has to fit with that, so it isn't possible to specify everything, then design everything, then code everything, and so on, even if you wanted to.

Nevertheless, when in this article we describe the development process we do so without taking into account the constraints of the management process. We do this because we want the development process to be usable with a variety of management processes. It also helps to make the development process understandable.

Workflows

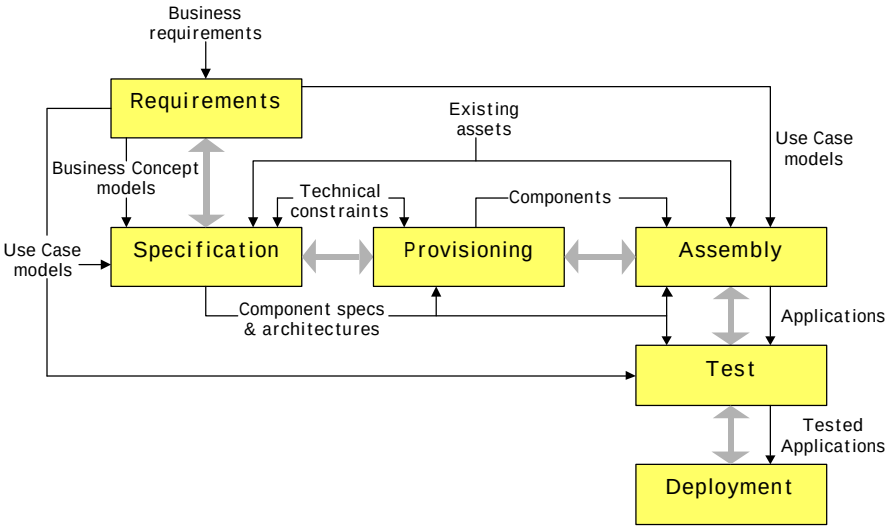


Figure 1 – The workflows in the overall development process

Figure 1 shows the overall development process. The boxes represent **workflows**, as found in the Rational Unified Process (RUP). In his book on RUP, Philippe Kruchten defines a workflow as "a sequence of activities that produces a result of observable value". The thin arrows represent the flow of **artifacts**—deliverables that carry information between workflows. Comparing the workflows of Figure 1 to those found in RUP, the requirements, test and deployment workflows correspond directly to those with the same names in RUP. The specification, provisioning and assembly workflows replace RUP's

Analysis & Design and Implementation workflows.

The specification workflow takes as its input from requirements a use case model and a business concept model. It also uses information about existing software assets, such as legacy systems, packages and databases, and technical constraints, such as use of particular architectures or tools. It generates a set of component specifications and a component architecture. The component specifications include the interface specifications they support or depend on, and the component architecture shows how the components interact with each other.

These outputs are used in the provisioning workflow to determine what components to build or buy, in the assembly workflow to guide the correct integration of components and in the test workflow as an input to test scripts.

The provisioning workflow ensures that the necessary components are made available, either by building them from scratch, buying them from a third-party, or reusing, integrating, mining or otherwise modifying an existing component or other software. It is also responsible for unit testing the component prior to assembly.

The assembly workflow takes all the components and puts them together with existing software assets and a suitable user interface to form an application that meets the business need.

Workflow Artifacts

In our process the requirements and specification workflows are responsible for producing a number of model artifacts. We expect the requirements workflow to produce a **Business Concept Model** and a **Use Case Model**. In specification we produce the **Business Type Model**, **Interface Specifications**, **Component Specifications** and the **Component Architecture**. Each of these is described below.

We've found it helpful to organize our model elements into a package structure which reflects these artifacts (see Figure 2). As we go we'll explain how UML is applied to model each of the artifacts.

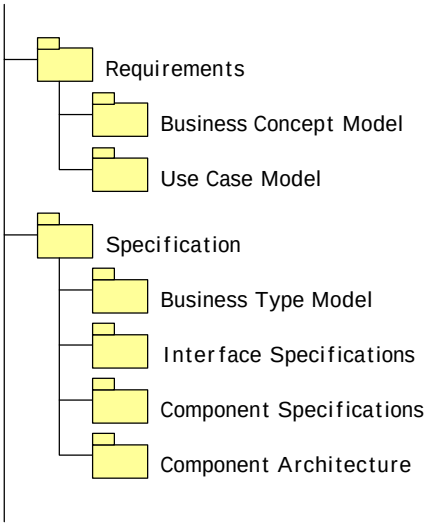


Figure 2 – Top-level organization of workflow artifacts

When we use the term "model" for an artifact, as in business concept model, we are using it in a general-purpose way simply to mean a self-contained set of UML model elements. We do not mean it carries the specific semantics of a UML model, which is a complete abstraction of a system.

Requirements Workflow

The purpose of the requirements workflow is to produce the business concept model and the use case model.

Business Concept Model

The business concept model is a conceptual model of the business domain concepts that need to be understood and agreed. It is not a

model of software, but a model of the information that exists in the problem domain. It is equivalent to the domain model in RUP. Its main purpose is to create a common vocabulary among the business people involved with the project. For example, if "Customer" means three different things within the business then you need to get this cleared up as early as possible so that everyone is working to the same set of terms with agreed meanings.

We use a UML class diagram to represent the business concept model, but it's important to remember that it is a software-independent model. If you wish you could use a <<concept>> stereotype for every class in this model, but since, from a packaging point of view it's kept under Requirements, and is quite separate from the Specification part of the model, we find in practice that we don't need to use a stereotype.
Business concept models typically capture conceptual classes and their associations. Association roles may or may not have their multiplicities specified. The model may contain attributes, if they are significant, but they need not be typed, and operations would not be used. Since the emphasis of the model is to capture domain knowledge, not to synthesize it or normalize it, you would rarely use generalization in this model. Similarly, dependency relationships would typically not be used.

Use Case Model

Love them or hate them, use cases are what the UML provides for semi-formal modeling of user-system interaction. A use case is a way of specifying certain aspects of the functional requirements of the system. It describes interactions between a user (or other external actor) and the system, and therefore helps to define what we call the system boundary. The use case model is the set of use cases you consider to be representative of the total functional requirements. You might start with the key ones, then add others later.

A perennial problem with use cases is deciding their scope and size.

The participants in a use case are **actors** and **the system**. An actor is an entity that interacts with the system, typically a person playing a role. It's possible for an actor to be another system but, if it is, the details of that system are hidden to us—we see it simply as a dull and predictable person. One actor is always identified as the actor who initiates the use case; the other actors, if any, are used by the system (and sometimes the initiating actor) to meet the initiator's goal.

In a use case the actors interact with the system as a whole, not some specific part of it. The system is viewed as a homogenous black box that accepts stimuli from actors and generates responses.

A perennial problem with use cases is deciding their scope and size. There seems to be no consensus on this issue but here's our view: to a first approximation we can say that a use case is smaller than a business process but larger than a single operation on a single component. The purpose of a use case is to meet the immediate goal of an actor, such as placing an order or checking a bank account balance. It includes everything that can be done now or nearly-now by the system to meet the goal. For example, if it is necessary to perform an electronic credit check with an agency before accepting an order we would expect the use case to perform the check and proceed. On the other hand, if goods need to be ordered from a supplier to fulfil the order being placed, the use case would end when the goods are ordered; it wouldn't wait for them to arrive. The subsequent arrival of the goods would stimulate another use case.

Use case diagrams, with their familiar stick figures and ellipses, are useful only as a catalogue or map. Use cases can be defined with a number of "extension points" to which extension use cases refer. This approach is useful for defining the broad structure of the use case in a diagram, but still leaves the real detail of the use case to be captured in textual **use case descriptions**.

A use case description contains at least:

- An identifying name and/or number.
- The name of the initiating actor.
- A short description of the goal of the use case.
- A single numbered sequence of steps that describe the **main success scenario**. Except in the case of an inclusion step as described below, each step takes the form "A does X," where A is an actor or "the system." The first step must indicate the stimulus that initiates the use case (i.e. what the initiating actor does to indicate to the system that it wants this goal met). The combination of initiating actor and stimulus must be unique across all use cases.

The main success scenario describes what happens in the most common case and when nothing goes wrong. It is broken into a number

of separate **use case steps**. The assumption is that the steps are performed strictly sequentially in the order given—unlike most business process languages there is no way of describing parallelism, which is one reason why a use case is typically smaller than a business process. Each use case step acts as a UML use case extension point. It is the anchor point from which an extend relationship to an extension use case may be defined.

Use case steps are always written in natural language. Use cases are a semi-formal technique and must be understandable by anyone familiar with the problem domain.

A use case description is a template for behavior that is instantiated in the environment of a deployed system each time an actor generates a stimulus. A **use case instance** either succeeds or fails in meeting the goal. A simple use case consisting only of a main success scenario is assumed always to succeed. Use cases can be elaborated by adding **extensions** that conditionally change the flow of the main success scenario to specify alternatives, and by factoring-out common parts into **included** use cases.

Specification Workflow

The specification workflow is rather tricky to explain since explanations tend to be sequential, whereas the workflow tasks are highly iterative. The various workflow artifacts have clear dependencies, but their development is incremental, with additions and modifications happening at every stage. We have attempted to summarize the workflow tasks into three "stages", as shown in Figure 4. They are called **Component Identification**, **Component Interaction** and **Component Specification**.

Note also that since we are staying management process neutral, we don't attempt to characterize the degree of "completeness", or other quality criteria, of these workflow artifacts. The management process phases will specify those.

Component forms

To understand the specification workflow it is necessary to understand how our view of a component changes during a project life-cycle. From requirements and specification, through design and provisioning, to assembly, deployment and runtime, the characteristics we want from a "component" vary. We can identify a number of "component forms," each form reflecting some aspect of a component during the development lifecycle. These different forms are shown on Figure 3.

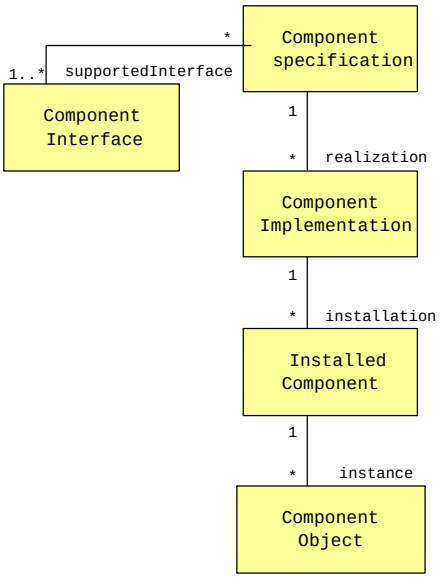


Figure 3 - Component forms

The purpose of each form is summarized in the following table.

Component Form	Description
Component Specification	The specification of a unit of software that describes the behavior of a set of Component Objects, and defines a unit of implementation. Behavior is defined as a set of Interfaces. A Component Specification is realized as a Component Implementation.
Component Interface	A definition of a set of behaviors (typically operations) that can be offered by a Component Object.
Component Implementation	A realization of a Component Specification, which is independently deployable. This means it can be installed and replaced independently of other components. It does *not* mean that it is independent of other components—it may have many dependencies. It does *not* necessarily mean that it is a single physical item, such as a single file.
Installed Component	An installed (or deployed) copy of a Component Implementation. A Component Implementation is deployed by registering it with the runtime environment. This enables the runtime environment to identify the Installed Component to use when creating an instance of the component, or when running one of its operations.
Component Object	An instance of an Installed Component. A run-time concept. An object with its own data and a unique identity. The thing that performs the implemented behavior. An Installed Component may have multiple Component Objects (which require explicit identification) or a single one (which may be implicit).

Although at first sight it may not be apparent why we need both interface and component specification as separate concepts, they actually perform quite distinct functions. The component specification forms the contract with the component implementer, assembler and tester. A component specification scopes the implementation unit, defines the encapsulation boundary, and consequently deter-mines the granularity of replaceability in the system. On the other hand, the interface forms the contract with the component client. It tells the client what to expect.

The component specification forms the contract with the component implementer, assembler & tester…on the other hand, the component interface forms the contract with the component client. It tells the client what to expect.

Specification artifacts

The specification workflow produces four artifacts.

- The **business type model** is an intermediate specification artifact, and not an output of the specification workflow. Its purpose is to scope and formalize the business concept model to define the system's knowledge of the outside world. This model is then the basis for initial component interface identification. While the business concept model describes the business domain as the business people understand it, the business type model captures exactly those aspects and rules of the business domain that the system knows about. The business concept model may be imprecise, but the business type model must be precise.
- The **interface specifications** artifact is a set of individual component interface specifications. Each interface specification is a contract with a client of a component object. Each interface specification defines the details of its contract in terms of the operations it provides, what their signatures are, what effects they have on the parameters of the operations and the state of the component object, and under what conditions these effects are guaranteed. This is where most of the detailed system behavior decisions are pinned down.
- The **component specifications** artifact is a set of individual component specifications. Each component specification is defined in terms of interface specifications and constraints. A component specification defines the interfaces it supports, and how their specifications correspond to each other, and also includes the interfaces it uses or consumes. While an interface specification represents the contract with the client, the component specification pulls these disparate client contracts together to define a single realization contract. This is where the building blocks of the system are defined.
- The **component architecture** describes how the component specifications fit together in a given configuration. It binds the interface dependencies of the individual component specifications into component dependencies, and describes how the component objects interact with each other. The architecture shows how the building blocks fit together to form a system which meets the requirements.

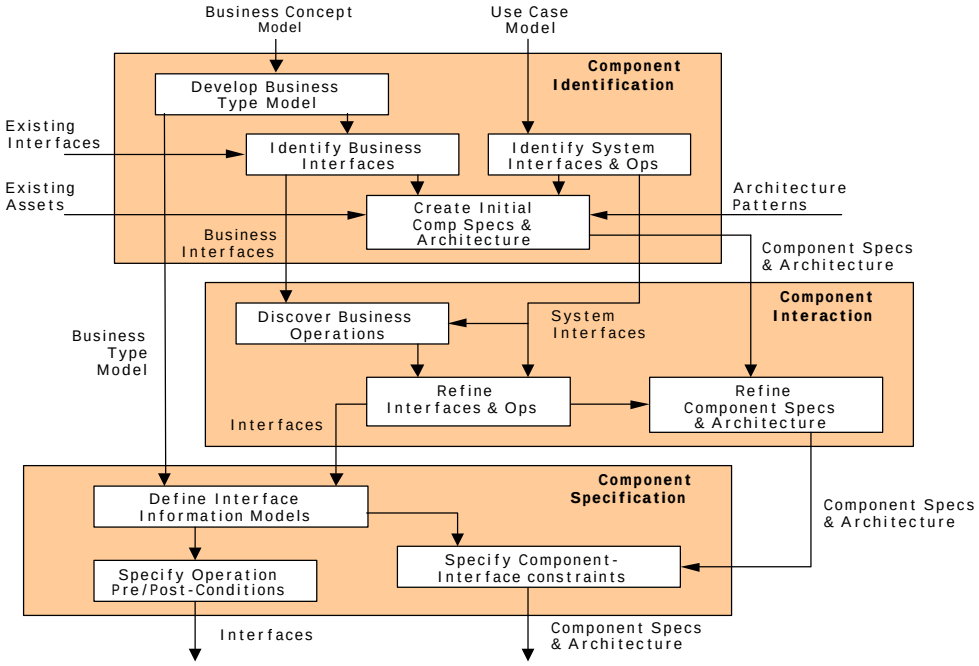


Figure 4 – The three stages of the specification workflow

Component Identification

The component identification stage takes as input the business concept model and the use case model from the requirements workflow. It assumes a layering of the application which includes a separation of system components and business components, where the system components provide a convenient façade for the system, based around the use cases, that simplifies access by the user interface or other external connections to the core business logic in the business components. The goal of this stage in the process is to identify an initial set of business interfaces for the business components and an initial set of system inter-faces for the system components, and to pull these together into an initial component architecture.

The business type model we mentioned earlier is an intermediate artifact from which the initial business interfaces are formed. It is also used later, in the component specification stage, as the raw material for the development of interface information models.

In addition to identifying business interfaces, the identification stage also makes a first cut at the operations that need to be supported by the system, and arranges them into system inter-faces. These operations are identified at first only by their name, but signatures and others details are added at a later stage. The system operations required are derived by examining the steps in the different use cases and deciding what the system's responsibilities are.

> The component specification stage assumes a layering of the application which includes a separation of system components and business components…

Put simply, the process in this stage is:

1. Copy the business concept model to form the initial business type model.
2. Scope the business type model by removing all elements irrelevant to the software.
3. Determine the core types. These are the dominant types in the business concept model. They have no mandatory associations to other types and have a business-level identity.
4. Define a business interface as the manager of each core type and all the non-core types dependent on the core.
5. Allocate responsibility for holding all relationships that exist between core types to one or other of the respective business interfaces.
6. Define one system interface per use case.
7. Examine each use case to identify operations on the matching system interface required to support it.
8. For each business interface defines a component specification that supports it.
9. Define a single component specification that supports all the system interfaces.
10. Draw a component architecture diagram that shows the "supports" and "uses" relationships between component specifications and interface specifications.

The business type model is drawn using a UML class diagram, using the <<type>> stereotype on all the classes. These "types" do not have behavior.

Interfaces are shown on the class diagram as classes with the stereotype <<interface type>>. Standard UML interfaces are not suitable—assuming your case tool implements them correctly—because UML interfaces can't have associations or attributes.

Component architectures are also drawn using UML class diagrams. Component specifications are drawn using the class symbol with a <<component spec>> stereotype.

Component Interaction

The component interaction stage examines how each of the system operations will be achieved using the component architecture. It uses interaction models to discover operations on the business interfaces. As more interactions are considered, common operations and patterns of usage emerge which can be factored out and reused. Responsibility choices become clearer and operations are moved from one interface to another. Alternative groupings of interfaces into components can be investigated. It is also the moment to think through the management of references between component objects so that dependencies are minimized and referential integrity policies are accommodated.

The component interaction stage is where the full detail of the structure of the system emerges, with a clear understanding of the dependencies between components, down to the individual operation level.

The process in this stage is:

1. For each system interface operation, trace through the control flow that will result from its execution.
2. As the control flow is defined, add the necessary operations to the business interfaces.
3. Reassess the component architecture.

The traces are drawn using UML sequence or collaboration diagrams.

Component Specification

The final stage of specification is where the detailed specification of operations and constraints takes place. For a given interface it means defining the potential states of component objects in an **Interface Information Model**, and then specifying pre- and post-conditions for operations, and capturing business rules as constraints. The interface information model is a representation of the apparent persistent (or more correctly, remembered) state of each component object that supports this interface, defined in terms of types & their associations. We say "apparent" because the model exists only to support the specification activities and might to translate directly into persistent state in an implementation of the interface. The pre- and post-conditions and other constraints make reference to the types in the interface information model, and the types of the parameters. In addition to these interface specification details, this stage also witnesses the specification of constraints that are specific to a particular component specification and independent of each interface. These component specification constraints determine how the type definitions in individual interfaces will correspond to each other in the context of that component.

For this stage the process is:

1. For each operation on each interface, define its pre- and post-conditions. As you do this, add to the interface's information model those elements needed to support the operation definition. The inspiration for the information model will come from the business type model, but any types reused must be copied into a package specific to the interface.
2. Considering each component specification as a whole, define any required constraints about the relationships between elements of the information models of the separate interfaces it supports.
3. Specify the particular usage the component will make of interfaces of other components when each operation it supports is invoked. This specification of usage creates a constraint on implementers of the component.

Interface information models are drawn using UML class diagrams. Information model types can be stereotyped as <<info>> if you so desire, but it isn't really necessary. Constraints, including pre- & post-conditions, can generally be written in natural language or OCL. Usage constraints can be more easily defined using sequence or collaboration diagrams.

UML usage

Figure 5 shows a summary of the use we make of UML diagrams.

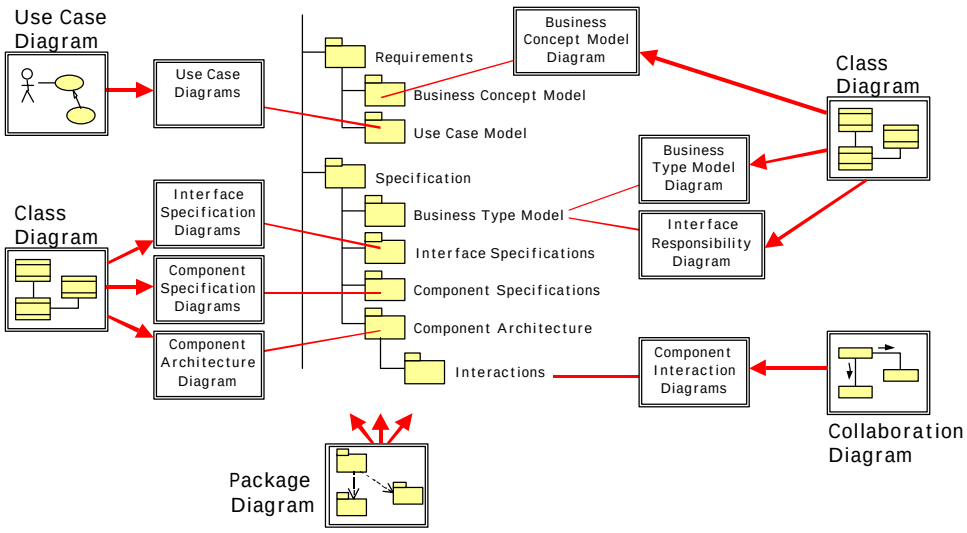


Figure 5 – The Component Modeling Diagrams

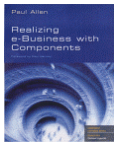
What next?

This article has, necessarily, provided only a brief overview of the process. It has simplified many issues, perhaps in some cases to the point of absurdity. But the overriding message we want you to take away from reading it is this: there can be a simple, logical and repeatable process for moving from requirements to detailed specifications of EJB, COM+ or similar components.

This article is adapted from UML Components, a book written by John Cheesman and John Daniels, and published by Addison Wesley. ISBN 0-201-70851-5. Copyright © Addison-Wesley 2001.
See www.umlcomponents.com for more details..

Contact John Daniels: jdaniels@cix.co.uk

Ebiz Components



CBD for e-business is neither a point solution driven nor a development process. It is an ongoing change management and integration process. **Paul Allen** puts us in the picture…

Organizations today do not have the luxury of starting each project from scratch using a clean white slate, as is commonly still assumed. Many organizations use processes geared to development of point solutions in the manner of a linear production line. This does not align well with today's needs for hybrid solutions to support e-business processes that not only cross divisions and departments, but also transcend organizational boundaries using the Internet. There is a growing need to solve these complex business problems by reusing and acquiring as much of the functionality as possible. Traditional software engineering processes were not designed to do this. New approaches are needed which are better suited to the new challenges, while building on lessons that have been learnt in streamlining software development in response to business needs.

This article sets the context for putting component-based modeling techniques to work using a practical component oriented process framework, so conspicuously lacking in most methods. Most current processes are too overwhelmingly detailed to be applied in practical enterprise settings. Our aim is rather to provide a workable project management framework for applying the modeling techniques described, largely by example, in this book. Our focus is on patterns and checklists, hints and tips.

> Oranisations today do not have the luxury of starting each project from scratch using a clean white slate

The CBD Process Framework

Development of e-business systems involves collaborative work of several different types of specialist with different areas of expertise; for example, business process consultants, software architects, legacy specialists, graphic designers and server engineers. We'll need a coordinating framework for dealing with these diverse skill sets and introduce a track-based pattern to help. It's also important to have a good idea of the kinds of deliverable that we can expect to produce. We describe a broad set of deliverables that work well on CBD projects. Techniques can then be applied in flexible fashion within our overall process framework of track-based pattern plus deliverables.

A Track-Based Pattern

A track-based pattern provides a coordinating framework for control of CBD & for organization of staff according to the roles played.

It's helpful to consider solution assembly and component provisioning working in parallel fashion as indicated in figure 1. This is often called "twin track development" (Allen and Frost, 1998). Consumers and producers follow separate processes geared to their respective needs of fast business solutions and high quality components. Solution developers seek to harvest components produced by the provisioning track. At the same time component provisioners seek to sow solutions as a basis from which to grow components.

Looking at the figure we can see that the twin track process is triggered in different ways. The assembly track is triggered by the need to produce a business solution in the shape of a timely business solution; for example direct sales over the Internet. Assembly involves searching for available components and, where necessary, raising requirements for new components from the provisioning track.

In contrast, the provisioning track may be triggered independently of specific solution requirements, by business reuse needs that provide the requirements for reusable components; for example commonly required business infrastructure components such as a product rule engine.

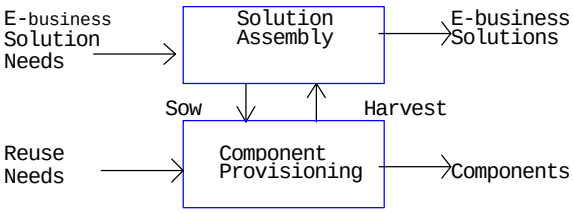


Figure 1 The twin track process

Increasingly there is pressure to reuse existing systems, packages and databases. Integration projects (Linthicum, 2000) use EAI software to remove data and process redundancy and introduce consistency across families of existing systems and packages, often implemented using diverse technologies. Note how reuse requirements are effectively filtered down to make use of legacy software as shown in figure 2.

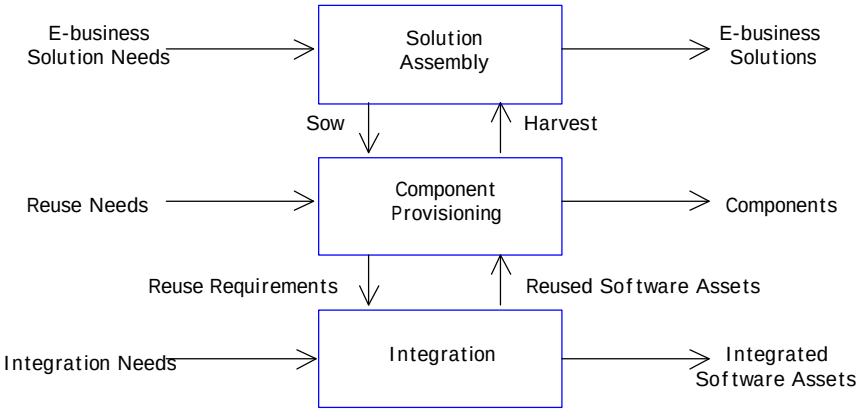


Figure 2 The twin track process and integration

The track-based pattern does not operate in a vacuum. e-Business process improvement provides the right business context for CBD, as shown in figure 3. Of particular importance for transitioning to e-business using CBD are the overall e-business improvement plan, which provides business direction for architecture planning and the business models, which focus on understanding specific processes requiring e-business solutions.

Note that the process is evolutionary and ongoing. Results from software projects are fed back to e-business process improvement for reassessment in the light of experience with e-business. Change must be managed. Similarly components are assessed with respect to architecture planning, in a process of progressive refinement. Architecture planning covers the high-level enterprise component architecture that provides a "big-picture" for projects to work to.

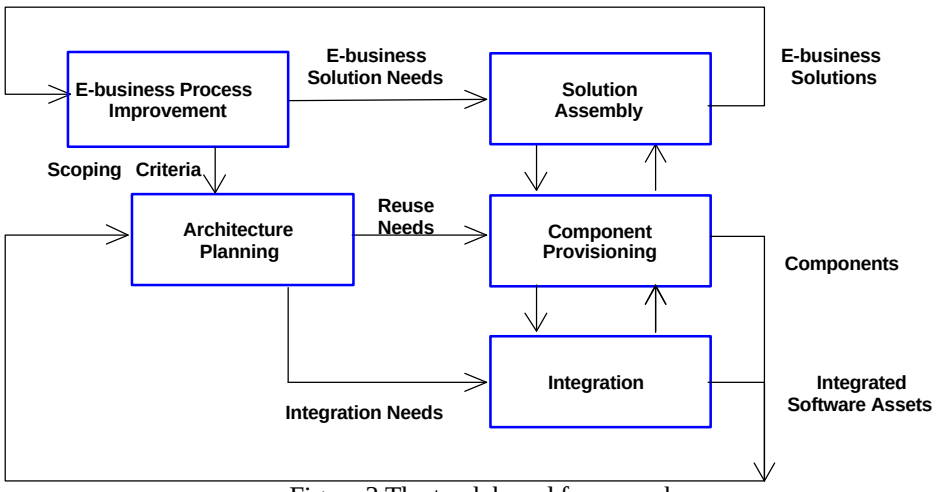


Figure 3 The track based framework

Underpinning the track-based framework are effective infrastructure facilities as shown in figure 4. These include support for component and Internet standards and configuration management. Some component management tools now provide component catalogues that hold component information within a repository and provide the ability to browse, install and register the components in harmony with model-driven approaches.

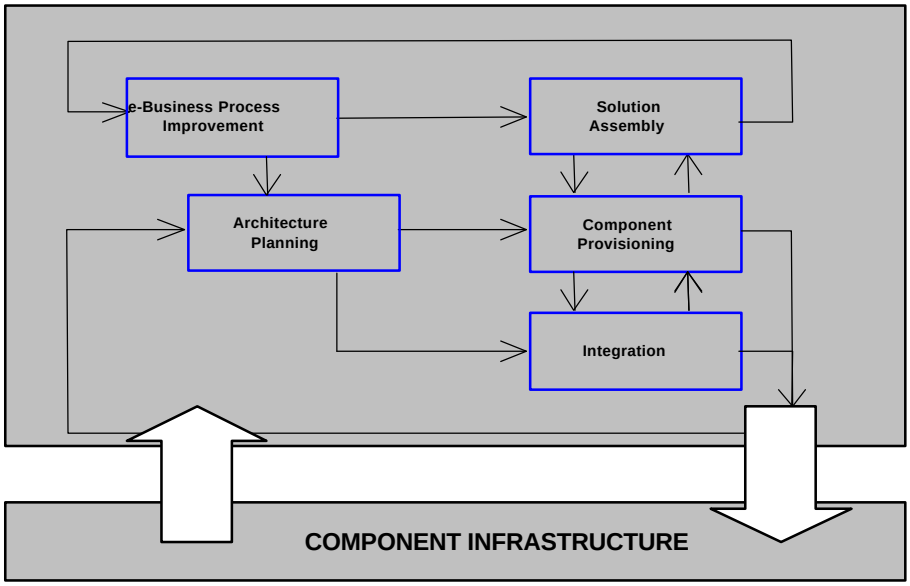


Figure 4 The track based framework and component infrastructure

Deliverables

There are many software processes that make the mistake of identifying prescriptive series of tasks that at the end of the day bear little relation to realities of everyday project life. The reason for this is that these kinds of process focus on the "how" before understanding the "what". Before considering techniques and activities we need therefore to focus on expected deliverables. Later in this article we'll show which types of models are expected within each type of deliverable.

Deliverables are needed to provide targets for projects, to measure progress and to ensure common understanding along the road to software delivery. There are "n" number of deliverable sets and accompanying techniques that we might identify across different organizations according to culture and industry type. Nevertheless we provide a generalized set in order to provide a context for the modeling techniques – feel free to adapt these to fit your own organizational needs. Five types of deliverables can be identified on typical CBD projects.

- e-Business process improvement plan
- Software requirements
- Component architecture
- Behavior specification
- Implementation

Implementation covers a number of further deliverables the details of which are outside the scope of this book.

Business modeling is used to understand the business requirements in a way that is easily understood by business people but at the same time usable with as little translation or rework as possible into a set of functional and non-functional requirements for some new software. **The e-Business process improvement plan** must include the business case for software requirements projects and scoping criteria for architecture planning (enterprise architecture).

Software requirements documents capture the scope of the proposed software together with enough information to enable a broad development schedule to be devised. The software requirements are usefully prototyped to facilitate user involvement and verification of correctness. Existing analysis patterns and software assets that might be used to meet the requirements are assessed. The latter includes existing systems, databases and available software packages, components, and interfaces. This is important for early identification of reuse opportunities and for understanding requirements and for assessing integration needs.

Component architecture documents work at two levels (project and enterprise). The underlying project & component architectures are progressively refined in the light of project experience and software delivery.

Behavior specifications provide a full and precise definition of the required software behavior. The externally visible software behavior is described, without dictating the internal design. However, constraints on the internal design are described. Such constraints

can take the shape of interface dependencies, nonfunctional requirements and invariants.

There are two types of behavior specification. **Interface specifications** specify the behavior of the interface from the point of view of its consumers. They include contractual specifications of services offered by the interface and a catalog of the information that the interface deals in (an "interface type model").

Component specifications specify behavior of sets of interfaces that provisioners are contracted to implement as software units. They include the interface dependencies, invariants and functional requirements that constrain the provisioner. The component specification can be likened to a compliance document; any implementation must comply with its component specification.

Interface and component specifications should be enrolled in catalogs and published using a component management tool.

Integration

Integration projects may exist at a tactical level and involve reuse of isolated legacy systems. Such projects may be non-invasive, seeking only to identify existing services that can be wrapped or adapted for use in component provisioning. On the other hand a legacy renewal project may be invasive and involve re-engineering parts of legacy systems in preparation for exposure of interfaces which are used in component provisioning projects.

Other integration projects are more strategic. However, the basic pattern of activities (planning, requirements, architecture, specification and implementation) remains the same. A strategic integration project should be looking to offer the integrated services for use in component provisioning projects. Legacy models may be used to assist in either tactical or strategic integration work. Tools that connect modeling capabilities with EAI middleware are extremely useful here.

Organizing Deliverables within the Track Based Pattern

Figure 5 shows the relationship between deliverables and track-based pattern. Software requirements, project architecture and behavior specification apply in a project context in any of the three delivery tracks. However, as we'll see a little later, de-scoping may occur at regular points through the lifecycle, resulting in hybrid projects. So, for example, a solution assembly project could branch into separate smaller assembly, provisioning and integration projects.

> Remember we are describing guidelines here, not rigid laws of nature.

For example, software requirements techniques may also be used within a business process improvement project. They are especially useful in conjunction with prototyping, as a means of scouting ahead to explore different designs. This is particularly appropriate for e-business systems where software becomes part of the very fabric of business and the distinction between business process improvement and solution assembly becomes naturally rather blurred.

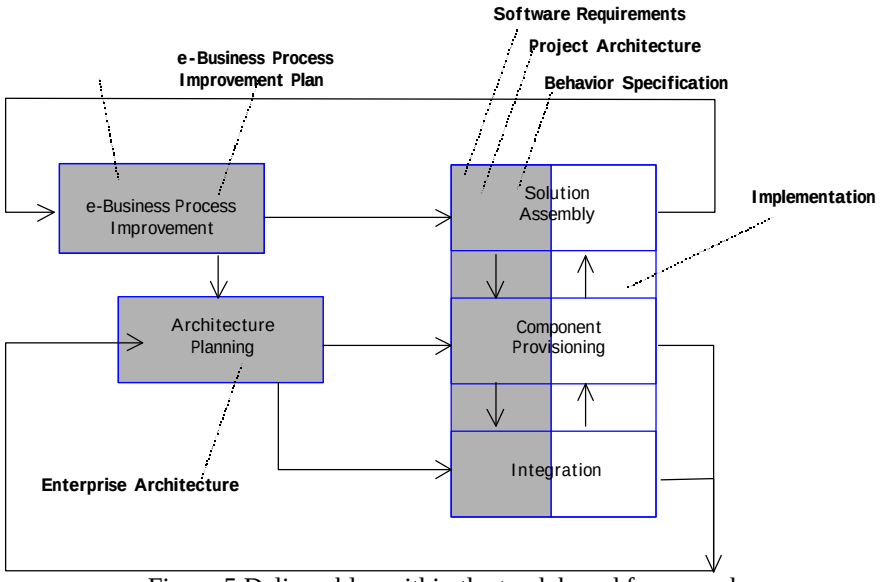


Figure 5 Deliverables within the track based framework

Reuse Checkpoints

Rather than the "Should I build or buy software?" choice that is presented in traditional processes, the questions in a CBD process now are:

- Which components should I buy and which do I develop?
- How do I scope and identify business components that align with my business needs?
- Is 80% of the solution (using pre-built components) at 20% of the cost/time acceptable to the business?
- Are there software packages that I can reuse and integrate to solve the problem?
- Are there legacy software assets I can reuse and integrate to solve the problem?
- Are there legacy models I can use to help address the problem?
- Are there component frameworks I can extend to solve the problem?
- Are there opportunities for outsourcing a component implementation?
- How can I pick and choose the best products, and integrate them in a mix and match manner?

It's useful at this point to consider some general guidelines for building reuse checkpoints into the track-based framework as shown in the table below.

Deliverable	Solution Assembly	Component Provisioning
e-Business Process Improvement Plan	Has the business problem been tackled before? Search for business templates, common business models.	
Enterprise Component Architecture	Are there cost-justified opportunities to reuse or genericize existing software assets or build frameworks, as part of the overall enterprise architecture? Consider architectural fit of identified software assets; consider architectural patterns. Examine dependencies and investigate overall feasibility.	
Software Requirements	Has the software problem been tackled before? Search for possible existing models, and frameworks that might help solve the problem. List candidates.	Can the problem be stated in more general terms? Conduct comparative studies with similar projects. Identify opportunities to generalize common features. Ensure analysis caters for sufficient diversity of contexts.
Project Component Architecture	Can existing software assets or frameworks be used to solve the problem? Consider architectural fit of reused asset. Examine dependencies and investigate overall feasibility.	Are there further opportunities to generalize software assets or build frameworks? Consider architectural fit of generic components or frameworks; consider architectural patterns. Examine dependencies and investigate overall feasibility.
Behavior Specification	Can existing interfaces be used to solve the problem? Extend and specialize interfaces.	Can generalized interfaces be provided? Generalize interfaces, consider analysis patterns.
Implementation (Internal Design & Acquisition)	Can existing implementations be used to solve the problem? Reuse implementation designs. Purchase implementations. Outsource implementations. Subscribe to virtual run-time services	Is the design flexible enough to cater for change? Design, purchase or outsource implementation for flexibility, consider design patterns. Ensure test plans cater for sufficient diversity of contexts.

CBD Process Themes

Another key feature of e-business systems is that, unlike traditional systems, they are subject to rapid change. Charles Schwab, for example, releases a new version of its electronic brokerage web site each month and evolves the underlying infrastructure to meet the demands of growing traffic. A balance must therefore be struck between the need for process guidance and the demands of rapid solution delivery. An effective process needs to exploit best practices but not constrain in an

overly bureaucratic way, as is unfortunately often the case.

> Good processes are meant to help, not hinder.

What is needed is an "adaptive" rather than an "optimizing" framework (Highsmith, 1999): "Rather than processes the model needs to focus on patterns. We need to move from a 20th century 'Command-Control' model to a 21st century 'Leadership-Collaboration' one." In this section we take a look at some themes of successful software process that help with the adaptive approach: iterative and incremental development, hybrid development and gap analysis.

Iterative and Incremental Integration

Iterative incremental processes are characteristic of object oriented development projects and have been well documented elsewhere (Kruchten, 1998; Jacobson et al, 1999). Our deliverables also evolve in iterative and incremental fashion as shown in figure 6. At the same time there is an important gear shift.

> CBD for e-business is neither a point solution driven nor a development process. It is an ongoing change management and integration process.

The process must recognize that architecture evolves in harmony with changing business needs. Also, you can only do as much architecture as the business will tolerate as reflected in the business case for CBD. Harvesting and sowing of reuse run right through the process from requirements to implementation as described in the previous section. A diversity of implementation options are involved, from bespoke design to outsourcing, from existing system integration to component purchase, and from framework extension to service subscription.

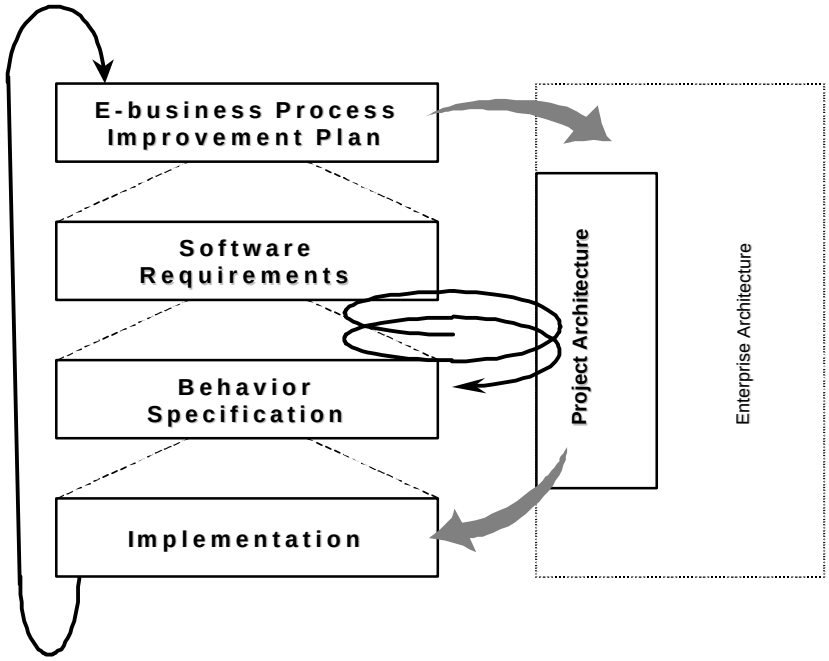


Figure 6 Iteration and de-scoping of CBD activities

e-Business process improvement planning sets scoping criteria for enterprise architecture and provides the business case for software requirements projects. Feedback is assessed on a regular basis from implementation of e-business solutions.

Refinement of software requirements, specification and project architecture is essentially iterative, with de-scoping at points of stabilization, providing a new context for drilling-down. Enterprise architecture provides an overall context that governs the evolving project architecture.

Once software requirements have stabilized, it may be that the project moves directly to implementation by incremental assembly. This is sometimes known as "fast-track" and is an evolution of RAD. Otherwise, specification moves to more detail, usually on a further scoped sub-set of requirements.
The refinement of specification and project architecture is again iterative. Once both have stabilized, the project then moves to

incremental implementation according to the provisioning strategy.

Hybrid Integration

Let's look at how the process supports a particularly key feature of CBD: hybrid integration. Suppose for example, that the first iteration of a complex project results in a software requirements document that gives rise to two sub-projects (P1 and P2), as illustrated in figure 7. P1 is a RAD project for fast-track assembly to serve one sub-set of requirements. P2 is a specification project to further investigate the architecture and specify components. P2 results in five implementation projects, say one for each of five components:

- P2.1 is another assembly implementation, though this time to provide a set of interfaces, in the form a process component (see 2.3) rather than direct user facing functionality
- P2.2 involves legacy integration work and the creation of adaptersm to implement components
- P2.3 is a straightforward wrapper on to a small legacy system
- P2.4 involves internal design of a component that the organization decides to build itself
- P2.5 is an outsourcing project

Once a component is implemented, further interfaces may be added or existing interfaces extended. For example, the fast-track assembly might be evolved to component status or further interfaces added to the adapter. This is an important feature of CBD: it is an evolutionary approach. Mixed implementations like this are an increasingly common feature in software development today.

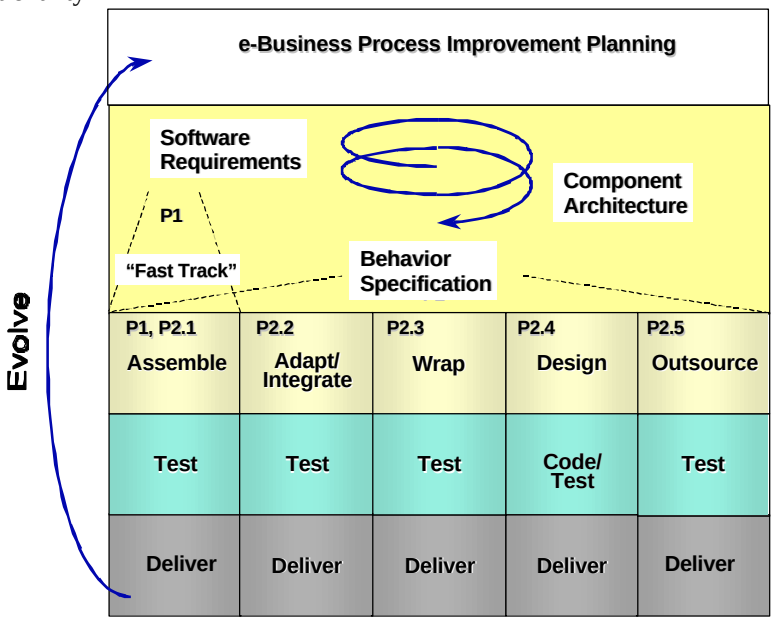


Figure 7 Hybrid Implementations within the CBD Process

We have already seen that applying CBD for e-business systems involves treating the interface as the unit of analysis and design. It should now be clear that the interface provides a natural delivery mechanism for incremental and parallel development.

> Gap analysis…assess the difference between stated requirements & existing components.

Interfaces are also a very convenient means of evolving an e-business solution. Early increments might involve incomplete and tentative interfaces designed largely to kite fly the solution. New, more complete and stabilized interfaces can be introduced, as the requirements become clearer. Client systems can be switched to the new versions of interfaces, as they become ready. Interfaces that are no longer used can be phased out. We must also address configuration management issues as the number of interfaces grows. Versioning of interfaces becomes a small discipline in its own right.

Gap Analysis

An important feature of CBD is the application of gap analysis to assess the difference between stated requirements and existing components. A gap analysis results in a provisioning strategy: as we saw in the

previous section recommendations that may include a hybrid of different options.

It is important to understand that the gap analysis might involve realigning the stated requirements; making a compromise to requirements in order to deliver the solution faster and cheaper. Even in an extreme case where use of a package is mandated as part of the e-business improvement plan, a gap analysis is still important if only at a high level in order to understand possible shortcomings of the package and to manage expectations, particularly of business people. After all the package may severely constrain the component architecture forcing integration of business components around its capabilities as opposed to those ideally required by the business!

Gap analysis is applied at decreasing levels of abstraction through the process, not just once at the specification phase as is sometimes mistakenly thought. CBD affects software's entire lifecycle.

Techniques versus Deliverables

We can now return to our basic set of steps for an interface based approach. Each step is addressed by a different family of techniques that are applied in iterative and incremental fashion:

- **Business Modeling:** Understand the business needs in terms of strategy, organization, process and information.
- **Use Case Modeling:** Scope and identify the business context of the software behavior.
- **Business Type Modeling:** Identify and describe business concepts to be managed by the software, and evolve to help identify candidate interfaces.
- **Interaction Modeling:** Examine dynamic interplay between software concepts to realize required behavior. Identify and partition the behavior to be exhibited by the software, assign responsibilities to and evolve candidate interfaces.
- **Architecture Modeling:** Understand software dependencies and adjust responsibilities of interfaces. Verify technical feasibility. Group interfaces into components and partition the software into deployable units.
- **Specification Modeling:** Specify the interfaces in terms of behavior and information. Specify groups of interfaces to be implemented as components, in the form of component specifications, include constraints. Enroll and publish specifications in catalogs.
- **Internal Design Modeling:** Develop or acquire component implementations.

Figure 8 illustrates the overall relationships between the techniques and shows how techniques relate to typical project deliverables. Internal design modeling is not shown here as it is outside our scope.

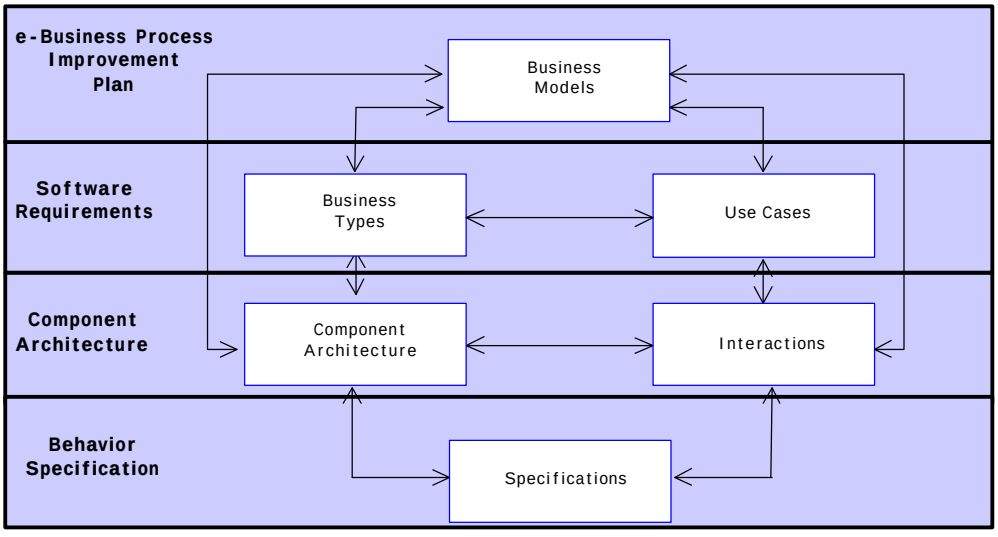


Figure 8 Technique types in relation to typical project deliverables

Technique Overview

The business models provide a context for modeling business types and use cases Business process flow diagrams help to identify use cases. Business concept models provide an initial pathway toward developing business types. The business models help to scope the component architecture and set a context for interaction modeling. Conversely information unearthed as a result of any of the

four software related techniques may result in a revision to the business models.

Modeling of use cases and business types is two-way. Both help in understanding software requirements. The business type model must support the use cases. Use cases help unearth new business types, attributes and associations. At the same time the business type model should reflect requirements not covered by use cases, such as business policy, business rules and information requirements.

Business type modeling drives both interaction modeling and architecture modeling (and vice-versa). Interfaces are initially declared by consideration of the business type model. An interface should manage cohesive sets of business types. Dependencies between the interfaces must be reflected in the component architecture. Interfaces are declared on interaction models, which help to further refine understanding of the interfaces.

Use case modeling drives interaction modeling (and vice-versa). Use cases are refined into lower level use case steps and the required types, and interfaces, declared on interaction models. (Note: In fact it is possible to proceed direct to the interactions without prior use case modeling). Thinking through interactions in this way may also cause use cases to be adjusted.

Interaction modeling and use case modeling drive architecture modeling (and vice-versa). The architecture provides interfaces that are declared on the interaction models. Conversely the interactions (and use cases) are "played through" and test out the architecture, causing adjustment of interface responsibilities, questioning of dependencies and new interfaces and dependencies to be identified.

Interaction modeling drives specification modeling (and vice-versa). Operations required to support the interactions and attributes and types required by the interactions are declared in interface specifications. Interaction modeling also helps identify constraints and dependencies recorded in component specifications. Conversely, thinking through specifications in this way may also cause the interactions to be adjusted (for example, perhaps an operation is moved to a different interface).

Architecture modeling drives specification modeling (and vice-versa). The interface specifications and component specifications must support dependencies identified in the architecture. Conversely, thinking through interfaces in this way may also cause the architecture to be adjusted.

Summary

In this article we have summarized the key features of an effective process for e-business. We have not attempted to catalog or structure this information more comprehensively. This is partly for reasons of scope and partly because CBD for e-business is a highly skilled, creative and adaptive process. Too often processes are produced for projects that ignore this fact and attempt to rigidly prescribe the development process to an extreme level of detail.

Nevertheless some structure is essential for planning CBD and for managing the diversity of potential provisioning routes and skill sets. A process framework and set of deliverables provide a good starting point together with checklists and guidelines such as those we looked at for reuse. The deliverables also help to provide a context for using the different CBD modeling techniques.

References

Allen, P. and Frost, S., *Component-Based Development for Enterprise Systems: Applying The SELECT Perspective*, Cambridge University Press - SIGS Publications, 1998

DSDM Consortium, DSDM Version 3, Tesseract Publishing, 1997.

Highsmith, J., Adaptive Management: Patterns for the e-Business Era, *Cutter IT Journal, vol 12 no 9*, Sept 1999.

Jacobson, I., Booch, G., Rumbaugh, J., *The Unified Software Development Process*, Addison Wesley Longman, 1999

Kruchten, P., *The Rational Unified Process: An Introduction*, Addison Wesley Longman, 1998

Linthicum, D., *Enterprise Application Integration*, Addison Wesley Longman, 2000

Martin, J., *Rapid Application Development*, Macmillan, New York, 1991.

Stapleton, J., *DSDM - The Method in Practice*, Addison Wesley Longman, 1997

This article is adapted from Realizing e-Business with components, a book written by Paul Allen, the Principal Component Strategist of Computer Associates & editor of Cutter's "Component Development Strategies" journal, and published by Addison Wesley. ISBN: 0 201 67520 X.

Contact Paul Allen: PAUL.ALLEN@ca.com

Let's Get Layered

Mark Collins-Cope & **Hubert Matthews** discuss an **Architectural Reference Model for Object & Component Based Development**.

THIS article proposes a reference architecture for object-oriented/component based systems consisting of five layers. Our purpose is to show how this model helps us to understand the overall structure of a system, how layering helps to clarify our thoughts, and how it encourages the separation of concerns such as the technical v. the problem domain, policy v. mechanism, and the buy-or-build decision.

Assuming an application is made up of a number of components, the layering we propose is based on how specific to the particular requirements of an application each component is. More specific (and therefore less reusable) components are placed in the higher layers, and the more general, reusable components are in the lower layers. Since general non-application components are less likely to change than application specific ones, this leads to a stable system as all dependencies are downward in the direction of stability, and so changes tend not to propagate across the system as a whole.

As well as presenting the reference model, this article also discusses and clarifies in concrete terms the *meaning* of one architectural layer being above another. Perhaps surprisingly, our background research has shown that the meaning of the layering metaphor is the subject of some confusion. Specific examples of this are given in the article.

The model presented contains five layers, which are as follows: the interface layer; the application layer; the domain layer; the infrastructure layer; and the platform layer.

Introduction

Architectural layering is a visual metaphor whereby the software that makes up a system is divided into bands (layers) in an architectural diagram. Many such diagrams have been used, and by way of introduction we show two of these.

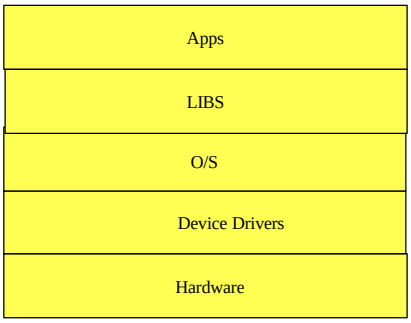


Figure 1 – Layered architecture – Example from Szyperski

Szyperski [Szperski98] presents a view of a strictly layered architecture as can be seen in figure 1. Note that this model has the device drivers below the operating system - a topic we will return to discuss later in this article.

Figure 2 shows a type of ad-hoc architecture diagram [Carlson99] that is not uncommon in modern technical documentation. The example shown describes the architecture of the IBM San Fransisco product.

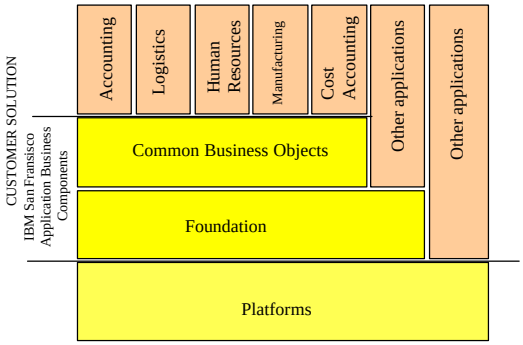


Figure 2 - Typical 'ad-hoc' architectural layering diagram

Some common themes run through these diagrams:

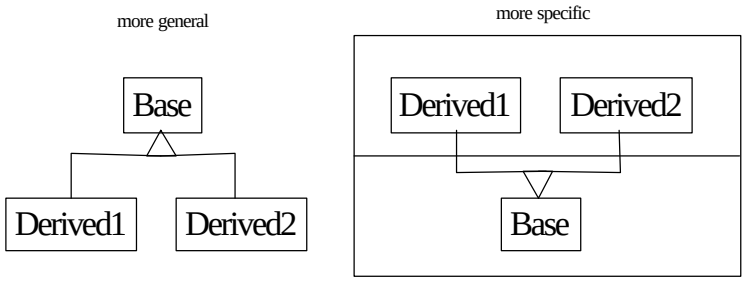
- that it is possible to identify a number of layers in the construction of pieces of software,
- that some layers sit on top of others (although there may be some question as

to what one layer being above another actually means), and

- that one may broadly categorise layers as being either horizontal (applicable across many, if not all business domains), and vertical (applicable across a subset or only one domain);

The reference model is intended to encourage re-use of business specific (not just technology) components.

Turning to UML class diagrams (a younger notation), we notice that common convention usually place subclasses, which are more specialised, below their parents, which are more general purpose. This convention is the exact opposite of the architectural convention *is most specific*, and the cause of a undoubtedly confusing visual metaphor mismatch which we discuss further in our article *The Topsy Turvy World of UML* [Collins-Cope+00].



Typical UML class diagram layout — Architectural diagram showing layers

Figure 3 - Class diagrams and architectural views

This article takes a revised look at application layering, with a particular focus on clarifying the unstated assumptions in such diagrams, and proposes a five layer architectural reference model for component based OO applications that can be used to assist in the design of component based systems.

Layer ordering (highest to lowest is based on component compile time dependencies.

Proposed model

Motivation

The objectives behind the architectural reference model presented in this article are as follows:

- to provide a framework for decision making during the design of components,
- to support and re-enforce the appropriate application of good OO design principles, in particular those concerned with stability and dependency management,
- to provide an architectural framework to encourage re-use,
- to encourage re-use of business specific (not just technology) components,
- to position components as the unifying concept that tie together different architectural views of a system, and
- to provide clarification on the meaning of layering in a component context.

We come back to these motivations in the conclusions section of this article.

Reference Model

We define the architecture of a system as the structural relationship between the individual components that together create an application as a whole. We define a component as an (object-oriented) software development deliverable implementing a well defined interface that is released at the binary (or equivalent) level, which may have a number of well-defined extension points to enable it to be customised.

Examples of components conforming to this definition might include '.o' or '.a' files on a Unix system, '.obj', '.dll' or '.ocx' files on a Windows based system. Components developed within a COM or EJB type environment are, of course, equally within this definition. Note, however, that in most of the following discussion we consider the *design view* of components, which we represent as packages in UML notation.

Figure 4 shows our proposed reference model. Figure 5 shows the same model populated with a number of classes, components and

relationships between them, taken from an example banking application. Two external *actors* [Jacobson+94] interact with the system: a bank clerk (using a debit dialog box), and an external banking system (using an intermediate file format).

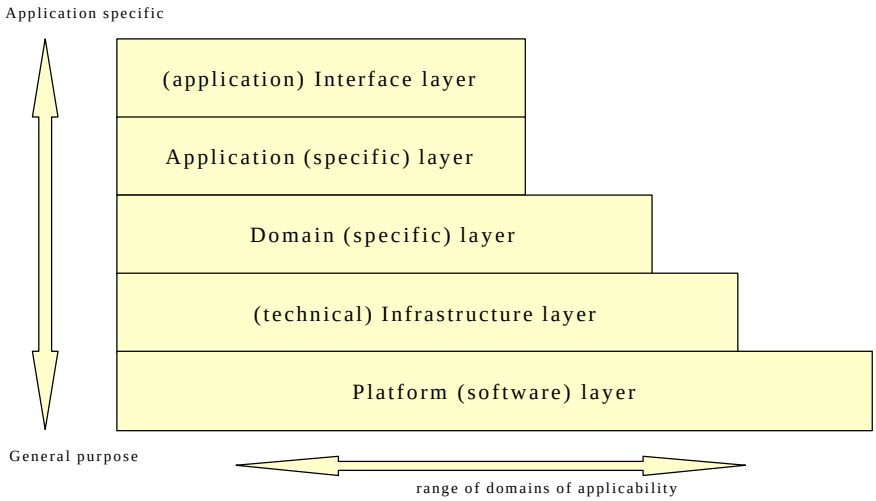


Figure 4 - Layered architecture reference model

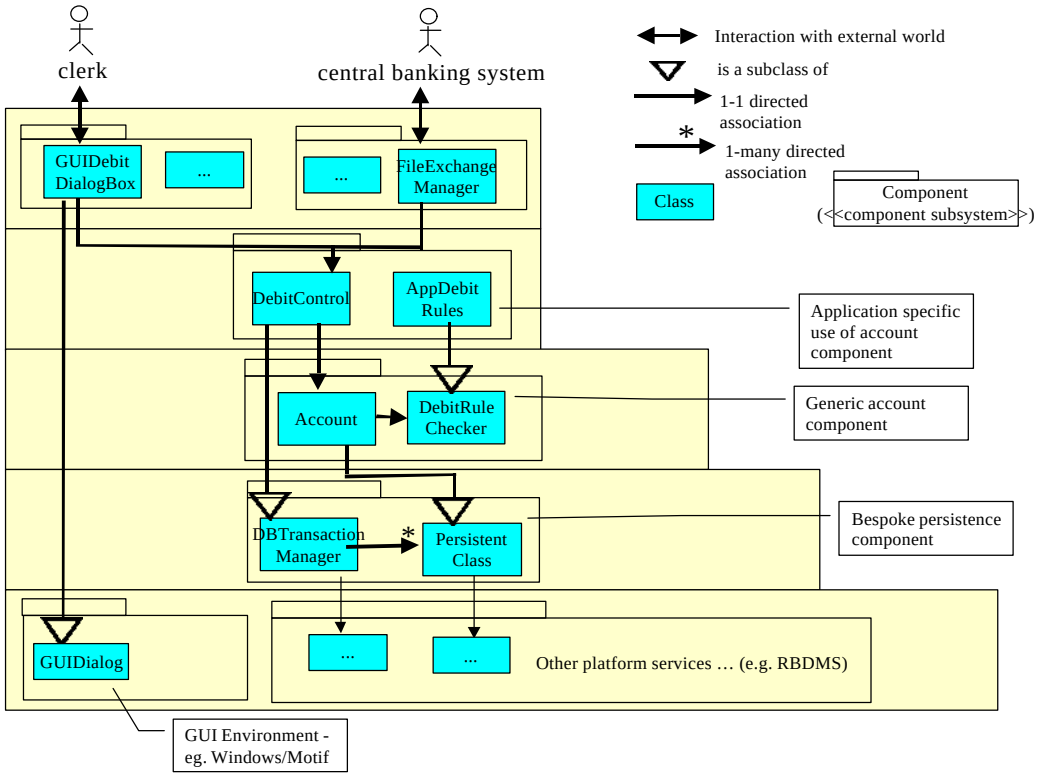


Figure 5 - Layered architecture populated with an example

The layers presented in this model may be summarised as follows:

- Highest (and most specific) in the layering is the application *interface* layer. This layer is responsible for managing the interaction between the 'outside world' and the lower layers within the application. It typically consists of components providing GUI interface functionality - managing the interface to human users, and/or components providing external system interfaces - managing the interface to external systems. This layer often contains what Jacobson et al. call *boundary* classes [Jacobson+94].

- In the example application shown in figure 6, we can see two classes packaged within this layer. The *GUIDebitDialogBox* class implements an application specific dialog (to enter a debit). The *FileExchangeManager* class reads an external file format. Both classes use the application specific *DebitControl* class to process the information they receive from the outside world.
- Below this is the *application* specific layer. This layer is comprised of objects and components that encapsulate the major business processes and associated business rules automated within an application. Typically it will contain many objects akin to Jacobson's (use case) 'control' objects [Jacobson+94]; and often also acts as the 'knowledge' layer in Fowler's operational/knowledge split [Fowler98] (another description of this is separating policy from mechanism.) It may also contain specialised subclasses implementing interfaces left 'open' (as in Open Closed Principle [Meyer97] [Martin96]) by the more general purpose components in the layer below, and typically does not contain persistent business classes. Most importantly, this layer contains the "glue" to tie together components within the domain layer below.

> Most re-usable of all, is the platform software layer. This is comprised of standard or common-place pieces of software that are brought in to underpin the application.

- In the example application shown in figure 6, we can see two classes packaged within an application specific debit component. The *DebitControl* class takes over application control when asked to do so by one of the higher level interface classes. It then drives the domain level *account* class to implement its functionality (which may involve several method calls on *account*). Note that the *DebitControl* class is derived from a database transaction management class defined in the bespoke persistence component in the infrastructure layer. The other class - *App. debit rules* - implements the debit-rule checker interface (derived from the lower level *account* component) to customise the debit checking rules as required by this application.
- Next is the business *domain* specific layer. This layer is comprised of components which encapsulate the interface to one or more business classes, which are specific to the domain (area of business) of the application, and are generally used from multiple places within the application. They might also be used by a family of related applications - a software product line. This layer typically contains the 'entity' classes discussed by Jacobson et al in [Jacobson+94].
- The example application shown in figure 6 shows an *account* component in this layer. The *account* class is driven by higher level components to undertake account related activities such as debiting and crediting of monies. As part of this, it uses a *DebitRuleChecker* interface (abstract class) to enable individual applications to customise the particular debit checking rules that may be applied (e.g. can go overdrawn, can't go overdrawn, etc.) This is an example of the open/closed principle [Meyer97][Martin96] being used to implement an operational/knowledge split [Fowler98]. Note also that, being persistent, the *account* class is derived from the *persistence* class in the infrastructure layer.
- Then comes the technical *infrastructure* layer. This layer is made up of *bespoke* components that are potentially re-usable across any domain, providing general purpose 'technical' services such as persistence infrastructure, general programming infrastructure (e.g. lists, collections).
- The example application shown in figure 6 shows a general purpose persistence component being present in this layer. In this component, a *DBTransactionManager* class keeps tabs on a number of *PersistentClasses*, which provide the hook by which higher level domain classes many be made persistent.
- Finally, most of all, is the *platform* software layer. This is comprised of standard or common-place pieces of software that are *brought in* to underpin the application (e.g. operating systems, distribution infrastructure (CORBA\ COM), etc.) The example application shown in figure 6 shows a relational database and a GUI class library being used to build the application.

Associated rules

Some simple rules are associated with this model:

- there should be a clear and simple mapping between component structure and source code structure (the simplest

being a 1-1 mapping), and between the component structure any other analysis and design artefacts produced during the development process (e.g. the design view of a component, as shown in a package diagram, or the use case view of a component, as shown using packages of use cases);

- the level of a component is the highest level of any of its constituent classes;
- components should not (and by the above definition, cannot) cross layers;
- the compile time dependencies between components within any particular layer should be to components in either the same or a lower layer;
- the application and domain layers should be technology free in the interface components within them present to the outside world;

Layering Semantics

Most layering diagrams omit to discuss the meaning of one layer appearing on top of another, or any description of the axis of the diagram. Earlier reviews of this article, and the example shown in figure 1 have lead us to believe some further discussion of these aspects of the layering model presented here is desirable:

- *Vertical axis semantics*. The vertical axis of figure 5 indicates the *specificity* (how specific it is to a particular application/environment) of a component in the application. The higher it appears in the layering of the reference model, the more specific it is. The lower it appears, the more general purpose it is. This has lead us to coin the phrase *the centre of gravity of the application* - essentially a way of classifying the overall architectural feel of a system as being either 'high' (very application specific, difficult to extend without substantial modification to existing components), or 'low' (good layering applied, likely to have hooks for extension without any modification to existing components).
- *Layer ordering (highest to lowest) is based on component compile time dependencies*. In figure 1, Szyperski shows a layering model in which the device driver layer is shown below the O/S layer. Whilst this seems appropriate at first glance, a deeper examination reveals the ordering in Szyperski's example is not based on the same criteria as the layering presented in our model. In our model, layer ordering is based on the compile time dependencies between the components that reside within the layer. In the terms presented in this article, the device driver interface of an operating system is an *extension point* to enable customisation of the operating system "component" to a piece of particular hardware. The operating system is *more* generic (general purpose) than the device drivers it uses (which are tied to particular hardware). The device drivers are also dependent *upon* the operating system for their definition – their interface must conform to the calling interface used by the operating system (they will use the function prototypes defined in an operating system header file) – not the other way round. For this reason, we would present the middle three layers of figure 1 in the following manner (with additional detail to show interface definitions and instantiation of interface):

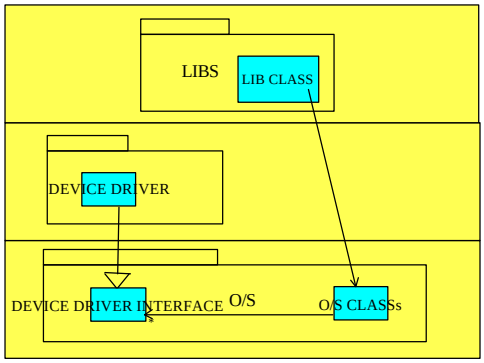


Figure 6 – Szyperski's example using our layering rules

Summarising, the layering semantics presented here tie together the concept of the specificity of a component (how much detail is filled in, how specific it is) with the notion of compile time dependencies. The higher a component in the model, the more specific it is likely to be, and the more dependent it is likely to be on other components, and *vice versa*.

Further notes

A number of additional points are worthy of brief discussion:

- *A component oriented approach*. We have defined our view of architecture as one being based on the structural inter-relationships between the binary components that are used to make up a system. We adopt this focus because at the end of the software development process we would like to have a number of well-defined and well-structured, loosely coupled, internally cohesive binary components that we may, without

modification, use again in extending the current application (or possibly another application).

- *GUI components*. All GUI components do *not* reside at the interface level. The interface level may contain application specific refinements of general purpose GUI components (e.g. an application specific dialog box), however the *general purpose* elements themselves (e.g. the generic dialog box from which the application specific one is derived) live at the platform level. The same is true for any general purpose GUI component without application specific customisation (e.g. a graph drawing widget).

All GUI components do not reside at the interface level.

- *Substitutability*. Many discussions of architectural layering focus on being able to replace one *whole layer* with another – they are effectively treating the whole layer as a single component. This is *not* the purpose of the model presented here, which is intended as a guide to determining the contents of a particular component by deciding upon within which layer it is appropriate it reside. Substitution would take place at the individual component level, not on a per layer basis.
- *Three-tier architecture*. It is interesting to see how the model presented here maps to the classical view of a three-tier architecture (presentation, business logic, database). The model presented here can be viewed at a conceptual level as being independent of detailed deployment issues. However, it can also be used for applications deployed across multiple machines/processes – for example as in the classic three-tier model. In this case:
  - the *presentation tier* would contain the interface layer, (possibly) the application layer (in a thick client configuration), and some components of the platform layer (e.g. CORBA client components, generic GUI components).
  - the *business logic tier* would (possibly) contain the application layer (in a thin client configuration), the domain layer, the infrastructure layer, and some components of the platform layer (e.g. distributed ODBC client components , CORBA server components), and
  - the *database* tier would contain the remainder of the platform components (e.g. the RDBMS, ODBC server components, etc.);
  - the net result being that the higher layers are deployed in one particular machines/process, and that the lower two layers are often present on multiple processes/machines.

How the layering helps us understand design

To see how layers and particularly our visual metaphor help us, let's examine the Adapter pattern from the Gang of Four's book [Gamma+95].

Figure 7 shows an adapter being used to allow two components with incompatible interfaces to be used together (a common problem in component design). The billing adapter implements the Account component's outward billing interface and passes on any messages to the credit card billing component's inward billing interface, with possibly modified parameters. Since both of the components are reusable and not specific to this application they belong in the domain layer. The adapter, on the other hand, is very specific to this particular configuration and so it is not in general reusable and so belongs in the application layer instead – it is acting as application-level "glue" for the other components. Note here that the two component dependencies point downwards – one being caused by an inheritance relationship, the other by a directed association.

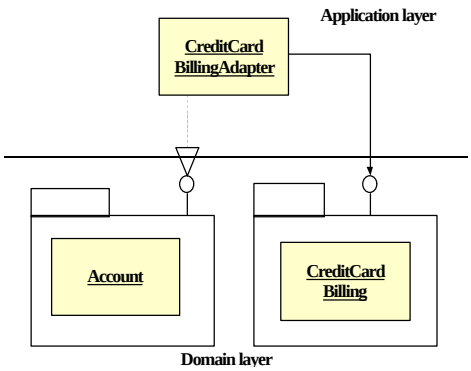


Figure 7 - Getting an account paid by credit card

For a second example, we show in figure 8 one of the refactoring patterns from Martin Fowler's book [Fowler+99]: Separate Domain from Presentation.

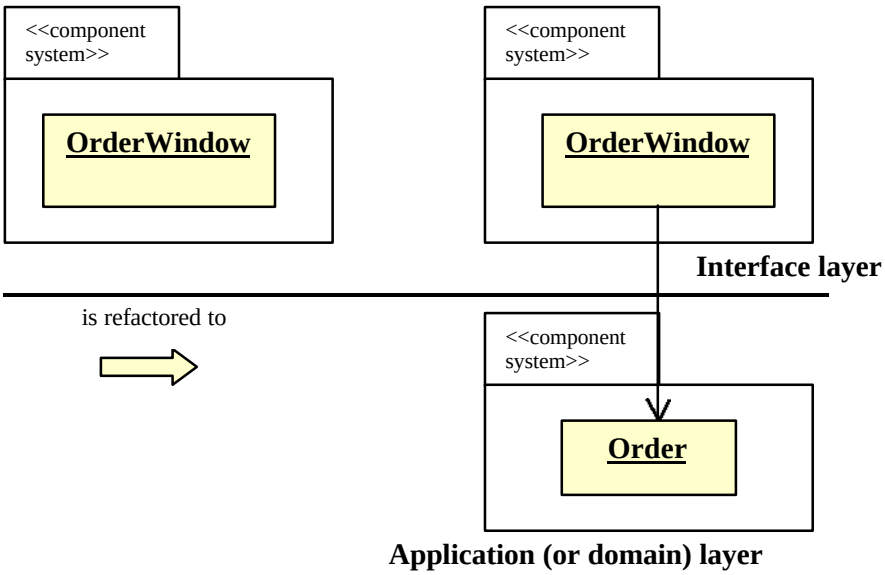


Figure 8 - Separate Domain from Presentation

Here we see a GUI dialog class containing business logic being split into two. Our architectural overlays add context to this, showing that in doing so what was previously an interface layer component has now been split into two components: one still in the interface layer, and one in the application specific (or possibly domain) layer. The refactoring visibly lowers the centre of gravity of the application. Two more benefits are that we have separated the usage of the Order component from its implementation (in other words we have separated policy from mechanism), and that we have separated the technology-free Order component from the inherently technology-based OrderWindow, thereby giving us more portable code and allowing us more freedom in deployment (for example in a three-tier system).

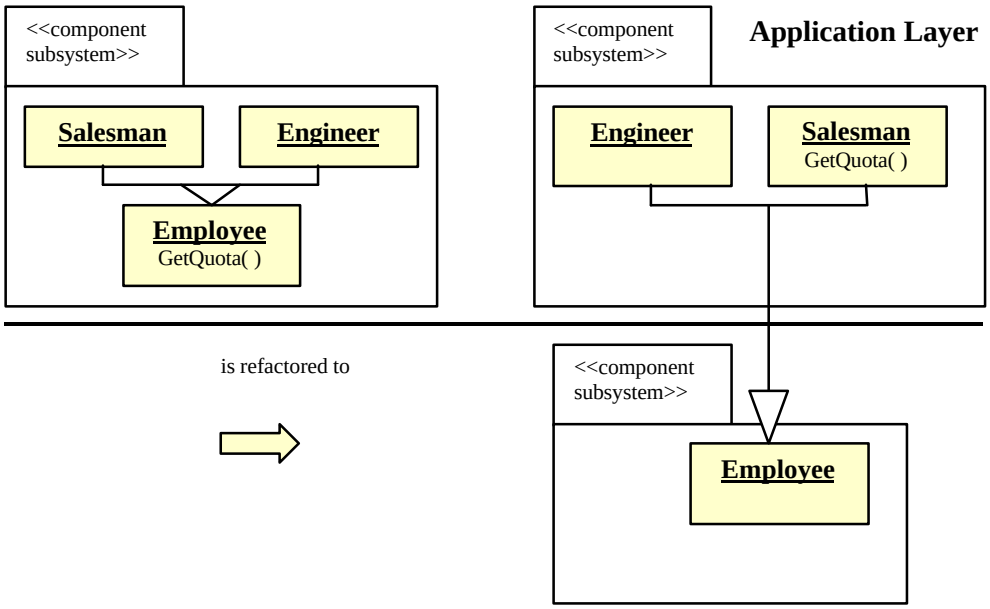


Figure 9 - Push Down Method (reworked by us as Pull Up Method!)

In figure 9, Fowler shows us an inheritance hierarchy with an inappropriately placed method, which is effectively *polluting* the component which contains it by forcing it up to an inappropriate level. The refactored version shows the component being split into two (over two levels), the method having being moved out the Employee class and into the Salesman class. This enables the Employee component to reside at a more general level in the hierarchy, and again visibly lowers the centre of gravity of the application.

Broader issues

- No name, no discussion
  Nowadays we talk about composites and flyweights [Gamma+95], but this would not have been possible ten years ago. Having a standard layering vocabulary would be of clear benefit in enabling developers to discuss the level at which components might lie.

- No architecture, no re-use
  An architectural framework is key to achieving re-use [Jacobson+97]. Re-use requires a clear understanding of what is specific and what is general in designing a set of related components. The reference model assists in understanding these separations, whilst also adding clarity to the layering semantics.

- Architecture adds context
  An architectural view adds additional context to many design patterns. Take the observer pattern [Gamma+95]. If packaged together in a single component, the base classes Subject and Observer can be seen as providing a low level (infrastructural) flexible component from which higher level (domain, application or interface) concrete observation mechanisms can be built. The derived ConcreteSubject and ConcreteObserver can be seen as application, domain or interface specific extensions to a general purpose mechanism.

- Architecture aids (good) design
  The layering presented also supports a number of OO design principles: the open closed principle [Meyer97][Martin96] – the idea being that lower layer components are closed against modification, and higher layer components extend the open aspects of them; the stable dependencies principle [Martin97] - the layers are organised based on expected stability of their contents – the lower layer components being more stable; and the acyclic dependency principle[Martin97] - the depend downwards rule supports this principle.

- *Invariants*
  One important part of the design of component based systems is the identification and allocation of responsibility for maintaining invariants *across* (possibly bought in) components. It may be necessary for two components in the domain layer to maintain an invariant relationship - e.g. customer address must be maintained in both components. The responsibility for maintaining invaritants across components resides in a layer *above* that in which the components reside. So in the case of our customer address invarient, it would be the responsibility of a component within the application layer to ensure it was not violated (perhaps using a change event generated when the customer address was changed in either domain layer component).

One important part of the design of component based systems is the identification and allocation of responsibility for maintaining invariants across components.

A brief philosophical aside

Many classification systems are blurred around the edges, and our layer classification is no exception. In his excellent book *Darwin's Dangerous Idea* [Dennet96] Dennet describes how examining the characteristics of a particular species of gull, starting in Britain and moving west to east around the globe, yields a set of gradually evolving changes until, as the loop closes and the examiner returns to Britain, a *different* species of gull is finally found next to the original! Species clearly have blurred edges, so we're in good company!

Compromises

The model presented here is not perfect, but a compromise between simplicity and meeting the stated set of objectives. Some potential shortcomings of the model are as follows:

- It would have been superficially pleasing to have a rule that said: *you can only depend on the layer below you*. However, upon deeper consideration we believe that the reason for this is an emphasis in previous discussions of layered architectures on complete substitutability of layers. As discussed in earlier section (*substitutability*) this is not the motivation behind the model presented here.
- There are clearly times when there will be sub-layerings within the layers presented, and we could have made these explicit. However, we feel the price would have been too high in terms of additional complexity. Instead, we prefer to allow the option of discussing the 'lower application' layer to resolve such issues.
- We have chosen to separate the infrastructure and platform layers based on a buy versus build criteria. Architectural purists may object to this - why should the

layer in which a particular component resides be dependent on whether you buy or build it? Our motivation is simple: we wanted to put the focus clearly on the *aspects of the application being developed*. For similar reasons we have been unconcerned with sub-layerings within the platform layer.

Conclusions

Summarising, in this article we have proposed a simple five layered reference model and a number of associated rules to assist the software designer. We have noted that, by convention, *UML class diagrams are upside down*, at least when considered in parallel with architectural layering conventions, and that this is a block to visualising one aspect of what happens during refactoring. We have shown that once this is addressed, UML and the architectural model complement and re-enforce each other.

We have identified examples from Gamma et al's Design Patterns book, and Fowler's refactoring book that show the reference model adds context to well known design (and refactoring) paradigms.

We have discussed how the model supports good OO design principles, in particular those concerned with ensuring stable dependency management, and have emphasised and clarified the rules on which our layering model is based (specificity/generality and compile time dependency).

Coming back to the objectives detailed in earlier in this article, we believe the architectural reference model presented here:

- *provides a framework for decision making during component design* by providing a number of layers within which the developer can position their components,
- *supports and re-enforces the appropriate application of good OO design principles*, by encouraging components to be extended (customised) by other components in higher layers, and by imposing a downwards only dependency rule,
- *encourages re-use* by providing a layering system and associated set of rules that puts the focus of design on the specificity/generality of components, encouraging components contents to be seperated on the basis of the layering provided,
- *provides clarification on the meaning of layering in a component context*, by putting the emphasis on compile time dependency management,
- *encourages re-use of business specific (not just technology) components* by presenting two layers within which business components may reside. This encourages more generic functionality to be in the domain layer, and application specific customisation/glue type function-ity to be in the application layer, and
- *positions components as the unifying concept that tie together different architectural views of a system*, by stating that the package structure of the system (and associated specification, design or use case views) should be based on the target component structure.

References and credits

[Szperski98] Clemens Szyperski, Component Software: Beyond Object-Oriented Programming, January, Addison Wesley Longman, 1998.

[Carlson99] Brent Carlson, Design Patterns for Business Applications, the IBM SanFransisco Approach, ObjectiveView Issue 3, available at www.ratio.co.uk/objectiveview.html, 1999.

[Collins-Cope+00] The Topsy Turvy World of UML, Hubert Matthews and Mark Colllins-Cope, ObjectiveView Issue 4, available at www.ratio.co.uk/objectiveview.html, 2000.

[Jacobson+94] Ivar Jacobson, Magnus Christerson, Partrik Jonsson, Gunnar Övergaard, Object Oriented Software Engineering - A Use Case Driven Approach, Addison-Wesley, 1994.

[Gamma+95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[Fowler98] Martin Fowler, Analysis Patterns - Reusable Object Models, Addison Wesley, 1998.

[Jacobson+97] Ivar Jacobson, Martin Griss, Patrik Johsson, Software Reuse - Architecture, Process and Organization for Business Success, Addison Wesley Longman, 1997.

[Meyer97] Bertrand Meyer, Object Oriented Software Engineering (second edition) - Prentice Hall Professional Technical Reference, Published 1997.

[Martin96] Robert C. Martin, The Open Closed Principle, C++ Report, Jan 1996.

[Fowler+99] Martin Fowler with contributions from Kent Beck, John Brant, William Opdyke, and Don Roberts, Refactoring - Improving the Design of Existing Code, Addison Wesley, 1999.

[Martin97] Robert C. Martin, Stability, C++ Report, Feb 1997.

[Gamma+95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[Dennet96] Daniel C. Dennet, Darwin's Dangerous Idea: Evolution and the Meanings of Life, Touchstone Books, 1996.

Particular thanks are due to Andy Vautier, Nigel Barnes, Andris Nestors and Keith Haviland of accenture, upon whose 1 million line+ C++ project many of the underlying concepts presented in this paper were formulated. Further detailed technical discussion this project can be found at www.ratio.co.uk/techlibrary.html.

Component Distribution Patterns - A mini-language for distributed component design

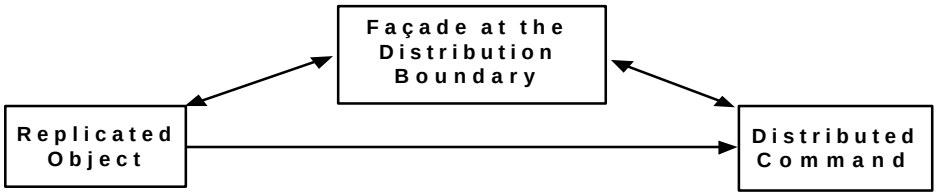
Philip Eskelin (with **Kyle Brown** & **Nat Pryce**) discuss three related patterns for distributed object and component based application development…

INTRODUCTION

THIS language is an exploration of the problem of building distributed systems using component technology. For the purpose of this language, a component is a reusable software entity with a well-defined interface that fulfils a specific purpose. Component frameworks like Enterprise JavaBeans and COM+ provide services (e.g., distribution, transaction support, or persistence support) that can be used by components.

This language follows the Alexander form [Alexander], which breaks patterns into the following elements: a *name* providing a descriptive label; a *picture* providing a visualization of the theme; a *context* description tying it to the patterns that link to it from their resulting contexts; a *problem* statement capturing the essence of the problem and description elaborating upon it; a *solution* statement succinctly communicating the essence of the solution with a description providing analysis, illustrations, and known uses; and a *resulting context* description tying it to the patterns flowing from it.

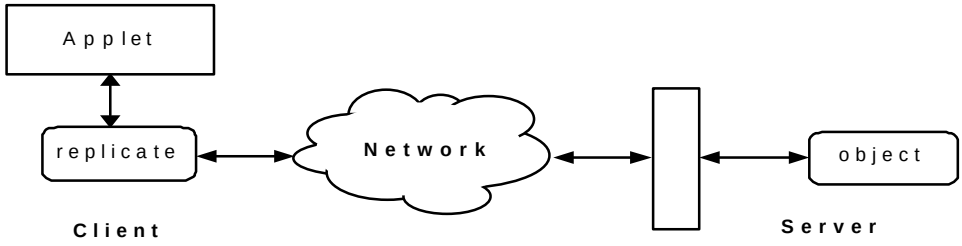
The language is a subset of the *Component Design Patterns* language, but can stand on its own as a micro-architecture for building software containing distributed components. Three patterns are included: FAÇADE AT THE DISTRIBUTION BOUNDARY, REPLICATED OBJECT, and DISTRIBUTED COMMAND.



Arrows denote pattern relationships. For example, REPLICATED OBJECT mentions DISTRIBUTED COMMAND in its resulting context description, and DISTRIBUTED COMMAND mentions REPLICATED OBJECT in its context description. Since this language is part of a larger one, other patterns are mentioned that are not covered in this language.

Readers of this language should be familiar with design patterns [Gamma] and the J2EE [Sun] standard and architectures of distributed component frameworks from Java application server vendors such as IBM, BEA, and ATG

REPLICATED OBJECT



When building distributed applications that require interactions between remote components, efficiency and code management concerns often dictate that alternatives to using a PROXY for remote interactions is required. Sometimes an event-driven approach is desired to provide better efficiency for real-time notification of changes to remote objects. This pattern documents a technique that can be used to efficiently access distributed objects.

A distributed application that uses pass-by-reference for method invocations via proxy objects can lead to an excessive number of network calls for complex data manipulation in remote objects. Polling methods to provide updates to the state of a remote object can limit scalability and lead to inconsistencies.

PROXY is such a useful pattern that many programmers begin thinking it is the complete solution to their distribution problems. However, proxies have the unfortunate side effect that every call to a proxy crosses the network. In many situations, this is not only too costly, but it is unnecessary when taking into account the desired behavior of the object. For instance, imagine a simple stock-trading application.

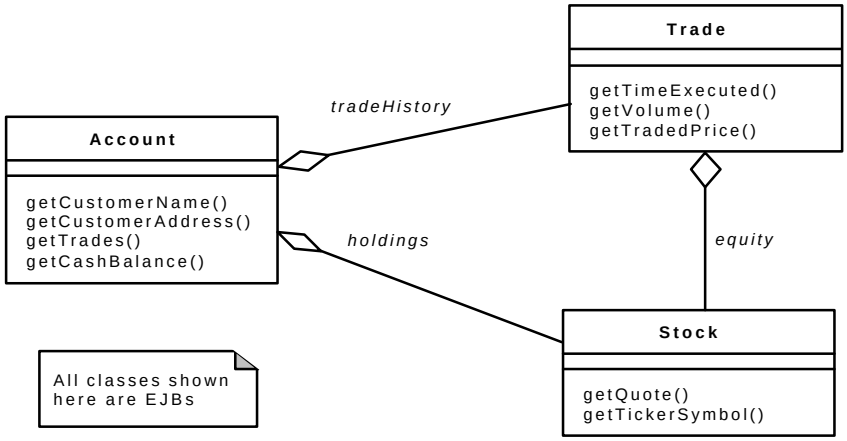


Figure 1: Original Design

In Figure 1, a session EJB is used to represent a customer **`Account`**. Other EJBs contain information about holdings and the trades that are made. To display all of this information on a web page, every single piece of information that is needed—the customer name, address, account number, stock ticker symbols and the amounts of trades—must be obtained through separate network calls. The following interaction diagram illustrates this:

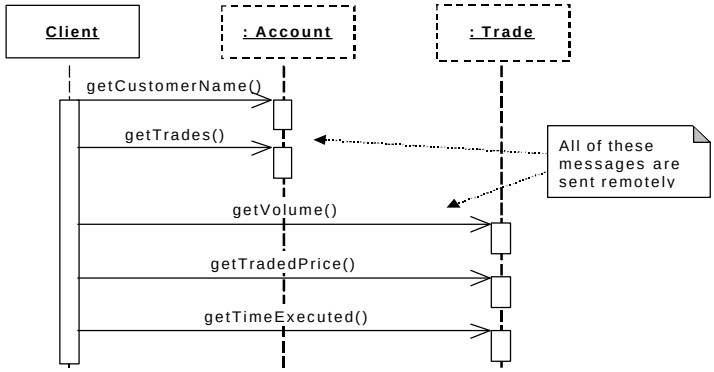


Figure 2: Remote Interactions

Figure 2 shows that excessive network calls occur, especially in the case where there are specific objects (like a **`Trade`**, or a **`Stock`**) that have been completely fetched from a back-end persistent store and do not vary from one method invocation made to our **`Account`** EJB to another.

The number of network calls must be optimized so the client can fetch attributes and perform other computation-intensive tasks locally without sacrificing bandwidth or consistency. By locally replicating the remote object via a façade, in-process method invocations can occur as many times as is required, and the object can be designed to invoke distributed commands when the remote object needs to be updated.

Therefore:

Use a pass-by-value approach for business objects requiring remote interactions in your distributed application. Serialize a snapshot of the object on the server so that a

replicate can be reconstituted on the client end, and propagate updates to the object as event notifications. Programmers choose which objects in their system will need to be manipulated on both ends of a client-server interaction, and replicate them accordingly.

In the above example, the **Trade** and **Stock** objects would be replicates that are serialized and sent together from the server (i.e., from the **Account** EJB) to the client application that needs to display them. In EJBs, this can be done by making the **Trade** and **Stock** serializable beans, rather than making them full EJBs. The following illustrates **Trade** and **Stock** as JavaBeans:

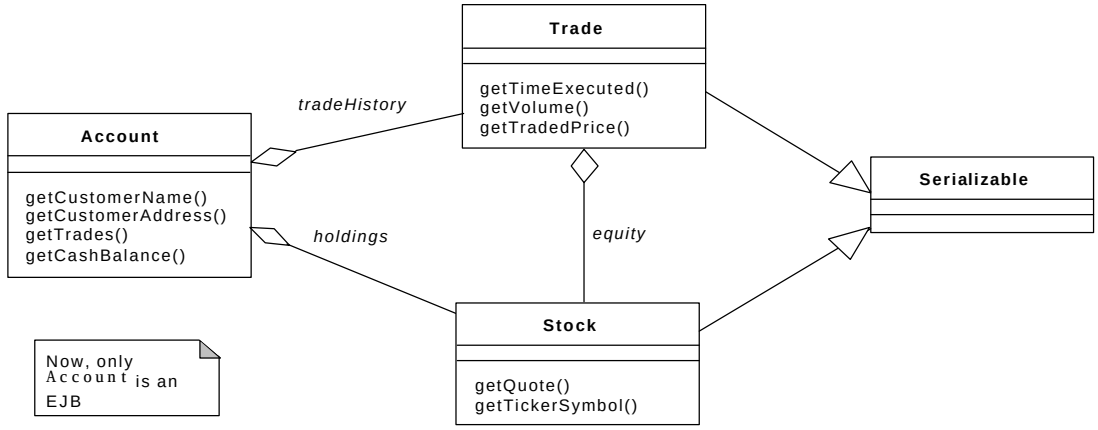


Figure 3: Serializable Objects

This change results in change in the distribution boundary. Now only some of the messages from the client to the other objects are remote calls. Calls to the **`Trade`** are local, and only calls to the **`Account`** component are remote. The following interaction diagram illustrates this:

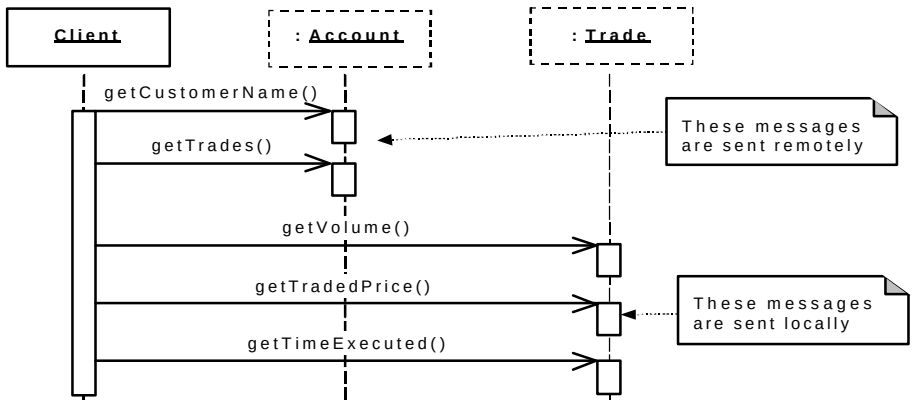


Figure 4: Local and Remote Mixed

Replication is particularly useful when a complex object net needs to be traversed. If each call to obtain a new "node" in an object net required a network call, the overhead from those calls would be too high to be practical in most applications. For example, remote traversal of the DOM tree of an XML document would be impractical. Even if the document were parsed remotely before traversal began, the navigation and retrieval of all elements and attributes alone would require hundreds, if not thousands, of network calls.

In addition, by replicating objects, you are not forced to deal with the cumbersome task of implementing all features for every remote object in the component framework. In our example, since **`Account`** is the only component that is an EJB, and both **`Trade`** and **`Stock`** are serializable JavaBeans, only the EJB needs to worry about the distributed, thread-safe, and transactional features offered by EJB containers. The EJB works with each of the beans internally to maintain consistency and thread safety, and the solution still benefits from drastic reduction in network overhead.

However, one consequence that arises is that replicated objects are not full-fledge components and do not have the same level of support in the environment as components do. An object factory can be created for each type of replicated object to fill this gap. They would be responsible for creation, update, and instance management of these objects, and can be implemented to participate in managed transactions and integrate with a façade at the distribution boundary. The following illustrates the design of a factory for the example above:

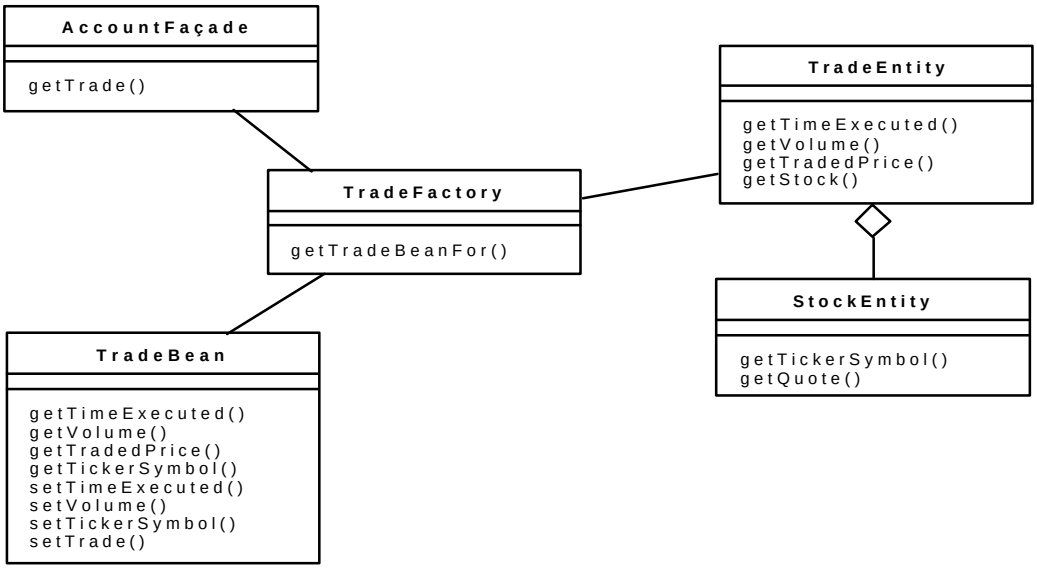


Figure 5: Object Factory

A **`TradeFactory`** is created that responds to requests from the session EJB to create a **`TradeBean`** for a particular key value. It would accomplish this by using the homes (i.e., component factory) of the appropriate entity EJBs to locate the EJBs that contained the needed information, and then copy that information into a new **`TradeEntity`**, which it would return to the session EJB. Likewise, when an update occurred, the session EJB would instruct the object factory to carry out the update. It would locate the appropriate entity EJBs and update them within a single transaction regulated by the session EJB.

In essence, an object factory designed to manage replicated objects could be thought of as a façade for the relational database access layer. In fact, a common way of using popular object-to-relational mapping tools is to use their built-in object factories to create objects that can be obtained by clients of a session EJB acting as a façade at the distribution boundary.

This pattern first arose in the GemStone object-oriented database for Smalltalk, which provided both OODB and application server functionality. When working with GemStone programmers had the option of choosing to execute methods in either the server process space, or the client process space. This meant that at runtime an object could be declared to either be a replicate, or a proxy.

The set of objects that can be passed by value may be disjoint from the set that can be passed by reference (the EJB) or may overlap that set (through Java RMI). For example, the semantics of Java RMI are such that methods of remote interfaces may return any primitive or legal Java class, so long as that class implements the interface **`Serializable`**. In this way, snapshots of objects are serialized in the server, and then deserialized on the client end and returned to the object that initiated the call to the proxy. So any serializable object may be a replicate in RMI (and by extension, EJB).

However, it is not easy to implement this pattern in all distribution frameworks. For instance, in CORBA 2.2 [98-07-01], there was no way to define an object that could be passed by value. CORBA 2.2 only provides for structs, analogous to C structures, that are data-only and do not allow for the definition of behavior. So if a programmer wishes to ask a distributed CORBA component for some information, and then manipulate that information on the client (receiving) end, he must first ask for a struct from the local proxy

to the CORBA component, and then copy the information from that struct into another object that can manipulate the information. This copying must be hand-coded, and is prone to error and also prone to break when the definition of the struct changes in IDL.

CORBA supports passing objects by value in its current specification (a joint submission [98-01-18] on Objects by Value was approved in 1998). It does so by proposing a new IDL keyword (value) that allows for the creation of objects that have state and methods, but are not descended from `CORBA::Object` and thus cannot be represented as an IOR.

Programmers accessing remote components through proxies eventually jam networks and bring down servers if the designers of their distributed applications lacked experience…

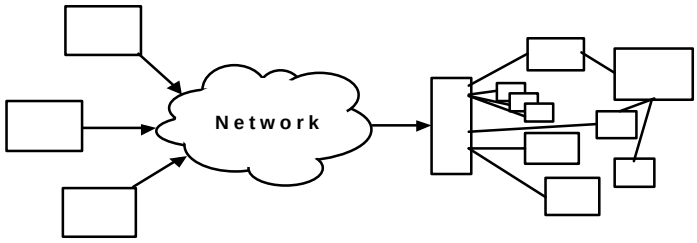
When you begin to use replicated objects, there are a number of consequences that you need to consider. The first is the issue of synchronization of replicates between the client and the server. One solution to this is to use dirty bits to record if the information that is stored in the replicate has been changed by the client.

This does not solve the problem of reconciling differences between client and server when the information on the server changes, however. In this case, callbacks may be employed to update the replicate whenever the server changes. However, this requires the server to be aware of what is current on each client – this can lead to excess memory requirements on the server, and network overhead in the number of calls needed to update each client.

To address the drawbacks of addressing this consequence, a variation could use a component bus implemented in the façade that allows the replicate to receive event notifications each time the remote object has been updated. Each replicate could subscribe to a topic dedicated to the remote object via the façade's component bus. Initially, the replicate receives a serialized snapshot of the remote object, and then receives only delta notifications that describe only what has changed.

Replicated objects are often returned by the methods of a FAÇADE AT THE DISTRIBUTION BOUNDARY. If replicated objects are sent from client to server and then later returned again to the server, the overhead of sending multiple copies of unchanging data can result in performance problems. This naturally leads to use of a DISTRIBUTED COMMAND to encapsulate changes between replicated objects. Also, if replicates need to stay consistent with the state of the remote object, then use of a COMPONENT BUS implemented in the façade could provide a facility for clients to receive initial snapshots and subsequent event notifications sent from the remote object to its replicates.

*FAÇADE AT THE DISTRIBUTION BOUNDARY**



Programmers of distributed applications that use PROXIES or REPLICATED OBJECTS sometimes need to strike a balance in number of objects that are made available to remote components located across the network. It might be necessary to reduce the number of remote invocations, or provide a controlled entry point to business objects located in an application tier. This pattern helps the programmer determine the best level of granularity for creating such an abstraction.

Programmers accessing remote components through proxies eventually jam networks and bring down servers if the designers of their distributed applications lacked experience when designing remote interactions. The number of proxies can become excessive, and many objects within

the application must become relatively static and unchanging. Changes to clients and/or servers can become problematic.

When mainstream programmers moved from building monolithic applications to building ones that were distributed across a network and connected by middleware, many developers attempted to apply design practices that were no longer suitable for distributed environments. Let us consider the following scenario.

A manufacturing system is being designed for a car factory. The first object designed is a **`Vehicle`**. In subsequent design discussions, a **`BodyStyle`** is designed to encapsulate similar properties between **`Vehicles`**. As the team thinks more about what a **`Vehicle`** is composed of, they design more objects. A **`Vehicle`** is refined to contain a list of **`Parts`**, which in turn reference **`Specifications`**. As designers focus on the manufacturing process, multiple **`WorkOrders`** containing **`BuildInstructions`** facilitate the construction of a **`Vehicle`** with a particular **`BodyStyle`**.

> Bind tightly coupled objects together inside a single process space. Define a façades between these new groups that act as gate-keepers hiding the complexity (and sheer number) of the objects they wrap…

The design – which elicits principles like polymorphism and encapsulation, and uses the OBSERVER and COMPOSITE design patterns – works well in a system running in a single address space. It implements ad hoc and canned reporting on the execution status of build instructions, and allows its users to determine which vehicles are currently in production and how many of what body styles are being built. Managers have the capability to add and update all of the above objects.

Requirements originally dictated that the system would only need to run in a single address space. But when upper management saw how it benefited one manufacturing plant, they wanted to deploy it in all plants worldwide. Also, they wanted the ability to run certain reports and analyze cycle times from corporate headquarters in Detroit.

Managers decided it would be relatively painless to distribute the system using a popular CORBA middleware product. Client machines would run the GUI from the standard corporate web browser, while the centralized server would handle most of the processing. Management also determined that it was required to spit server processing into assembly (which must never go down) and reporting (which can go down occasionally).

One of the architects, who had prior experience with DCE RPCs, said "No problem. CORBA gives us proxies, so we'll just take our existing objects and write IDL interfaces for them." Once the IDL compiler was run against the IDL file, it would produce proxy classes providing class methods that would be virtually identical to the single system version.

But they soon discover that they need to write CORBA interfaces for nearly every class in the system. Not just the **`Vehicle`** and **`BodyStyle`**, but for all the classes they associate with such as **`PaintColor`** and **`Accessory`**. The IDL becomes large, and the proxy code generated from the IDL has a very large footprint. During a test run, the web browser took several minutes to load all of the client-side objects and proxies that accessed the server.

Also, they start noticing that network traffic had increased substantially, and that performance and reliability are unacceptable. During analysis, they found that objects in one process were sending hundreds of messages tightly coupled objects that existed in another process space. In addition, every change to the system required a recompilation of the IDL file and of all classes that use its proxies. Every update meant users needed to wait for minutes as the browser re-cached the client-side objects and proxies. Testing became problematic, since every test needed to be done over the network.

In scenarios like the one above, the first step to solving performance, reliability, and scalability problems is to limit the number of remote interfaces. Best practices in building distributed systems are not just trivial extensions of those used in monolithic systems. Architects and designers must define a distributed architecture that supports the level of scalability, flexibility, performance, and reliability required to allow the business to reap the expected benefits of a distributed application. Therefore:

Bind tightly coupled objects together inside a single process space. Define a façades between these new groups that act as gate-

keepers hiding the complexity (and sheer number) of the objects they wrap, and providing a central abstraction that indicates which messages are crossing the network.

A façade located at the distribution boundary designs fewer and more centralized remote interfaces, and a clear line between foreign and domestic interactions. The following benefits and liabilities apply to using a façade at the distribution boundary:

- *Less flexibility*. When a component acts as a façade containing many smaller components, one tradeoff can be that adding new components means you must update the interface and implementation of the façade component and test to ensure that all existing components still work properly.
- *Easier to manage change*. A benefit of the façade is that you are in effect wrapping what would be separate smaller physical components with one larger physical one, then allowing logical access to each component inside it. You present a "view" into these components with the façade. Each of these components can share a common infrastructure and operate off of the same framework. They can reuse standard libraries, and reduce version discrepancy headaches that sometimes emerge in less-controlled development, test, or production environments.

The IP version 4.0 (IPv4) addressing architecture is a related example. To address capacity issues with the antiquated addressing scheme being used by the Internet in the early 1980s, the Internet Engineering Task Force (IETF) added addressing standards that allowed domains to create logical networks with more hosts than the physical network would allow.

The problem fundamentally lies in the fact that Internet routers must be able to route data grams between every possible host. Originally, every machine on the Internet had a unique IP address, so there was a maximum of 2^{32} (or 4,294,967,296) possible hosts, which seemed extremely excessive. The problem was that routers would eventually need to manage very large lookup tables.

Then, the IETF broke addressing down into classes of IP addresses. This led to the concept of a network address and a host address.

In a large GemStone project, the problem of wanting to reduce the distribution cross-section to minimize network traffic and swapping between the local and distributed object spaces (We were using a pass-by-value approach, e.g. replicated objects for most of our objects) existed. A façade was applied while refactoring it to solve it.

This pattern was applied in an options trading system that where a set of "services" were developed that each did one key thing, like "trade options" or "handle quotes". Each service encapsulated many of domain objects within a relatively simple façade API that it published as an interface for other services.

The FAÇADE pattern was previously documented in *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma],

Design Patterns Smalltalk Companion [Alpert] in the section on the FAÇADE pattern (but not in pattern form), and in *Analysis Patterns* [Fowler] in Chapter 12, "Layered Architecture for Information Systems."

A façade at the distribution boundary must interact with the rest of the parts of the application. It will often create a REPLICATED OBJECT that also resides on the client. It may also be able to use and interpret DISTRIBUTED COMMANDS that are sent to the server to determine what actions to take within the server.

DISTRIBUTED COMMAND

When developing a distributed application that uses REPLICATED OBJECTS, object updates must be sent in both directions across the network connection (from client to server and server to client) to maintain consistency between the original object and its replicates. When replicating objects, sending updates between client and server can lead to efficiency problems for large objects or make planning for transactions and other architectural features overly complex. This pattern documents a way to encapsulate changes as composite groups of commands.

For instance, let's suppose you are working on an application that allows a user to modify a complex, highly interrelated object model. Consider a genomics system that tracks genetic markers through a family tree in order to pinpoint how genes are inherited. There are at least three axes of information that users would be interested in:

- The family tree itself (who descends from who)
- The information about individuals in the tree (who showed what symptoms and who's assays showed them to have what markers)
- Information about the genetic markers (what markers are used, where they are found on the genome, etc.)

Encapsulate the user changes as commands and treat the group of commands as a single composite command that executes within a transaction boundary.

The problem is that the three axes are interlocking. Modifications to one object can have serious ramifications to other parts of the application. For instance, if a marker is changed, its relation to the individuals must change. Likewise, if an individual's genetic assay is found to be incorrect and changes, then statistics and calculations about the family and markers might now be incorrect. There are two "standard" approaches to maintaining the consistency of this information on the entire structure that can be attempted:

- P*essimistic concurrency*. In this case, a large chunk of the object structure is locked when the first user requests it. This has the drawback that other users are kept from modifying the structure at the same time – something that is not reasonable in a multi-user environment.
- O*ptimistic concurrency*. In this case, users are allowed to modify the structure as they choose, but the first one to "commit" his changes "wins". Anyone who had also modified those same structures would "lose" and find that their changes were now lost.

A third approach, versioning, can also be tried. In this case, a new version of the structure is created for each user. However, this just trades the current problem for a different, but equally difficult, version reconciliation problem. A different solution is required.

Therefore:

Encapsulate the user changes as commands and treat the group of commands as a single composite command that executes within a transaction boundary. Use a strategy like two-phase commit to merge commands issued by different users together.

In a distributed system this solution becomes even more attractive. Imagine that our hypothetical genomics system was built using a layered architecture, with the genetic objects being replicated objects. Further, imagine that we had a façade at the distribution boundary to encapsulate the "real" interactions of the domain model so that the GUI front-end only communicated with the business model through the intermediaries of the façade components.

In this case we find that commands not only help with the concurrency control of the system, but also provide a significant benefit in that the changes that are sent from the upper (presentation) layers of an application to the lower layers are sent in the form of "deltas" to objects, rather than full copies of the objects themselves. This reduces the amount of network traffic, and reduces the amount of logic needed on the server side to determine which parts of the model have changed and which have not.

A known use of this pattern is information bus middleware (e.g., TIBCO Rendezvous, Progress SonicMQ, SoftWired iBus, Talarian SmartSockets, etc.) software that allows publishers to send event notifications to subscribers in an efficient, reliable manner. Some of the vendors conform to the J2EE 2.0 specification and have designed their software to be compatible with message-driven beans. While each vendor has their own internal structure and implementation for messaging, they support the JMS standard for Java messaging.

One possible implementation of this pattern is to encapsulate this information bus middleware with distributed commands that takes more of an event-driven approach to allowing message flow between remote objects and their replicates to occur in real-time.

In addition, object interaction protocols based on open standards such as XML have emerged. Among the most popular are Simple Object Access Protocol (SOAP), XML-RPC, Web Services Description Language (WSDL), and Universal Description and Discovery Integration (UDDI). All of them utilize XML to describe remote object invocations.

If taking a web services architecture where one or more of these interaction protocols are being implemented and operated by application servers such as IBM WebSphere, another implementation would be to use distributed commands to encapsulate use of one or more of these protocols.

FAÇADE AT THE DISTRIBUTION BOUNDARY can be used to accept distributed commands and act as an intermediary that balances load across multiple servers. COMPONENT BUS could be applied to implement event-driven behavior in distributed commands. ABSTRACT FACTORY can be used for implementations that need to support multiple strategies for executing distributed commands.

Acknowledgements

We would especially like to thank our Shepherd, Robert Hirschfield for all the hard work he put into commenting and providing great suggestions for improvement on the patterns in this mini-language as they were submitted to PLoP'99.

Bibliography

[98-01-18] OMG TC Document orbos/98-01-18, "Objects By Value" Joint Revised Submission with Errata, Object Management Group

[98-07-01] OMG TC Document 98-07-01, "The Complete CORBA/IIOP 2.2 Specification", Object Management Group

[Alexander] Christopher Alexander et al, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, London, 1977

[Alpert] Sherman R. Alpert, et al, *The Design Patterns Smalltalk Companion*, Addison-Wesley Longman, Reading, MA, 1998

[Buschmann] Frank Buschmann, et al, *Pattern-Oriented Software Architecture*: A System of Patterns, John Wiley and Sons, West Sussex, England, 1996

[Fowler] Martin Fowler, *Analysis Patterns*, Addison-Wesley Longman, Reading MA, 1996

[Gamma] Erich Gamma et.al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Longman, Reading MA,1994

[Sun] "EJB Learning Center"; http://java.sun.com/products/ejb/training.html

This miniature pattern language is part of a publication titled *Component Design Patterns: A Pattern Language for Component Based Development*, which will be published by Addison-Wesley in 2001. Therefore, please note that the publication and contents of this pattern language are part of a work in progress and subject to change prior to its publication.

Contact Philip Eskelin: philip@eskelin.com

FIND OUT HOW TO BUILD AND SELL SOFTWARE COMPONENTS

ComponentSource and Ratio join forces to bring you technical and market data content on Component Based Development (CBD)

This is a rare opportunity to get the full picture on what is happening in the software industry with regard to reusable software

14th June 2001 - ½ Day Free Seminar – London

REVEALED

Discover

How to designing extensible components and frameworks
How to assemble, test and document components
How to create a layered reference model for CBD

What's more….

Receive exclusive market research data, source PricewaterhouseCoopers
Find out which business software components are in demand
Discover the route to successful sales of your software components

BOOK NOW!

For more information or to register contact Andrea Graham

andreag@componentsource.com / Telephone +44 (0) 118 982 2102

This free event is open to all and will be held at the Grange Fitzrovia Hotel, Bolsover Street, London, W1 on the 14th June 2001, 9.30am until 12.30pm

WE KNOW THE OBJECT OF

TRAINING

Excellence in Object and Component Training

We offer a variety of both in-house and public schedule OO-related training courses.

New!

XML for Software Developers (RAT 290)

This **recently updated** 4-day course gives you a sound theoretical understanding of XML & its related specifications, while providing valuable practical experience in implementing & applying XML within applications. It covers a range of tools, technologies & approaches essential for managing the data interchange requirements of a distributed computing environment.

New!

CBD & Advanced OO Design with UML (RAT 111)

This **NEW** 3-day course gives you a firm understanding of how to analyse and design extensible and customisable re-usable business & technical (domain) components, and how to assemble such components to create bespoke applications. The course has a clear focus on the Advanced Principles of OOD needed to achieve this end.

New!

Java Development Workshop (RAT220)

This **recently updated** 5-day course will give you a practical understanding of the major features of the Java development environment and language, both in the context of web applets, and in the context of stand-alone applications. Students will leave the course able to start productive work immediately.

New!

Design Patterns & Advanced Principles of OOD (RAT160)

After this **NEW** 3-day course, students will be able to apply design pattern in their everyday development work, discuss & communicate with each other using pattern terminology, and understand & be able to apply the Advanced Principles of OOD to create higher quality, more flexible software in their everyday work.

Object-Oriented Analysis & Design using UML (RAT102)

This 5-day course gives you a practical understanding of the major techniques of the UML (Unified Modelling Language) object-oriented analysis & design notation, and how these techniques can be applied to improve quality and productivity during the analysis and design of computer systems.

Bespoke Courses

We are committed to meet your needs and providing a high level of services. We understand that each and individual client is different, therefore we tailor our training services to suit your needs. Contact us for a customised training course that is specially designed for you.

Email info@ratio.co.uk or call us on +44 (0)20 8579 7900 for more information.