Chapter 30

# DESIGNING THE LOGICAL ARCHITECTURE WITH PATTERNS

## Objectives

Design a logical architecture in terms of layers and partitions with the Layers pattern.

Illustrate the logical architecture using UML package diagrams.

Apply the Facade, Observer and Controller patterns.

## Introduction

First, to set the expectation level, this is an *introduction* to the topic of logical architecture, a fairly large topic.

The prior iterations emphasized a strongly related group of "domain" software objects' in the Design Model (such as *Sale* and *Payment).* No attention was paid to the user interface or access to resources such as a database. The motivation was to keep things simple and focus on core object design skills.

However, a typical system is composed of many logical packages, such as a user interface package, a database access package, and so forth. Each package groups a set of cohesive responsibilities (e.g., database access). This is the basic practice of modularization to support a separation of concerns.

This chapter briefly explores logical architectures, and communication and coupling between packages.

## 30.1 Software Architecture

One definition of **software architecture is:**

> An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization---these elements and their interfaces, their collaborations, and their composition. [BRJ99]

Regardless of the definition (and there are many) the common theme in all software architecture definitions is that it has to do with the large scale—the Big Ideas in the forces, organization, styles, patterns, responsibilities, collaborations, connections, and motivations of a system (or a system of systems), and major subsystems.

In software development, architecture is thought of as both a noun and a verb.

As a noun, the architecture includes—as the prior definition indicates—the organization and structure of the major elements of the system. Beyond this static definition, it includes the system behavior, especially in terms of large scale responsibilities of systems and subsystems, and their collaborations. In terms of a description, the architecture includes the *motivations* or rationale for why the system is designed the way it is.

As a verb, architecture is part investigation and part design work; for clarity, the term is best qualified, as in architectural investigation or architectural design.

**Architectural investigation** involves identifying those functional and (especially) non-functional requirements that have (or should have) a significant impact on the system design, such as market trends, performance, cost, maintainability, and points of evolution. Broadly, it is requirements analysis with a focus on those requirements that have special influence on the major system design decisions.

**Architectural design** is the resolution of these forces and requirements in the design of the software, the hardware and networking, operations, policies, and so forth.

In the UP, architectural investigation and design are together called **architectural analysis,** the process of which is briefly introduced in Chapter 32.

## Architectural Dimensions and Views in the Unified Process

The architecture of a system encompasses several dimensions. For example:

- The logical architecture, which describes the system in terms of its conceptual organization in layers, packages, major frameworks, classes, interfaces, and subsystems.
- The deployment architecture, which describes the system in terms of the allocation of processes to processing units, and the network configuration.

The Unified Process suggests six views of the architecture (logical, deployment, and so on), all of which are defined in Chapter 32.

This chapter focuses on a logical view of the architecture.

## Architectural Patterns and Pattern Categories

There are well-known best practices in architectural design, especially regarding large-scale logical architecture, and these have been written as patterns, such as Layers. The first book dedicated to the subject of architectural patterns *was Pattern-Oriented Software Architecture* (POSA) [BMRSS96].

The POSA book also offered a simple, useful categorization of patterns at different levels:

1. **Architectural patterns**—related to the large-scale and coarse-grained
design, and typically applied during the early iterations (the elaboration phase) when the major structures and connections are established.
    - The Layers patterns, which structures a system into major layers.
2. **Design patterns**—related to the small and medium-scale design of objects
and frameworks. Applicable to designing a solution for connecting the large scale elements defined via architectural patterns, and during detailed
design work for any local design aspect. Also known as micro-architectural patterns.
    - The Facade pattern, which can be used to provide the interface from one layer to the next.
    - The Strategy pattern, to allow pluggable algorithms.
3. **Idioms**—language or implementation-oriented low-level design solutions.
    - The Singleton pattern, to ensure global access to a single instance of a class.

This chapter focuses on architectural patterns and the application of design patterns to make connections between the large-scale structures.

**There are other pattern categories.** The POSA categories form a neat triad, and are useful for many patterns, but do not cover the entire gamut of published patterns. As the risk of oversimplification, a pattern is the repeating best practice of what works—in any domain. Other published categories of patterns include:

- organizational and software development process patterns
- user interface patterns
- testing patterns

## 30.2 Architectural Pattern: Layers

**Solution** The essential ideas of the Layers pattern [BMRSS96] are simple:

- Organize the large-scale logical structure of a system into discrete layers of distinct, related responsibilities, with a clean, cohesive separation of concerns such that the "lower" layers are low-level and general services, and the higher layers are more application specific.
- Collaboration and coupling is from higher to lower layers; lower-to-higher layer coupling is avoided.

A layer is a large-scale element, often composed of several packages or subsystems.

The Layers pattern relates to the logical architecture; that is, it describes the conceptual organization of the design elements into groups, independent of their physical packaging or deployment.

Layers defines a general N-tier model for the logical architecture; it produces a **layered architecture.** It has been applied and written about so often as a pattern that the *Pattern Almanac 2000* [Rising00] lists over 100 patterns that are variants of or related to the Layers pattern.

**Problems**

- Source code changes are rippling throughout the system—many parts of the systems are highly coupled.
- Application logic is intertwined with the user interface, and so can not be reused with a different interface, nor distributed to another processing node.
- Potentially general technical services or business logic is intertwined with more application-specific logic, and so can not be reused, distributed to another node, or easily replaced with a different implementation.

- There is high coupling across different areas of concern (as suggested in the previous problems). It is thus difficult to divide the work along clear bound aries for different developers.
- Due to the high coupling and mixing of oncerns, it is laborious and costly to evolve the application's functionality, scale up the system, or update it to use new technologies.

**Example** The purpose and number of layers varies across applications and application domains (information systems, operating systems, and so forth. Applied to information systems, typical layers are illustrated and explained in Figure 30.1.

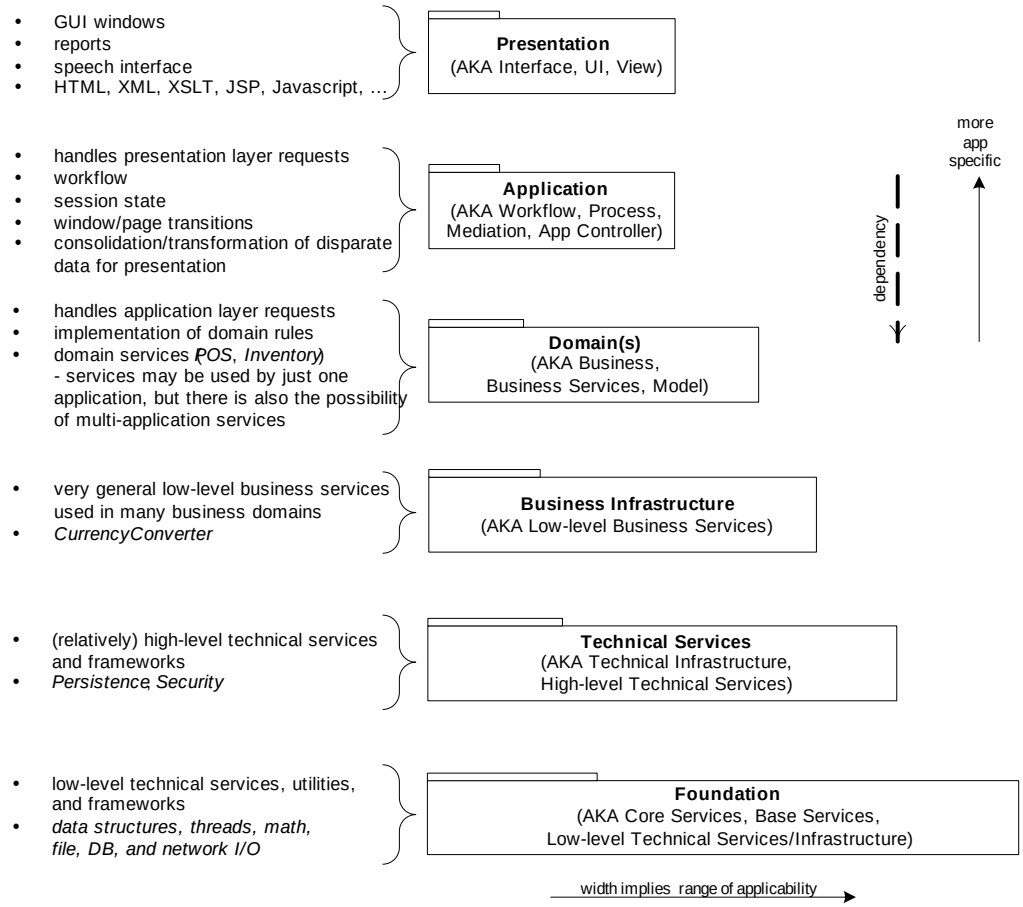


Figure 30.1 Common layers in an information system logical architecture.[1]

Based on these archetypes, Figure 30.2 illustrates a partial logical layered architecture for the NextGen application.

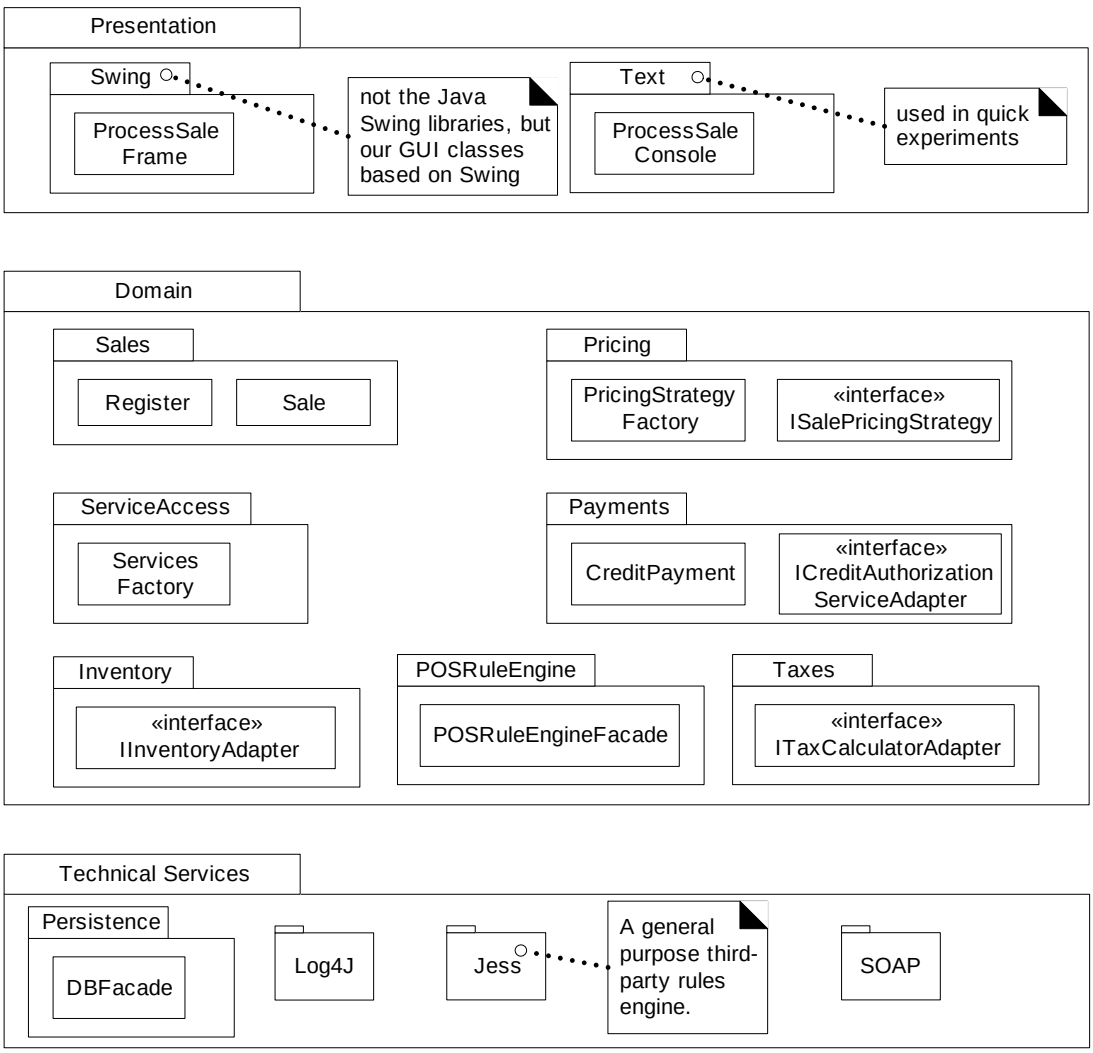



Figure 30.2 Partial logical view of layers in the NextGen application.

*UML notation*—Package diagrams are used to illustrate the layers. In the UML, a layer is simply a package.

1. The width of the package is used to communicate range of applicability in this diagram, but this is not a general UML practice. AKA means also known as.

Note the absence of an Application layer for this iteration of the design; as discussed later, it is not always necessary.

Since this is iterative development, it is normal to create a design of layers that starts simple, and evolves over the iterations of the elaboration phase. One goal of this phase is to have the core architecture established (designed and implemented) by the end of the iterations in elaboration, but this does not mean doing a large up-front speculative architectural design before starting to program. Rather, a tentative logical architecture is designed in the early iterations, and it evolves incrementally through the elaboration phase.

Observe that just a few sample types are present in this package diagram; this is not only motivated by limited page space in formatting this book, but is a signature quality of an **architectural view** diagram—it only shows a few noteworthy elements in order to concisely convey the major ideas of the architecturally significant aspects. The idea in a UP architectural view document is to say to the reader, "I've chosen this small set of instructive elements to convey the big ideas."

*Diagram Comments:*

- There are other types in these packages; only a few are shown to indicate noteworthy aspects.
- The Foundation layer was not shown in this view; the architect (me) decided it did not add interesting information, even though the development team will certainly be adding some Foundation classes, such as more advanced *String* manipulation utilities.
- For now, a separate Application layer is not used. The responsibilities of control or session objects in the Application layer are handled by the *Register* object. The architect will add an Application layer in a later iteration as the behavior grows in complexity, and alternative client interfaces are introduced (such as a web browser and wireless networked handheld PDA).

## Inter-Layer and Inter-Package Coupling

It is also informative to include a diagram in the logical view that illustrates noteworthy coupling between the layers and packages. A partial example is illustrated in Figure 30.3.

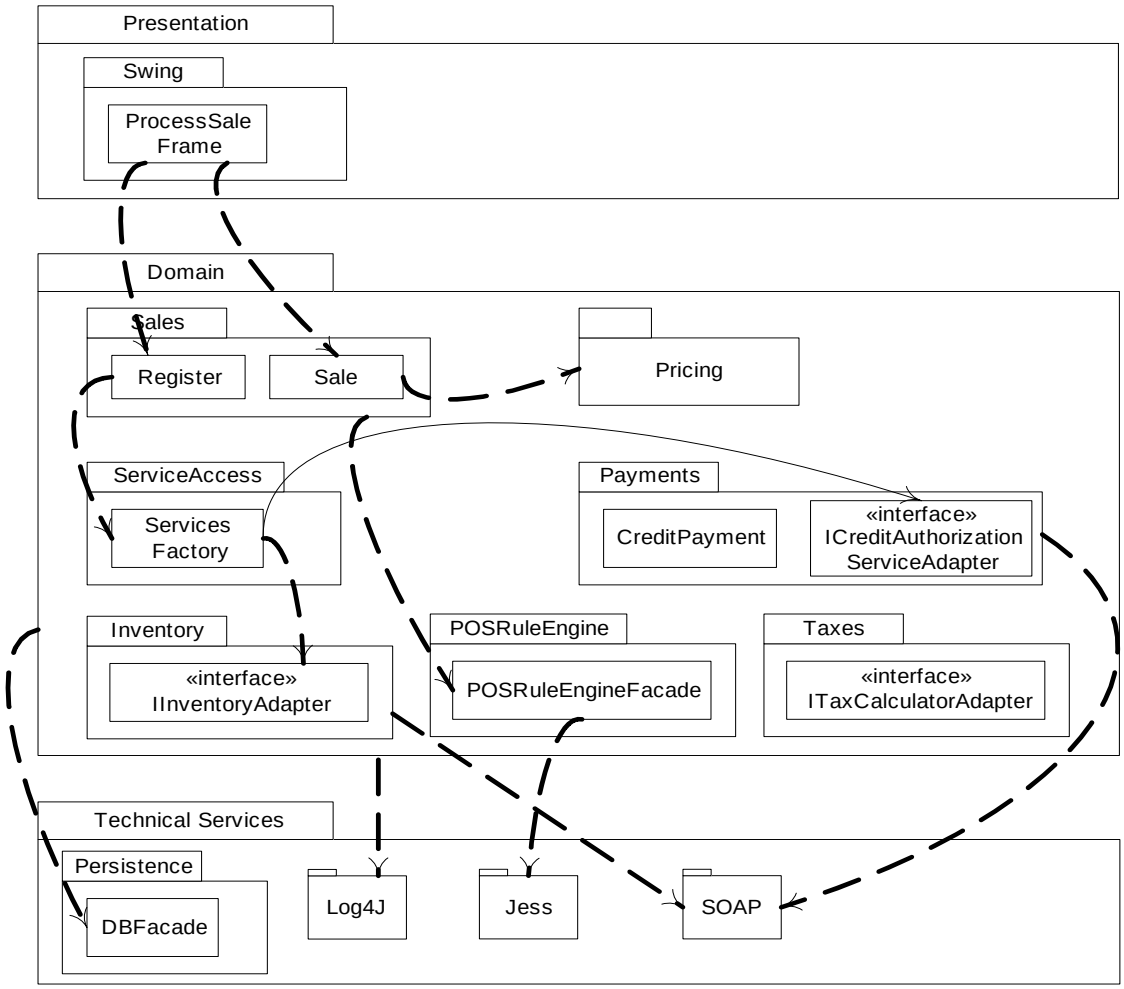



Figure 30.3 Partial coupling between packages.

*UML notation:*

- Observe that dependency lines can be used to communicate coupling between packages or types in packages. Plain dependency lines are excellent when the communicator does not care to be more specific on the exact dependency (attribute visibility, subclassing, ...), but just wants to highlight general dependencies.
- Note also the use of a dependency line emitting from a package rather than a particular type, such as from the *Sales* package to *POSRuleEngineFacade* class, and the *Domain* package to the *Log4J* package. This is useful when either the specific dependent type is not interesting, or the communicator

wants to suggest that many elements of the package may share that dependency.

Another common use of a package diagram is to hide the specific types, and focus on illustrating the package-package coupling, as in the partial diagram of Figure 30.4.

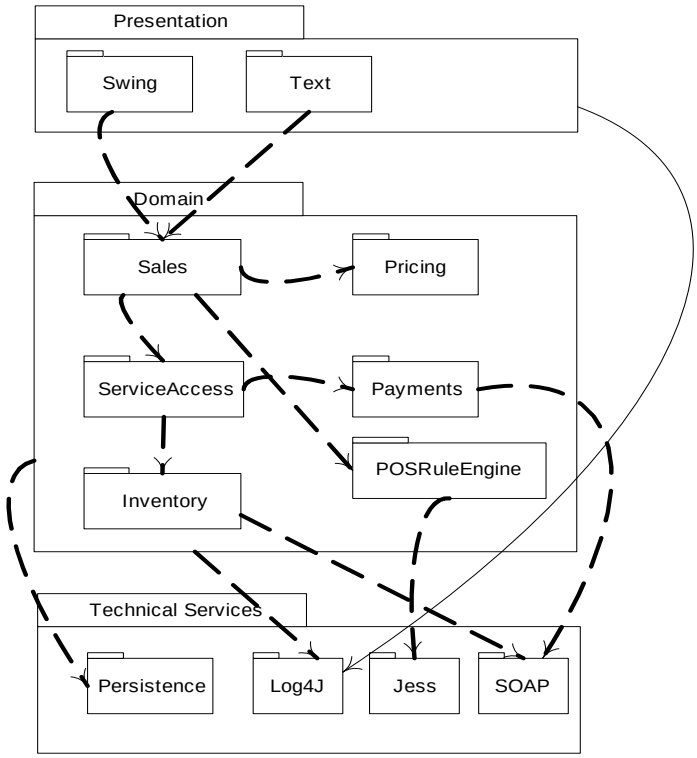



Figure 30.4 Partial package coupling.

In fact, Figure 30.4 illustrates probably the most common style of logical architecture diagram in the UML—a package diagram that shows between perhaps 5 to 20 major packages, and their dependencies.

## Inter-Layer and Inter-Package Interaction Scenarios

Package diagrams show static information. To understand the dynamics of how objects across the layers connect and communicate, an interaction diagram is informative. In the spirit of an "architectural view" which hides uninteresting details, and emphasizes what the architect wants to convey, an interaction dia-

gram in the logical view of the architecture focuses on the collaborations as they cross layer and package boundaries. A set of interaction diagrams that illustrate **architecturally significant scenarios** (in the sense that they illustrate many aspects of the large-scale or big ideas in the design) is thus useful.

For example, Figure 30.5 illustrates part *of a Process Sale* scenario that emphasizes the connection points across the layers and packages.

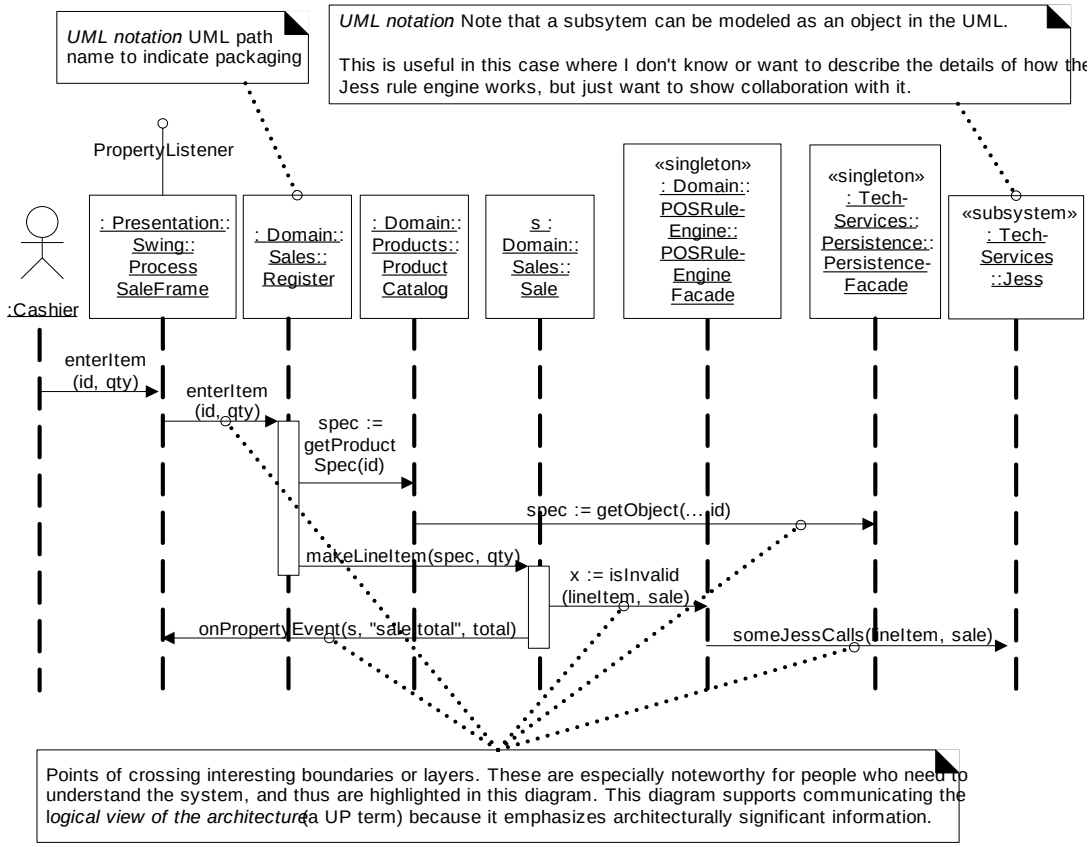



Figure 30.5 An architecturally significant interaction diagram that emphasizes cross-boundary connections.

*UML notation:*

- The package of a type can optionally be shown by qualifying the type with the UML **path name** expression <*PackageName*>::<*TypeName*>. For exam-

ple, *Domain::Sales::Register*. This can be exploited to highlight to the reader the inter-package and inter-layer connections in the interaction diagram.

- Note also the use of the « subsystem» stereotype. In the UML, a subsystem is a discrete entity that has behavior and interfaces. A subsystem can be modeled as a special kind of package, or—as shown here—as an object, which is useful when one wants to show inter-subsystem (or system) collaborations. In the UML, the entire system is also a "subsystem" (the root one), and thus can also be shown as an object in interaction diagrams (such as an SSD).

Observe that the diagram ignores showing some messages, such as certain *Sale* collaborations, in order to highlight architecturally significant interactions.

**Collaborations** Two design decisions at an architectural level are:

1. What are the big parts?
2. How are they connected?

Whereas the architectural Layers pattern guides defining the big parts, micro-architectural design patterns such as Facade, Controller, and Observer are commonly used for the design of the connections between layers and packages. This section examines patterns in connection and communication between layers and packages.

### Simple Packages vs. Subsystems

Some packages or layers are not just conceptual groups of things, but are true subsystems with behavior and interfaces. To contrast:

- The *Pricing* package is not a subsystem; it simply groups the factory and strategies used in pricing. Likewise with Foundation packages such as *java.util*.
- On the other hand, the *Persistence, POSRuleEngine,* and *Jess* packages are subsystems. They are discrete engines with cohesive responsibilities that do work.

In the UML, a subsystem can be identified with a stereotype, as in Figure 30.6.

### Facade

For packages that represent subsystems, the most common pattern of access is Facade, a GoF design pattern. That is, a public facade object defines the services for the subsystem, and clients collaborate with the facade, not internal subsystem components. This is true of the *POSRuleEngineFacade* and the *PersistcnceFacade* for access to the rules engine and persistence subsystem.

The facade should not normally expose many low-level operations. Rather, it is desirable for the facade to expose a small number of high-level operations—the coarse-grained services. When a facade does expose many low-level operations,

it tends to become incohesive. Furthermore, if the facade will be, or might become, a distributed or remote object (such as an EJB session bean, or RMI server object), fine-grained services lead to remote communication performance problems—lots of little remote calls are a performance bottleneck in distributed systems.

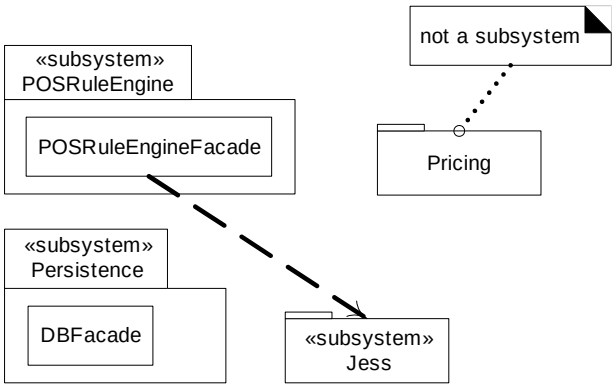



Figure 30.6 Subsystem stereotypes.

Also, a facade does not normally do its own work. Rather, it is consolidator or mediator to the underlying subsystem objects, which do the work.

For example, the *POSRuleEngineFacade* is the wrapper and single point of access into the rules engine for the POS application. Other packages do not see the implementation of this subsystem, as it is hidden behind the facade. Suppose (this is just one of many implementations) that the POS rules engine subsystem is implemented by collaborating with the Jess rules engine. Jess is a subsystem which exposes many fine-grained operations (this is common for very general, third-party subsystems). But the *POSRuleEngineFacade* does not expose the low level Jess operations in its interface. Rather, it provides only a few high-level operation such as *isInvalid(lineltem, sale).*

If the application has only a "small" number of system operations, then it is common for the Application or Domain layer to expose only one object to an upper layer. On the other hand, the Technical Services layer, which contains several subsystems, exposes at least one facade (or several public objects, if facades aren't used) for each subsystem to upper layers. See Figure 30.7.

## Session Facades and the Application Layer

In contrast to Figure 30.7, when an application has many system operations and supports many use cases, it is common to have more than one object mediating between the Presentation and Domain layers.

In the current version of the NextGen system, there is a simple design of a single *Register* object acting as the facade onto the Domain layer (by virtue of the GRASP controller pattern).

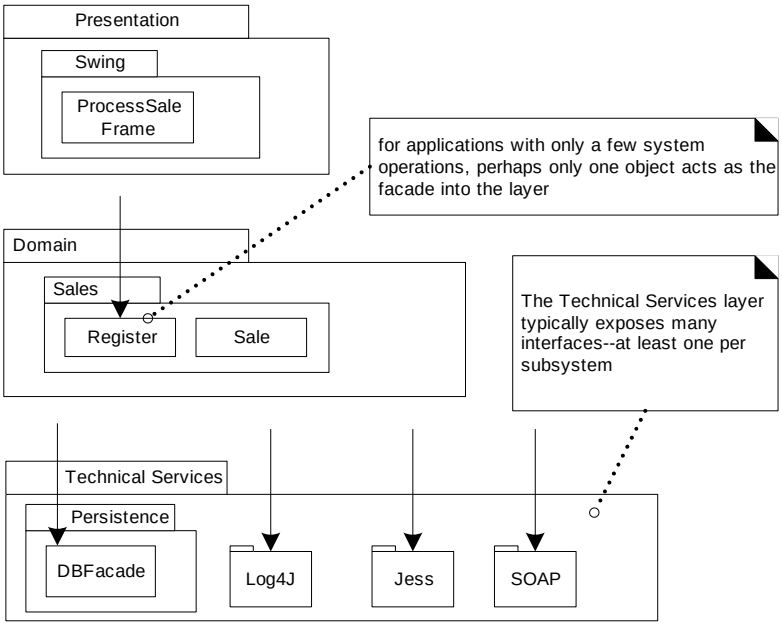



Figure 30.7 Number of interfaces exposed to upper layers.

However, as the system grows to handle many use cases and system operations, it is not uncommon to introduce an Application layer of objects that maintain session state for the operations of a use case, where each session instance represents a session with one client. These are called Session Facades, and their use is another recommendation of the GRASP Controller pattern, such as in the use-case session facade controller variant of the pattern. See Figure 30.8 for an example of how the NextGen architecture may evolve with an Application layer and session facades.

## Controller

The GRASP Controller pattern describes common choices in client-side handlers (or controllers, as they've been called) for system operation requests emitting from the Presentation layer. Figure 30.9 illustrates.

Figure 30.9 The Controller choices.

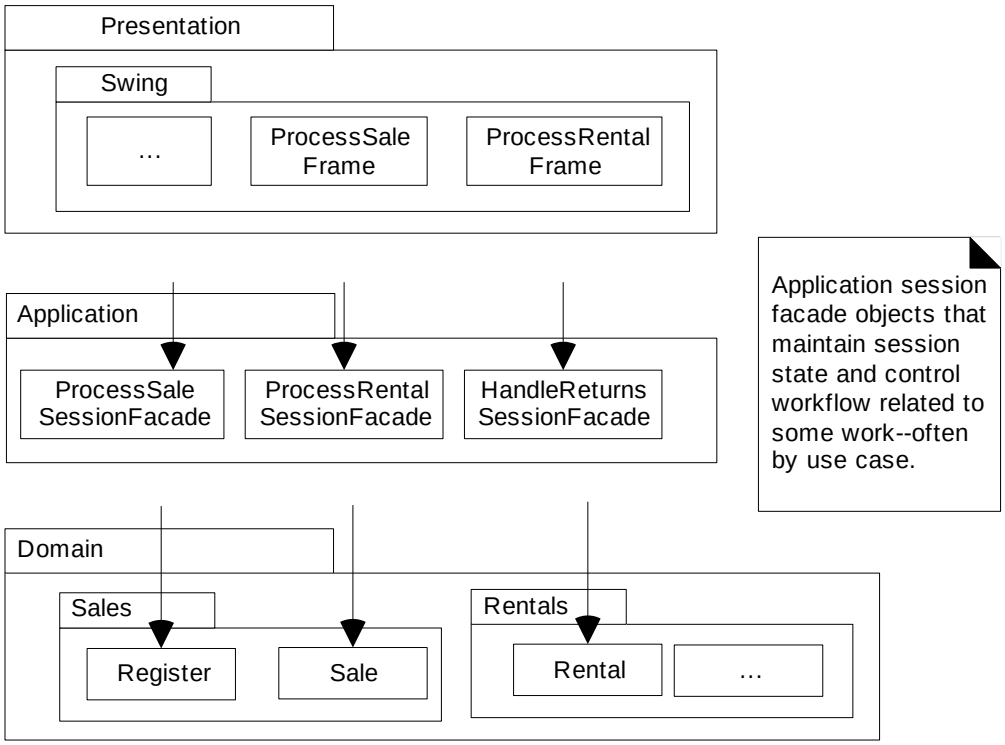



Figure 30.8 Session facades and an Application Layer.

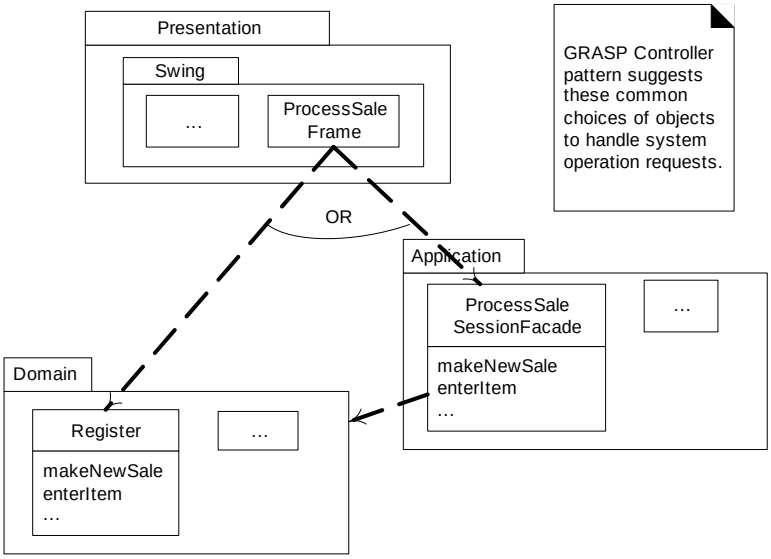



Figure 30.9 The Controller choices

## System Operations and Layers

The SSDs illustrate the system operations, hiding presentation objects from the diagram. The system operations being invoked on the system in Figure 30.10 are requests being generated by an actor via the Presentation layer, onto the Application or Domain layer.

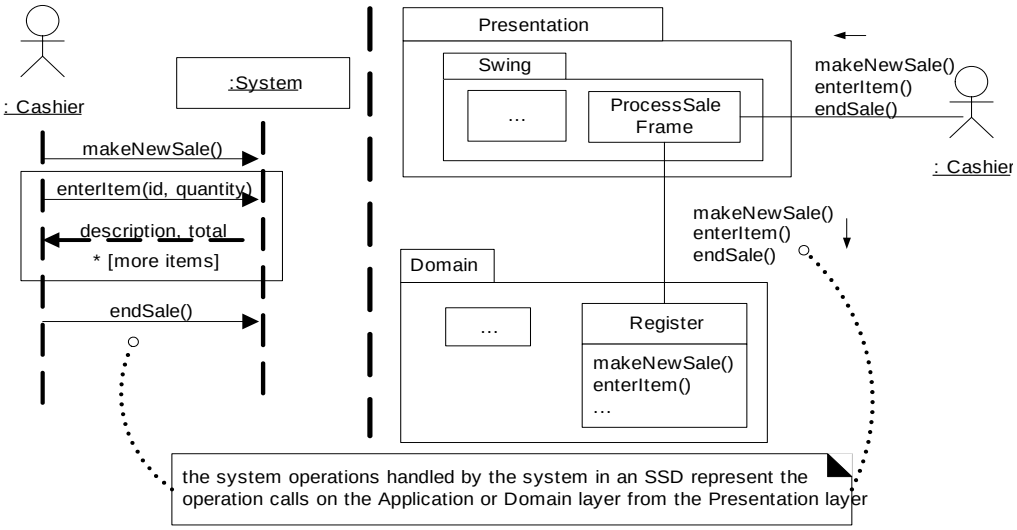



Figure 30.10 System operations in the SSDs and in terms of layers.

## Upward Collaboration with Observer

The Facade pattern is commonly used for "downward" collaboration from a higher to a lower layer, or for access to services in another subsystem of the same layer. When the lower Application or Domain layer needs to communicate upward with the Presentation layer, it is usually via the Observer pattern. That is, UI objects in the higher Presentation layer implement an interface such as *Property Listener* or *AlarmListener,* and are subscribers or listeners to events (such as property or alarm events) coming from objects in the lower layers. The lower layer objects are directly sending messages to the upper layer UI objects, but the coupling is only to the objects viewed as things that implement an interface such as *PropertyListener,* not viewed as specific GUI windows.

This was examined when the Observer pattern was introduced. Figure 30.11 summarizes the idea in relation to layers.

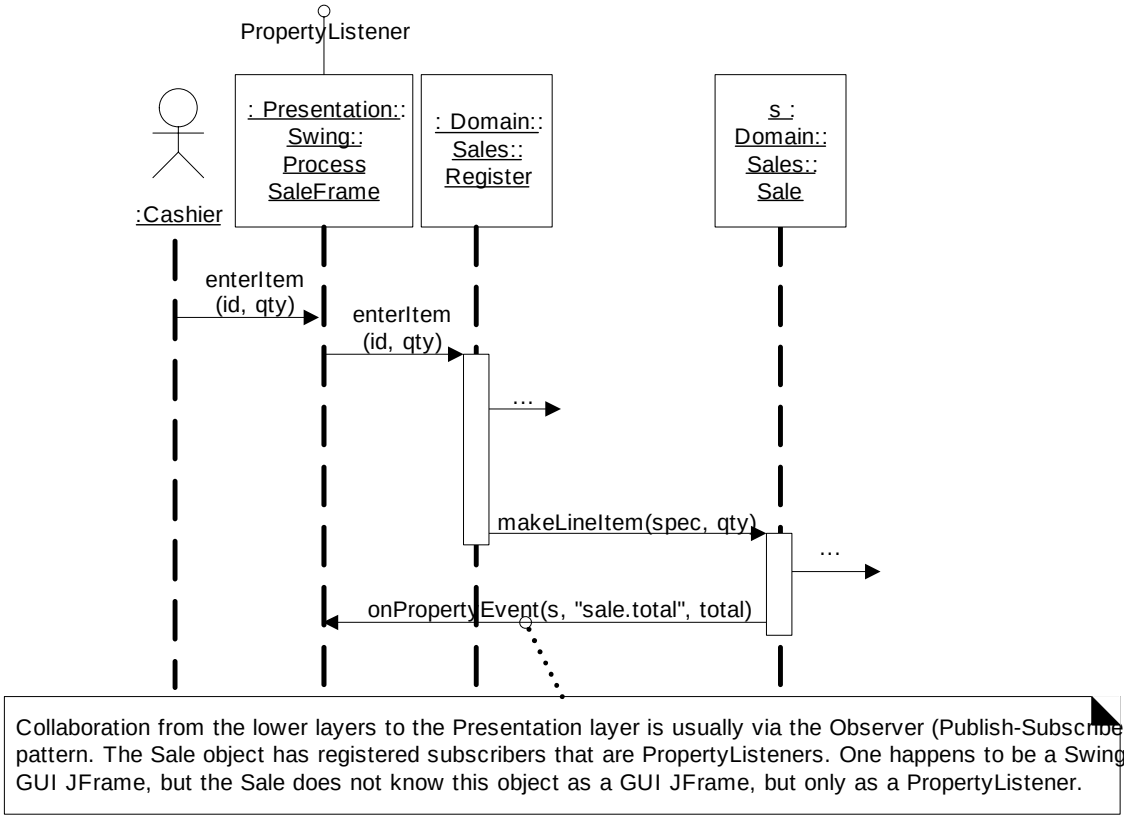



Figure 30.11 Observer for "upward" communication to the Presentation layer.

## Relaxed Layered Coupling

The layers in most layered architectures are *not* coupled in the same limited sense as a network protocol based on the OSI 7-Layer Model. In the protocol model, there is strict restriction that elements of layer N only access the services of the immediate lower layer N-l.

This is rarely followed in information system architectures. Rather, the standard is a "relaxed layered" or "transparent layered" architecture IBMRSS96], in which elements of a layer collaborate with or are coupled to several other layers.

Comments on typical coupling between layers:

- All higher layers have dependencies on the Technical Services and Founda tions layer.
    - For example, in Java all layers depend *onjava.util* package elements.
- It is primarily the Domain layer that has dependency on the Business Infra structure layer.

- The Presentation layer makes calls on the Application layer, which makes service calls on the Domain layer; the Presentation layer does not call on the Domain, unless there is no Application layer.
- If it is a single-process "desktop" application, software objects in the Domain layer are directly visible to, or passed between, Presentation, Application, and to a lesser extent, Technical Services.
    - For example, assuming the NextGen POS system is of this type, a *Sale* and a *Payment* object could be directly visible to the GUI Presentation Layer, and also passed into the Persistence subsystem in the Technical Services layer.
- On the other hand, if it is a distributed system, then serializable **replicates** (also known as **data holder or value objects)** of objects in the Domain layer are usually passed to a Presentation layer. In this case, the Domain layer is deployed on a server computer, and client nodes get copies of server data.

### *Isn't Coupling to Technical Service and Foundation Layers Dangerous?*

As the GRASP Protected Variations and Low Coupling discussions explored, it is not coupling per se that is a problem, but unnecessary coupling to variation and evolution points that are unstable and expensive to fix. There is very little justification in spending time and money attempting to abstract or hide something that is unlikely to change, or if it did, the change impact cost would be negligible. For example, if building a Java technologies application, what value is there in hiding the application from access to the Java libraries? High coupling into many points of the libraries is an unlikely problem, as they are (relatively) stable and ubiquitous.

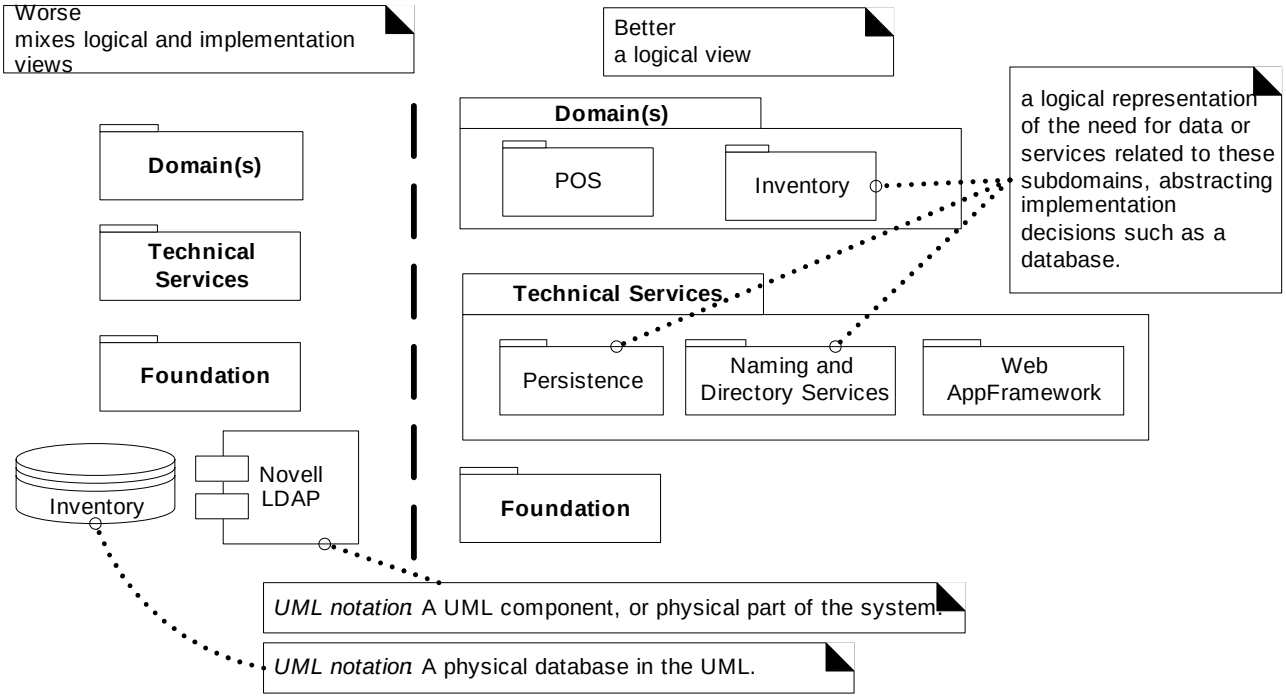
**Discussion** In addition to the structural and collaboration issues discussed above for this pattern, other issues include the following.

## External Resources or External Database Layer at the Bottom?

Most systems rely on external resources or services, such as an Oracle database and a Novell LDAP naming and directory service. These are *physical* implementation components, not a layer in the *logical* view of the architecture.

Showing external resources such as a particular database in a layer "below" the Foundation layer (for example) mixes up the logical view and the deployment or implementation views of the architecture.

Rather, in terms of the logical view of the architecture and its layers, access to a particular set of persistent data (such as inventory data) can be viewed as a sub-domain of the Domain Layer—the Inventory subdomain. And the general services that provide access to databases may be viewed as a Technical Service partition—the Persistence service. See Figure 30.12.




30.12 Mixing views of the architecture.

## Logical vs. Process and Deployment Views of the Architecture

The architectural layers are a logical view of the architecture, not a deployment view of elements to processes and processing nodes. Depending on the platform, *all* layers could be deployed within the same process on the same node, such as an application within a handheld PDA, or spread across many computers and processes for a large-scale web application.

The UP Deployment Model that maps this logical architecture to processes and nodes is strongly influenced by the choice of software and hardware platform and associated application frameworks. For example, J2EE versus .NET influence the deployment architecture.

There are many ways to slice and dice these logical layers for deployment, and in general the subject of deployment architecture will only be lightly introduced, as it is non-trivial, largely outside the scope of the book, and dependent on detailed discussion of the chosen software platform, such as J2EE.

## Optional Application Layer?

If present, the Application layer contains objects responsible for knowing the session state of clients, mediating between the Presentation and Domain layers, and controlling the flow of work.

The flow may be organized by controlling the order of windows or web pages, for example.

In terms of the GRASP patterns, GRASP Controller objects such as a use case facade controller are part of this layer. In distributed systems, components such as EJB session beans (and stateful session objects in general) are part of this layer.

In some applications, this layer is not required. It is useful (this is not an exhaustive list) when one or more of the following is true:

- Multiple user interfaces (for example, web pages and a Swing GUI) will be used for the system. The Application layer objects can act as Adapters that collect and consolidate the data as needed for different UIs, and as Facades that wrap and hide access to the Domain layer.
- It is a distributed system and the Domain layer is on a different node than the Presentation layer, and shared by multiple clients. It is usually necessary to keep track of session state, and Application layer objects are a useful choice for this responsibility.
- The Domain Layer can not or should not maintain session state.
- There is a defined workflow in terms of the controlled order of windows or web pages that must be presented.

## Fuzzy Set Membership in Different Layers

Some elements are strongly a member of one layer; a *Math* class is part of the Foundation layer. However, especially between the Technical Services and Foundation layers, and Domain and Business Infrastructure, some elements are harder to classify, because the differentiation between these layers is, roughly, "high" versus "low," or "specific" versus "general." which are fuzzy set terms. This is normal, and it is seldom necessary to decide upon a definitive categorization—the development team may consider an element roughly part of the Technical Services and/or Foundations layer considered as a group, broadly called the Infrastructure layer.[2]

For example:

- Suppose this is a Java technologies project, and the open source logging framework *Log4J* (part of the Jakarta project) has been chosen. Is logging part of the Technical Service or Foundation layer? Log4J is a low-level, small, general framework. It is moderately a member of both the Technical Services and the Foundations fuzzy sets.

2. Note that there are not well-established naming conventions for layers, and name overloading and contradiction in the architecture literature is common.

- Suppose this is a web application, and the Jakarta *Struts* framework for web applications has been chosen. Struts is a relatively high-level, large, specific technical framework. It is arguably strongly a member of the Technical Services set, and weakly a member of the Foundation set.

But, one person's High-level Technical Service is another's Foundation...

Finally, it is not the case that the libraries provided by a software platform only represent low-level Foundation services. For example, in both .NET and J2SE+J2EE, services include relatively high-level functions such as naming and directory services.

## Terminology: Tiers, Layers, and Partitions

The original notion of a **tier** in architecture was a logical layer, not a physical node, but the word has become widely used to mean a physical processing node (or cluster of nodes), such as the "client tier" (the client computer). This presentation will avoid the term for clarity, but bear this in mind when reading architecture literature.

The **layers** of an architecture are said to represent the vertical slices, while **partitions** represent a horizontal division of relatively parallel subsystems of a layer. For example, the *Services* layer may be divided into partitions such as *Security* and *Reporting* (Figure 30.13).

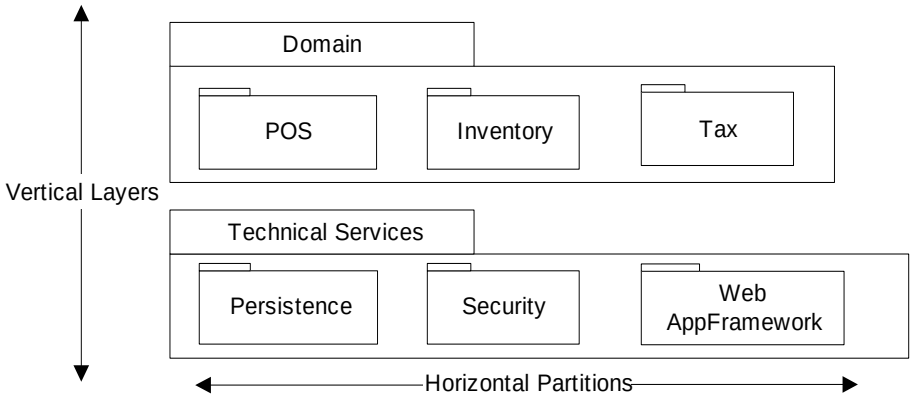



Figure 30.13 Layers and partitions.

**Contraindications and Liabilities**

- In some contexts, adding layers introduces performance problems. For example, in a high-performance graphics-intensive game adding layers of abstraction and indirection on top of direct access to graphics card components may introduce performance problems.
- The Layers pattern is one of several core architectural patterns; it is not applicable to every problem. For example, an alternate is Pipes and Filters [BMRSS96]. This is useful when the main theme of the application involves processing something through a series transformations, such as image

transformations, and the ordering of the transformations is changeable. Yet even in the case when the highest level architectural pattern is Pipes and Filters, individual pipes or filters can be design with Layers.

**Benefits**

- In general, there is a separation of concerns, a separation of high from low-level services, and of application-specific from general services. This reduces coupling and dependencies, improves cohesion, increases reuse potential, and increases clarity.
- Related complexity is encapsulated and decomposable.
- Some layers can be replaced with new implementations. This is generally not possible for lower-level Technical Service or Foundation layers (e.g., *java.util),* but may be possible for Presentation, Application, and Domain layers.
- Lower layers contain reusable functions.
- Some layers (primarily the Domain and Technical Services) can be distributed.
- Development by teams is aided because of the logical segmentation.

### implementation Implementing the Layers: People and Process

It is common and recommended, within an iteration, to have a developer specialize within one layer or one service.

Yet, it is not the case that the entire project team focuses on one layer or service in an iteration. Rather, it is more common to implement vertical slices across the layers. This is the UP approach in the elaboration phase: Choose scenarios and requirements that force, in each iteration, a broad coverage across many architecturally significant packages/layers/subsystems, in order to reveal and stabilize the major architectural elements in the early iterations.

However, in this book, this approach was not illustrated in the NextGen case study, because to do so would require early discussion across many and vast topics—from GUI programming to object-relational mapping and optimizing SQL statements. The book case study has focused on the design of Domain layer objects, while recognizing that in reality there would be parallel work going on to develop other layers and subsystems.

The design principles illustrated for the case study are applicable in virtually all layers of the design.

## Implementation View: Mapping Source Code Organization to Layers and Packages

Part of the UP Implementation Model is the organization of the source code. For languages such as Java or C#, which provide easy package (namespace) support, the mapping from the logical packaging to the implementation packaging is sim-

ilar, with notable exceptions when third-party libraries are used.[3] In fact, it is only in the early stages of development, when packages have been speculatively drawn, but not implemented, that there are meaningful differences.

Over time, as the code base grows, it is common to abandon the early speculative drawings (such as the ones we have just seen), and instead use a reverse-engineering UML CASE tool that reads the source code and generates a package diagram. Then, these automatically generated package diagrams, which accurately reflect the code (the real design) become the basis for the logical view of the architecture.

To use Java as an example for mapping to implementation packages, the layers and packages illustrated in Figure 30.4 might map to Java package names as follows:

```
//---- PRESENTATION

com.foo.nextgen.ui.swing
com.foo.nextgen.ui.text


//---- DOMAIN

    // packages relatively specific to the NextGen project
com.foo.nextgen.domain.sales com.foo.nextgen.domain.pricing
com.foo.nextgen.domain.serviceaccess
com.foo.nextgen.domain.posruleengine

    // packages that can easily be designed as
    // multi-application common business services
com.foo.domain.inventory
com.foo.domain.creditpayment


// ---  TECHNICAL SERVICES

    // our team creates
com.foo.service.persistencelite

    // third party
org.apache.log4j
org.apache.soap.rpc
jess


// ---  FOUNDATION

    // our team creates
com.foo.util
com.foo.stringutil
```

Notice that an effort has been made to avoid using a specific application qualifier ("nextgen") in the package names unless necessary. For example, the UI

3. C++ also supports namespaces, but it is awkward to use the language with dozens or hundreds of fine-grained namespaces; not so for Java or C#.

packages are related to the NextGen application, and so are qualified with the application name *com.foo.nextgen.ui.**.

To support reuse, one practice is to name elements in an application-independent manner, when appropriate. As a straightforward example, general purpose *String* utilities created by the NextGen team, are placed in *com.foo.stringutils*, not *com.foo.nextgen.stringutils*. Furthermore, *com.foo.stringutils* should be placed in the company's source code repository at a company level, rather than buried within the NextGen project's source code folders. You can't reuse it if you can't see it.

As another example, consider the services to access external third-party inventory and credit payment authorization systems. Although they were created by the NextGen team in the service of the NextGen POS project, they are general business services—one could imagine accessing inventory systems from within other applications; so too for credit payment authorization. Hence, *com.foo.domain.inventory* rather than *com.foo.nextgen.domain.inventory*.

On the other hand, the POSRuleEngine package is completely related to the NextGen POS project. Thus, *com.foo.nextgen.domain.posruleengine*.

If in doubt, qualify the package with the project name. It can always be refac-tored at a later date.

**Known Uses** A vast number of modern object-oriented systems (from desktop applications to distributed J2EE web systems) are developed with Layers; it might be harder to find one that is not, than is. Going farther back in history:

## Virtual Machines and Operating Systems

Starting in the 1960s, operating system architects advocated the design of operating systems in terms of clearly defined layers, where the "lower" layers encapsulated access to the physical resources and provided process and I/O services, and higher layers called on these services. These included Multics [CV65] and the THE system [Dijkstra68].

Earlier still—in the 1950s—researchers su ggested the idea of a virtual machine (VM) with a bytecode universal machine language (for example, UNCOL [Conwayl958]), so that applications could be written at higher layers in the architecture (and executed without recompilation across different platforms), on top of the virtual machine layer, which in turn would sit on top of the operating system and machine resources. A VM layered architecture was applied by Alan Kay in his landmark Flex object-oriented based personal computer system [Kay68] and later (1972) by Kay and Dan Ingalls in the influential Smalltalk virtual machine [GK76]—the progenitor of more recent VMs such as the Java Virtual Machine.

## Information Systems: The Classic Three-Tier Architecture

An early influential description of a layered architecture for information systems that included a user interface and persistent storage of data was known as a **three-tier architecture** (Figure 30.14), described in the 1970s in [TK78]. The phrase did not achieve popularity until the mid 1990s, in part due to its promotion in [Gartner95] as a solution to problems associated with the widespread use of two-tier architectures.

The original term is now less common, but its motivation is still relevant. A

classic description of the vertical tiers in a three-tier architecture is:

1. **Interface**—windows, reports, and so on.
2. **Application Logic**—tasks and rules that govern the process.
3. **Storage**—persistent storage mechanism.

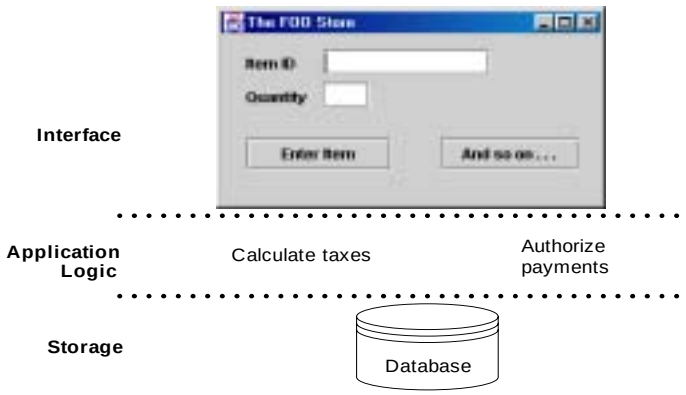



Figure 30.14 Classic view of a three-tier architecture.

The singular quality of a three-tier architecture is the separation of the application logic into a distinct logical middle tier of software. The interface tier is relatively free of application processing; windows or web pages forward task requests to the middle tier. The middle tier communicates with the back-end storage layer.

There was some misunderstanding that the original description implied or required a physical deployment on three computers, but the intended description was purely logical; the allocation of the tiers to compute nodes could vary from one to three. See Figure 30.15.

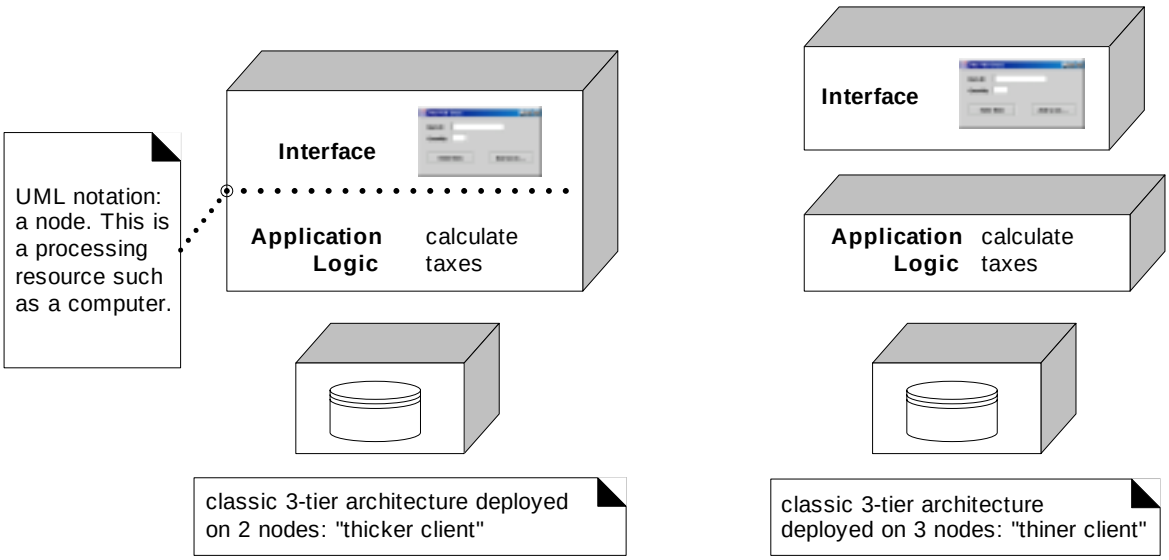



Figure 30.15 A three-tier logical division deployed in two physical architectures.

The three-tier architecture was contrasted by the Gartner Group with a **two-tier** design, in which, for example, application logic is placed within window definitions, which read and write directly to a database; there is no middle tier that separates out the application logic. Two-tier client-server architectures became especially popular with the rise of tools such as Visual Basic and PowerBuilder.

Two-tier designs have (in some cases) the advantage of initial quick development, but can suffer the complaints covered in the *Problems* section. Nevertheless, there are applications that are primarily simple CRUD (create, retrieve, update, delete) data intensive systems, for which this is a suitable choice.

**Related Patterns**

- Indirection—layers can add a level indirection to lower-level services.
- Protected Variation—layers can protect against the impact of varying implementations.
- Low Coupling and High Cohesion—layers strongly support these goals.
- Its application specifically to object-oriented information systems is described in [Fowler96].

**Also Known As** Layered Architecture [Shaw96, Gemstone00]

## 30.3 The Model-View Separation Principle

This principle has been discussed several times; this section summarizes it.

What kind of visibility should other packages have to the Presentation layer?

How should non-window classes communicate with windows? It is desirable that there is no direct coupling from other components to window objects because the windows are related to a particular application, while (ideally) the non-windowing components may be reused in new applications or attached to a new interface. The is the Model-View Separation principle.

In this context, **model** is a synonym for the Domain layer of objects. **View** is a synonym for presentation objects, such as windows, applets and reports.

The **Model-View Separation** principle[4] states that model (domain) objects should not have *direct* knowledge of view (presentation) objects, at least as view objects. So, for example, a *Register* or *Sale* object should not directly send a message to a GUI window object *ProcessSaleFrame*, asking it to display something, change color, close, and so forth.

As previously discussed, a legitimate relaxation of this principle is the Observer pattern, where the domain objects send messages to UI objects viewed only in terms of an interface such as *PropertyListener* or *AlarmListener*.

A further part of this principle is that the domain classes encapsulate the information and behavior related to application logic. The window classes are relatively thin; they are responsible for input and output, and catching GUI events, but do not maintain data or directly provide application functionality.

The motivation for Model-View Separation includes:

- To support cohesive model definitions that focus on the domain processes, rather than on user interfaces.
- To allow separate development of the model and user interface layers.
- To minimize the impact of requirements changes in the interface upon the domain layer.
- To allow new views to be easily connected to an existing domain layer, without affecting the domain layer.
- To allow multiple simultaneous views on the same model object, such as both a tabular and business chart view of sales information.
- To allow execution of the model layer independent of the user interface layer, such as in a message-processing or batch-mode system.
- To allow easy porting of the model layer to another user interface framework.

4. This is a key principle in the pattern *Model-View-Controller* (MVC). MVC was originally a small-scale Smalltalk-80 pattern, and related data objects (models), GUI widgets (views), and mouse and keyboard event handlers (controllers). More recently, the term "MVC" has been coopted by the distributed design community to also apply on a large-scale architectural level. The Model is the Domain Layer, the View is the Presentation Layer, and the Controllers are the workflow objects in the Application layer.

## *Model-View Separation and "Upward" Communication*

How can windows obtain information to display? Usually, it is sufficient for them to send messages to domain objects, querying for information which they then display in widgets—a **polling or pull-from-above** model of display updates.

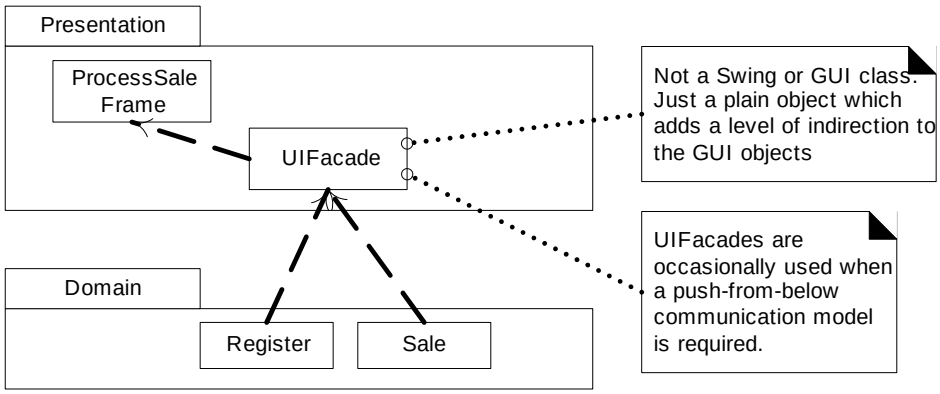



Figure 30.16 A Presentation layer UIFacade is occasionally used for push-from-below designs.

However, a polling model is sometimes insufficient. For example, polling every second across thousands of objects to discover only one or two changes, which are then used to refresh a GUI display, is not efficient. In this case it is more efficient for the few changing domain objects to communicate with windows to cause a display update as the state of domain objects changes. Typical situations of this case include:

- Monitoring applications, such as telecommunications network management.
- Simulation applications which require visualization, such as aerodynamics modeling.

In these situations, a **push-from-below** model of display update is required. Because of the restriction of the Model-View Separation pattern, this leads to the need for "indirect" communication from lower objects up to windows—pushing up notification to update from below.

There are two common solutions:

1. The Observer pattern, via making the GUI object simply appear as an object that implements an interface such as *PropertyListener*.
2. A Presentation facade object. That is, adding a facade within the Presentation layer that receives requests from below. This is an example of adding Indirection to provide Protected Variation if the GUI changes. For example, see Figure 30.16.

## 30.4 Further Readings

There's a wealth of literature on layered architectures, both in print and on the Web. A series of patterns in *Pattern Languages of Program Design,* volume 1, [CS95] first address the topic in pattern form, although layered architectures have been used and written about since at least the 1960s; volume 2 continues with further layers-related patterns. *Pattern-Oriented Software Architecture* volume 1 [BMRSS96] provides a good treatment of the Layers pattern.

Chapter 31

# ORGANIZING THE DESIGN AND IMPLEMENTATION MODEL PACKAGES

*If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?*

*—Seymour Cray*

## Objectives

Organize packages to reduce the impact of changes.

Know alternative UML package structure notation.

## Introduction

If some package X is widely depended upon by the development team, it is undesirable for X to be very unstable (going through many new versions), since it increases the impact on the team in terms of constant version re-synchronization and fixing dependent software that breaks in response to changes in X **(version thrashing).**

This sounds and is obvious, but sometimes a team does not pay attention to identifying and stabilizing the most depended-upon packages, and ends up experiencing more version thrashing than necessary.

This chapter builds on the previous chapter's introduction to layers and packages, by suggesting more fine-grained heuristics for the organization of packages, to reduce these kinds of change impact. The goal is to create a robust physical package design.

One feels the pain of fragile dependency-sensitive package organization much more quickly in C++ than in Java because of the hyper-sensitive compile and link dependencies in C++; a change in one class can have a strong transitive dependency impact leading to recompilation of many classes, and re-linking.[1] Therefore, these suggestions are especially helpful for C++ projects, and moderately so for Java, Smalltalk, or C# (as examples) projects.

The useful work of Robert Martin [Martin95], who has grappled with physical design and packaging of C++ applications, influenced some of the following guidelines.

### *Source Code Physical Design in the Implementation Model*

This issue is an aspect of **physical design**-the UP Implementation Model for source code packaging.

While simply diagramming a package design on a whiteboard or CASE tool, we can arbitrarily place types in any functionally cohesive package without impact. But during source code physical design—the organization of types into physical units of release as Java or C++ "packages"—our choices will influence the degree of developer impact when changes in those packages occur, if there are many developers sharing a common code base.

## 31.1 Package Organization Guidelines

### *Guideline: Package Functionally Cohesive Vertical and Horizontal Slices*

The basic "intuitive" principle is modularization based on functional cohesion—types are grouped together that are strongly related in terms of their participation in a common purpose, service, collaborations, policy, and function. For example, all the types in the NextGen *Pricing* package are related to product pricing. The layers and packages in the NextGen design are organized by functional groups.

In addition to the usually sufficient informal guesswork on grouping by function ("I think class *SalesLineItem* belongs in *Sales")* another clue to functional grouping is a cluster of types with strong internal coupling and weaker extra-cluster coupling. For example, *Register* has a strong coupling to *Sale,* which has a strong coupling to *SalesLineItem.*

1. In C++ the packages may be realized as namespaces, but more likely it means the organization of the source code into separate physical directories—one for each "package."

Internal package coupling, or **relational cohesion,** can be quantified, although such formal analysis is rarely of practical necessity. For the curious, one measure is:

$$RC = \frac{NumberOfInternalRelations}{NumberOfTypes}$$

Where *NumberOfInternalRelations* includes attribute and parameter relations, inheritance, and interface implementations between types in the package.

A package of 6 types with 12 internal relations has RC=2. A package of 6 types with 3 intra-type relations has RC=0.5. Higher numbers suggest more cohesion or relatedness for the package.

Note that this measure is less applicable to packages of mostly interfaces; it is most useful for packages that contain some implementation classes.

A very low RC value suggests either:

- The package contains unrelated things and is not factored well.
- The package contains unrelated things and the designer deliberately does not care. This is common with utility packages of disparate services (e.g., *java.util),* where high or low RC is not important.
- It contains one or more subset clusters with high RC, but overall does not.

## Guideline: Package a Family of Interfaces

Place a family of functionally related *interfaces* in a separate package—separate from implementation classes. This is not primarily for the case of one or two related interfaces, but rather when there is a family of perhaps three or more interfaces. The Java technologies EJB package *javax.ejb* is an example: It is a package of at least twelve interfaces; implementations are in separate packages.

## Guideline: Package by Work and by Clusters of Unstable Classes

The context for this discussion is that packages are usually the basic unit of development work and of release. It is less common to work on and release just one class.

Suppose 1) there is an existing large package P1 with thirty classes, and 2) there is a work trend that a particular subset often classes (Cl through C10) is regularly modified and re-released.

In this case, refactor P1 into Pl-a and Pl-b, where Pl-b contains the ten frequently worked on classes.

Thus, the package has been refactored into more stable and less stable subsets, or more generally, into groups related to work. That is, if most types in a package are worked on together, then it is a useful grouping.

Ideally, fewer developers have a dependency on Pl-b than on Pl-a, and by factoring out this unstable part to a separate package, not as many developers are affected by new releases of Pl-b as by re-releasing the larger original package P1.

Note that this refactoring is in reaction to an emerging work trend. It is difficult to speculatively identify a good package structure in very early iterations. It incrementally evolves over the elaboration iterations, and it should be a goal of the elaboration phase (because it is architecturally significant) to have the majority of the package structure stabilized by elaboration completion.

This guideline illustrates the basic strategy: **Reduce widespread dependency on unstable packages.**

## *Guideline: Most Responsible Are Most Stable*

If the most responsible (depended-on) packages are unstable, there is a greater chance of widespread change dependency impact. As an extreme case, if a widely used utility package such as *com.foo.util* changed frequently, many things could break. Therefore, Figure 31.1 illustrates an appropriate dependency structure.

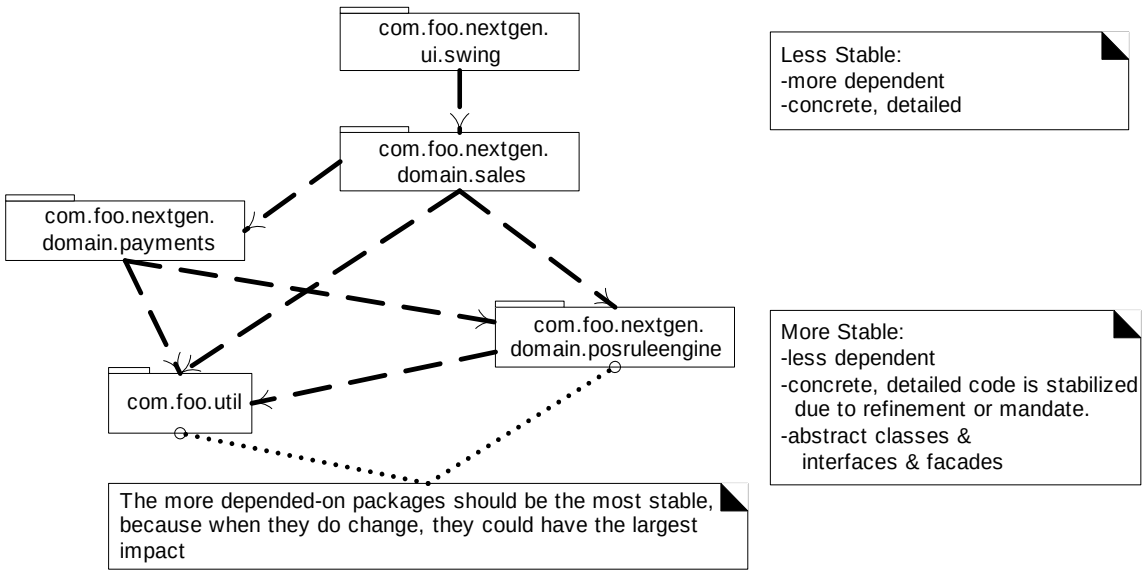



Figure 31.1 More responsible packages should be more stable.

Visually, the lower packages in this diagram should be the most stable. There are different ways to increase stability in a package:

- It contains only or mostly interfaces and abstract classes.
    - For example, *java.sql* contains eight interfaces and six classes, and the classes are mostly simple, stable types such as *Time* and *Date.*
- It has no dependencies on other packages (it is independent), or it depends on other very stable packages, or it encapsulates its dependencies such that dependents are not affected.
    - For example, *com.foo.nextgen.domain.posruleengine* hides its rule engine implementation behind a single facade object. Even if the implementation changes, dependent packages are not affected.
- It contains relatively stable code because it was well-exercised and refined before release.
    - For example, *java.util.*
- It is mandated to have a slow change schedule
    - For example, *java.lang,* the core package in the Java libraries, is simply not allowed to change frequently.

## *Guideline: Factor out Independent Types*

Organize types that can be used independently or in different contexts into separate packages. Without careful consideration, grouping by common functionality may not provide the right level of granularity in the factoring of packages.

For example, suppose that a subsystem for persistence services has been defined in one package *com.foo.seruice.persistence.* In this package are two very general utility/helper classes *JDBCUtililities* and *SQLCommand.* If these are general utilities for working with JDBC (Java's services for relational database access), then they can be used independently of the persistence subsystem, for any occasion when the developer is using JDBC. Therefore, it is better to migrate these types into a separate package, such as *com.foo.util.jdbc.* Figure 31.2 illustrates.

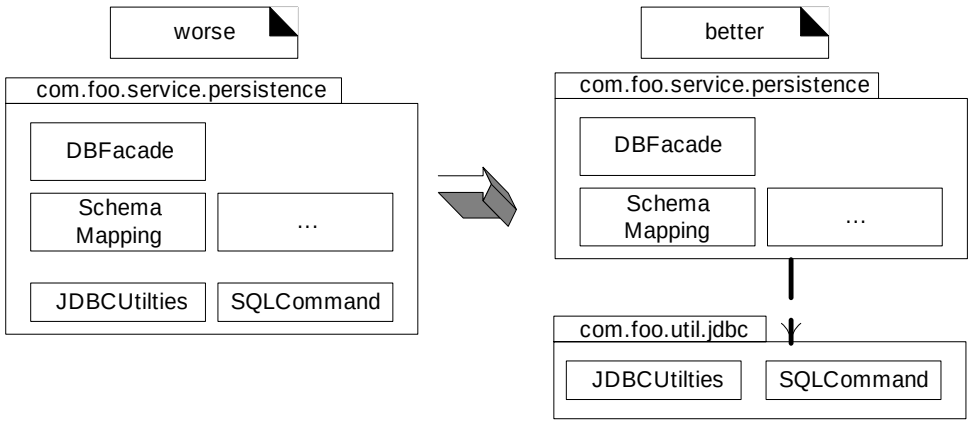



Figure 31.2 Factoring out independent types.

## *Guideline: Use Factories to Reduce Dependency on Concrete Packages*

One way to increase package stability is to reduce its dependency on concrete classes in other packages. Figure 31.3 illustrates the "before" situation.

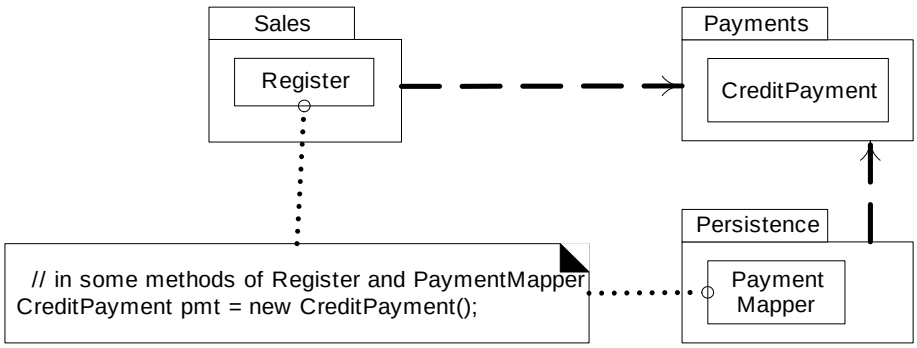



Figure 31.3 Direct coupling to concrete package due to creation.

Suppose that both *Register* and *PaymentMapper* (a class that maps payment objects to/from a relational database) create instances of *CreditPayment* from package *Payments.* One mechanism to increase the long-term stability of the *Sales* and *Persistence* packages is to stop explicitly creating concrete classes defined in other packages *(CreditPayment* in *Payments).*

We can reduce the coupling to this concrete package by using a factory object that creates the instances, but whose create methods return objects declared in terms of interfaces rather than classes. See Figure 31.4.

### Domain Object Factory Pattern

The use of domain object factories with interfaces for the creation of *all* domain objects is a common design idiom. I have seen it mentioned informally in design literature as the Domain Object Factory pattern, but do not know of a reference to it formally written as a pattern.

## *Guideline: No Cycles in Packages*

If a group of packages have cyclic dependency then they may need to be treated as one larger package in terms of a release unit. This is undesirable because releasing larger packages (or package aggregates) increases the likelihood of affecting something.

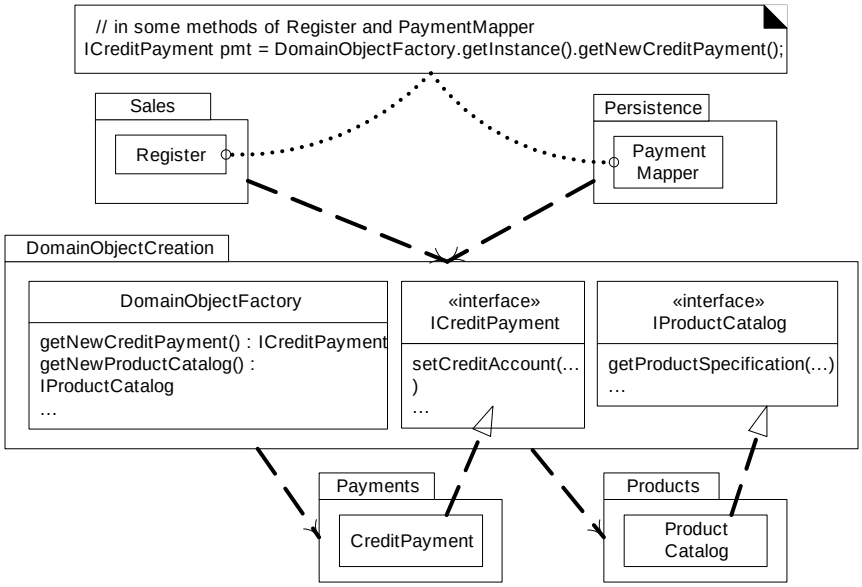



Figure 31.4 Reduced coupling to a concrete package by using a factory object

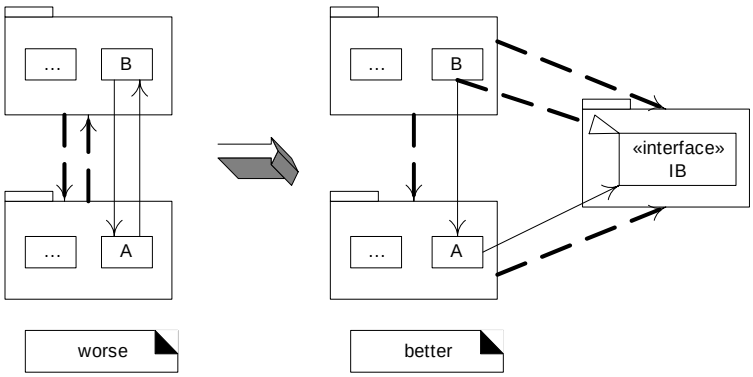



Figure 31.5 Breaking a cyclic dependency.

There are two solutions:

1. Factor out the types participating in the cycle into a new smaller package.
2. Break the cycle with an interface.

The steps to break the cycle with an interface are:

1. Redefine the depended-on classes in one of the packages to implement new interfaces.
2. Define the new interfaces in a new package.
3. Redefine the dependent types to depend on the interfaces in the new pack age, rather than the original classes.

Figure 31.5 illustrates this strategy.

## 31.2 More UML Package Notation

Finally, while on the subject of packages, the UML provides alternate notation to illustrate outer and inner packages. Sometimes it is awkward to draw an outer package box around inner packages. Alternatives are shown in Figure 31.6.

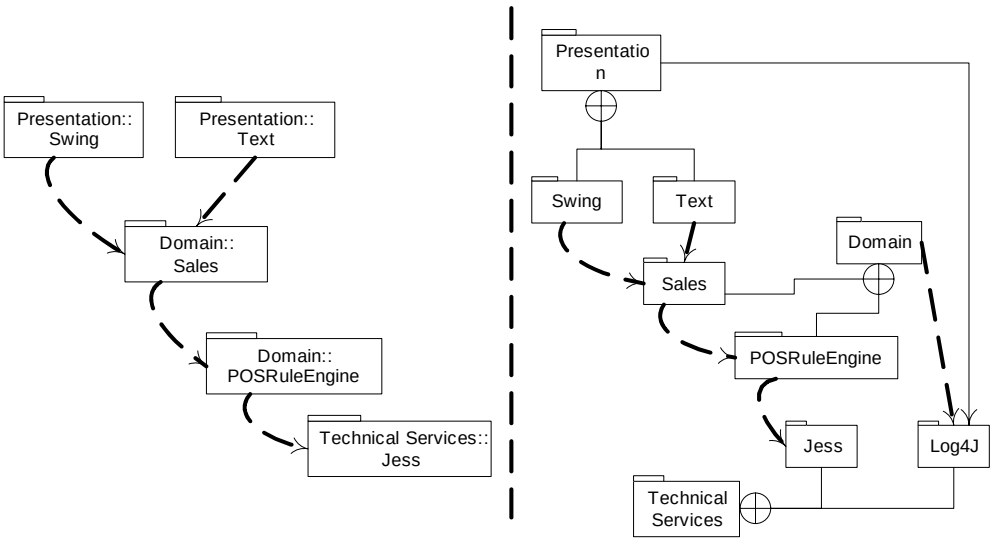



Figure 31.6 Alternate UML approaches to showing packages structure, using UML path names, or the circle-cross symbol.

## 31.3 Further Readings

Most of the detailed work—not surprisingly—on improving package design to reduce dependency impact comes from the C++ community, although the principles apply to other languages. Martin's *Designing Object-Oriented* C++ *Applications Using the Booch Method* [Martin95] provides good coverage, as does *Large-Scale* C++ *Software Design* [Lakos96]. The subject is also introduced in *Java 2 Performance and Idiom Guide* [GL99].

Chapter 32

# INTRODUCTION TO ARCHITECTURAL ANALYSIS AND THE SAD

*Error, no keyboard - press F1 to continue,*

*—early PC BIOS message*

## Objectives

- Create architectural factor tables.
- Create technical memos that record architectural decisions.
- Know basic principles of architectural design.
- Know resources for learning architectural patterns.

## Introduction

The essence of architectural analysis is to identify factors which should influence the architecture, understand their variability and priority, and resolve them. The difficult part is knowing what questions to ask, weighing the trade-offs, and knowing the many ways to resolve an architecturally significant factor, ranging from benign neglect, to fancy designs, to third-party products.

In the UP, the architectural factors are recorded in the Supplementary Specification, and the architectural decisions that resolve them are recorded in the **Software Architecture Document** (SAD, described in more detail near the end of this chapter).

Architectural analysis starts early, during the inception phase, and is a focus of the elaboration phase; it is a high-priority and very influential activity in soft-

ware development. The topic was deferred until this point of the book so that fundamentals of OOA/D could be first presented. It is a useful activity to:

- reduce the risk of missing something centrally important in the design of the systems
- avoid applying excessive effort to low priority issues help
- align the product with business goals

This chapter is an introduction to basic steps and ideas in architectural analysis from a UP perspective; that is, to the method, rather than to tips and tricks of master architects. Thus, it is not a cookbook of architectural solutions—a large and context-dependent subject that is beyond the scope of this introductory book. Nevertheless, the NextGen POS case study comments in the chapter do provide concrete examples of architectural solutions.

## 32.1 Architectural Analysis

**Architectural analysis** is concerned with the identification and resolution of the system's non-functional (for example, quality) requirements, in the context of the functional requirements.

In the UP, the term encompasses both architectural investigation (identification) and architectural design (resolution). Here are some examples of the many issues to be identified and resolved at an architectural level:

- How do reliability and fault-tolerance requirements affect the design?
  - For example, in the NextGen POS, for what remote services (e.g., tax calculator) will fail-over to local services be allowed? Why? Do they provide exactly the same services locally as remotely, or are there differences?
- How do the licensing costs of purchased subcomponents affect profitability?
  - For example, the producer of the excellent database server, *Clueless*, wants 2% of each NextGen POS sale, if their product is used as a subcomponent. Using their product will speed development (and time to market) because it is robust and provides many services, and many developers know it, but at a price. Should the team instead use the less robust, open source *YourSQL* database server? At what risk? How does it restrict the ability to charge for the NextGen product?
- How does distribution of services affect the quality requirements and functional requirements?
  - For example, using a remote (single, centralized) tax calculator reduces the footprint of each NextGen client, reduces licensing fees (only one copy is needed), and minimizes the custom configu-

ration effort (each installation requires weekly adjustment due to changing government and business policies). However, the remote service reduces response time sufficiently that taxes can only be calculated once, after all line items have been entered; one cannot see a running total with taxes after each line item entry; and the remote call takes too long. It also creates a single point of failure.

- How do the adaptability and configurability requirements affect the design?
    - For example, most retailers have variations in business rules they want represented in their POS applications. What are the variations? What is the "best" way to design for them? What is the criteria for best? Can NextGen make more money by requiring customized programming for each customer (and how much effort will that be?), or with a solution that allows the customer to add the customization easily themselves? Should "more money" be the goal in the short-run?

## *Common Steps in Architectural Analysis*

There are several methods of architectural analysis. Common to most of these is some variation of the following steps:

1. Identify and analyze the non-functional requirements that have an impact on the architecture. Functional requirements are also relevant (especially in terms of variability or change), but the non-functional are given thorough attention. In general, all these may be called **architectural factors** (also known as the **architectural drivers).**
    - This step could be characterized as regular requirements analysis, but since it is done in the context of identifying architectural impact and deciding high-level architectural solutions, it is considered a part of architectural analysis in the UP.
    - In terms of the UP, some of these requirements will be roughly identified and recorded in the Supplementary Specification or use cases during inception. During architectural analysis, which occurs in early elaboration, the team investigates these requirements more closely.
2. For those requirements with a significant architectural impact, analyze alternatives and create solutions that resolve the impact. These are **architectural decisions.**
    - Decisions range from "remove the requirement," to a custom solution, to "stop the project," to "hire an expert."

This presentation introduces these basic steps in the context of the NextGen POS case study. For simplicity, it avoids architectural deployment issues such as the hardware and operating system configuration, which are very context and time sensitive.

## 32.2 Types and Views of Architecture

Some descriptions of architecture define different types, such as the "application architecture" (allocation of features to components) or "system architecture" (hardware and operating system configuration).

In the UP, there is a similar specialization of information, but these are described in "views" of the architecture, which summarize and emphasize a particular perspective. For example, the **logical view** of the architecture, which was introduced in Chapter 30, summarizes the organization and functionality of the major software elements (such as the layers)—it is similar to the term application architecture. The **deployment view** summarizes the system topology, communications, and mapping of executable elements to processing nodes—it is similar to the term system architecture.

The UP defines six views of the architecture, which are described in detail near the end of this chapter. Concretely, the views combine text and diagrams, and—if described at all—are recorded in the SAD.

Architectural analysis is related to the architectural views because the architectural decisions are reflected in, and described in, one or more architectural views.

## 32.3 The Science: Identification and Analysis of Architectural Factors

### *Architectural Factors*

Any and all of the FURPS+ requirements may have a significant influence on the architecture of a system, ranging from reliability, to schedule, to skills, and to cost constraints. For example, a case of tight schedule with limited skills and sufficient money probably favors buying or outsourcing to specialists, rather than building all components in-house.

However, the factors with the strongest architectural influence tend to be within the high-level FURPS+ categories of functionality, reliability, performance, sup-portability, implementation, and interface (see Chapter 5 for a detailed breakdown). Interestingly, it is usually the non-functional quality attributes (such as reliability or performance) that give a particular architecture its unique flavor, rather than its functional requirements. For example, the design in the Next-Gen system to support different third-party components with unique interfaces, and the design to support easily plugging in different sets of business rules.

In the UP, these factors with architectural implications are called **architecturally significant requirements.** "Factors" is used here for brevity.

Many technical and organizational factors can be characterized as *constraints* that restrict the solution in some way (such as, must run on Linux, or, the budget for purchasing third-party components is X).

## Quality Scenarios

When defining quality requirements during architectural factor analysis, **quality scenarios**[1] are recommended, as they define measurable (or at least observable) responses, and thus can be verified. It is not much use to vaguely state "the system will be easy to modify" without some measure of what that means.

Quantifying some things, such as performance goals and mean time between failure, are well known practices, but quality scenarios extend this idea and encourages recording all (or at least, most) factors as measurable statements.

Quality scenarios are short statements of the form <stimulus> <measurable response>; for example:

- When the completed sale is sent to the remote tax calculator to add the taxes, the result is returned within 2 seconds "most" of the time, measured in a production environment under "average" load conditions.
- When a bug report arrives from a NextGen beta test volunteer, reply with a phone call within 1 working day.

Note that "most" and "average" will need further investigation and definition by the NextGen architect; a quality scenario is not really valid until it is testable, which implies fully specified. Also, observe the qualification in the first quality scenario in terms of the environment to which it applies. It does little good to specify a quality scenario, verify that it passes in a lightly loaded development environment, but fail to evaluate it in a realistic production environment.

### Pick Your Battles

A caution: Writing these quality scenarios can be a mirage of usefulness. It's easy to *write* these detailed specifications, but not to realize them. Will anyone ever really test them? How and by whom? A strong dose of realism is required when writing these; there's no point in listing many sophisticated goals if no one will ever really follow through on testing them.

There is a relationship here to the "pick your battles" discussion that was presented in an earlier chapter on the Protected Variations pattern. What are the really critical make-or-break quality scenarios? For example, in an airline reservation system, consistently fast transaction completion under very high load conditions is truly critical to the success of the system—it must definitely be

1. A term used in various architectural methods promoted by the Software Engineering Institute (SEI); for example, in *the Architecture Based Design* method.

tested. In the NextGen system, the application really must be fault-tolerant and fail over to local replicated services when the remote ones fail—it must definitely be properly tested and validated. Therefore, focus on writing quality scenarios for the important battles, and follow through with a plan for their evaluation.

## Describing Factors

One important goal of architectural analysis is to understand the influence of the factors, their priorities, and their variability (immediate need for flexibility and future evolution). Therefore, most architectural methods (for example, see [HNS00]) advocate creating a table or tree with variations of the following information (the format varies depending on the method). The following style shown in Table 32.1 is called a **factor table,** which in the UP is part of the Supplementary Specification.

| Factor | Measures and quality scenarios | Variability (current flexibility and future evolution) | Impact of factor (and its variability) on stakeholders, architecture and other factors | Priority for Success | Difficulty or Risk |
|---|---|---|---|---|---|
| Reliability — Recoverability | | | | | |
| Recovery from remote service failure | When a remote service fails, reestablish connectivity with it within 1 minute of its detected re-availability, under normal store load in a production environment. | current flexibility - our SME says local client-side simplified services are acceptable (and desirable) until reconnection is possible.<br>evolution - within 2 years, some retailers may be willing to pay for full local replication of remote services (such as the tax calculator).<br>Probability? High. | High impact on the large-scale design.<br>Retailers really dislike it when remote services fail, as it prevents or restricts them from using a POS to make sales. | H | M |
| | | | | | |

Table 32.1 Sample factor table. Legend: H-high. M-medium. SME-subject matter expert.

Notice the categorization scheme: *Reliability—Recoverability* (from the FURPS+ categories). This isn't presented as the best or only scheme, but it is useful to group architectural factors into categories. For example, certain categories (such as reliability and performance) strongly relate to identifying and defining test plans, and thus it is useful to group them.

The basic priority and risk code values of H/M/L are simply suggestive of using some codes the team finds useful; there are a variety of coding schemes (numeric and qualitative) from different architectural methods and standards (such as ISO 9126). A caution: If the extra effort of using a more complex scheme does not lead to any practical action, it isn't worthwhile.

### *Factors and UP Artifacts*

The central functional requirements repository in the UP are the use cases, and they, along with the Vision and Supplementary Specification, are an important source of inspiration when creating a factor table. In the use cases, the *Special Requirements*, *Technology Variations,* and *Open Issues* should be reviewed, and their implied or explicit architectural factors consolidated in the Supplementary Specification.

It is reasonable to at first record use-case related factors with the use case during its creation, because of the obvious relationship, but it is ultimately more convenient (in terms of content management, tracking, and readability) to consolidate all the architectural factors in one location—in the factor table in the Supplementary Specification.

**Use Case UC1: Process Sale**

**Main Success Scenario:**
1. ...
**Special Requirements:**
- Credit authorization response within 30 seconds 90% of the time.
- Somehow, we want robust recovery when access to remote services such the inven tory system is failing.

**Technology and Data Variations List:**
2a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.

**Open Issues:**
- What are the tax law variations?
- Explore the remote service recovery issue.

## 32.4 Example: Partial NextGen POS Architectural Factor Table

The partial factor table in Table 32.2 shows some factors related to later discussion.

| Factor | Measures and quality scenarios | Variability (current flexibility and future evolution) | Impact of factor (and its variability) on stakeholders, architecture and other factors | Priority for Success | Difficulty or Risk |
|---|---|---|---|---|---|
| Reliability — Rec<br>Recovery from remote service failure | coverability<br>When a remote service fails, reestablish connectivity with it within 1 minute of its detected re-availability, under normal store load in a production environment. | current flexibility - our SME says local client-side simplified services are acceptable (and desirable) until reconnection is possible.<br>evolution - within 2 years, some retailers may be willing to pay for full local replication of remote services (such as the tax calculator).<br>Probability? High. | High impact on the large-scale design.<br>Retailers really dislike it when remote services fail, as it prevents them from using a POS to make sales. | H | M |
| Recovery from remote product database failure | as above | current flexibility - our SME says local client-side use of cached "most common" product info is acceptable (and desirable) until reconnection is possible.<br>evolution - within 3 years, client-side mass storage and replication solutions will be cheap and effective, allowing permanent complete replication and thus local usage. Probability? High. | as above | H | M |
| Supportability - Adaptability | | | | | |
| Support many third-party services (tax calculator, inventory, HR, accounting). They will vary at each installation. | When a new third-party system must be integrated, it can be, and within 10 person days of effort. | current flexibility - as described by factor<br>evolution - none | Required for product acceptance. Small impact on design. | H | L |
| Support wireless PDA terminals for the POS client? | When support is added, it does not require a change to the design of the non-UI layers of the architecture. | current flexibility - not required at present<br>evolution - within 3 years, we think the probability is very high that wireless "PDA" POS clients will be desired by the market. | High design impact in terms of protected variation from many elements. For example, the operating systems and UIs are different on small devices. | L | H |
| Other - Legal | | | | | |
| Current tax rules must be applied. | When the auditor evaluates conform-ance, 100% con-formance will be found.<br>When tax rules change, they will be operational within the period allowed by government. | current flexibility - conformance is inflexible, but tax rules can change almost weekly because of the many rules and levels of government taxation (national, state, ...)<br>evolution - none | Failure to comply is a criminal offense.<br>Impacts tax calculation services.<br>Difficult to write our own service-complex rules, constant change, need to track all levels of government.<br>But, easy/low risk if buy a package. | H | L |

Table 32.2 Partial factor table for the NextGen architectural analysis.

## 32.5 The Art: Resolution of Architectural Factors

One could say the *science* of architecture is the collection and organization of information about the architectural factors, as in the factor table. The *art* of architecture is making skillful choices to resolve these factors, in light of trade-offs, interdependencies, and priorities.

Adept architects have knowledge in a variety of areas (for example, architectural styles and patterns, technologies, products, pitfalls, and trends) and apply this to their decisions.

### *Recording Architectural Alternatives, Decisions, and Motivation*

Ignoring for now principles of architectural decision-making, virtually all architectural methods recommend keeping a record of alternative solutions, decisions, influential factors, and motivations for the noteworthy issues and decisions.

Such records have been called **technical memos** [Cunningham96J, issue cards [HNS00], and **architectural approach documents** (SEI architectural proposals), with varying degrees of formality and sophistication. In some methods, these memos are the basis for yet another step of review and refinement.

In the UP, the memos should be recorded in the SAD.

An important aspect of the technical memo is the *motivation* or rationale. When a future developer or architect needs to modify the system,[2] it is immensely helpful to understand the motivations behind the design, such as *why* a particular approach to recovery from remote service failure in the NextGen POS was chosen and others rejected, in order to make informed decisions about changing the system.

Explaining the rationale of rejecting the alternatives is important, as during future product evolution, an architect may reconsider these alternatives, or at least want to know what alternatives were considered, and why one was chosen.

A sample technical memo follows that records an architectural decision for the NextGen POS. The exact format is, of course, not important. Keep it simple and just record information that will help the future reader make an informed decision when changing the system.

2. Or when four weeks have passed and the original architect has forgotten their own rationale!

**Technical Memo Issue:**
**Reliability—Recovery from Remote Service Failure**

**Solution Summary: Location transparency using service lookup, failover from remote to local, and local service partial replication.**

**Factors**

- Robust recovery from remote service failure (e.g., tax calculator, inventory)
- Robust recovery from remote product (e.g., descriptions and prices) database failure

**Solution**

Achieve protected variation with respect to location of services using an Adapter created in a Services-Factory. Where possible, offer local implementations of remote services, usually with simplified or constrained behavior. For example, the local tax calculator will use constant tax rates. The local product information database will be a small cache of the most common products. Inventory updates will be stored and forwarded at reconnection.

See also the *Adaptability—Third-Party Services* technical memo for the adaptability aspects of this solutions, because remote service implementations will vary at each installation.

To satisfy the quality scenarios of reconnection with the remote services ASAP, use smart Proxy objects for the services, that on each service call test for remote service reactivation, and redirect to them when possible.

**Motivation**

Retailers really don't want to stop making sales! Therefore, if the NextGen POS offers this level of reliability and recovery, it will be a very attractive product, as none of our competitors provide this capability. The small product cache is motivated by very limited client-side resources. The real third-party tax calculator is not replicated on the client primarily because of the higher licensing costs, and configuration efforts (as each calculator installation requires almost weekly adjustments). This design also supports the evolution point of future customers willing and able to permanently replicate services such as the tax calculator to each client terminal.

**Unresolved Issues**

none

**Alternatives Considered**

A "gold level" quality of service agreement with remote credit authorization services to improve reliability. It was available, but much too expensive.

---

Note as illustrated in this example—and this is a key point—that an architectural decision described in one technical memo may resolve a group of factors, not only one.

## *Priorities*

There is a hierarchy of goals that guides architectural decisions:

1. Inflexible constraints, including safety and legal compliance,
   - The NextGen POS must correctly apply tax policies.

2. Business goals.
    - Demo of noteworthy features ready for the POSWorld trade show in Hamburg in 18 months.
    - Has qualities and features attractive to department stores in Europe (for example, multi-currency support and customizable business rules).
3. All other goals
    - These can often be traced back to directly stated business goals, but are indirect. For example, "easily extendible: can add <some unit of functionality> in 10 person weeks" could trace to a business goal of "new release every six months."

In the UP, many of these goals are recorded in the Vision artifact. Mind that the *Priority for Success* scores in the factor table should reflect the priority of these goals.

There is a distinguishing aspect of decision-making at this level vs. small-scale object design: one has to simultaneously consider more (and often globally influential) goals and their trade-offs. Furthermore, the business goals become central to the technical decisions (or at least they should). For example:

---

**Technical Memo Issue: Legal—Tax Rule Compliance**

**Solution Summary: Purchase a tax calculator component.**

**Factors**

- Current tax rules must be applied, by law.

**Solution**

Purchase a tax calculator with a licensing agreement to receive ongoing tax rule updates. Note that different calculators may be used at different installations.

**Motivation**

Time-to-market, correctness, low maintenance requirements, and happy developers (see alternatives). These products are costly, which affects our cost-containment and product pricing business goals, but the alternative is considered unacceptable.

**Unresolved Issues**

What are the leading products and their qualities?

**Alternatives Considered**

Build one by the NextGen team? It is estimated to take too long, be error prone, and create an ongoing costly and uninteresting (to the company's developers) maintenance responsibility, which affects the goal of "happy developers" (surely, the most important goal of all).

---

### Priorities and Evolution Points: Under- and Over-engineering

Another distinguishing feature of architectural decision-making is prioritization by probability of **evolution points**—points of variability or change that *may* arise in the future. For example, in NextGen, there is a chance that wireless handheld client terminals will become desirable. Designing for this has a significant impact because of differences in operating systems, user interface, hardware resources, and so forth.

The company could spend a huge amount of money (and increase a variety of risks) to achieve this "future proofing." If it turns out in the future that this was not relevant, doing it would be a very expensive exercise in over-engineering. Note also that future proofing is arguably rarely perfect, since it is speculation; even if the predicted change occurs, some change in the speculated design is likely.

On the other hand, future proofing against the Y2K date problem would have been money very well spent; instead, there was under-engineering with a wickedly expensive result.

> The art of the architect is knowing what battles are worth fighting—where it's worth investing in designs that provide protection against evolutionary change.

To decide if early "future-proofing" should be avoided, realistically consider the scenario of deferring the change to the future, when it is called for. How much of the design and code will actually have to change? What will be the effort? Perhaps a close look at the potential change will reveal that what was at first considered a gigantic issue to protect against, is estimated to consume only a few person-weeks of effort.

This is just a hard problem; "Prediction is very difficult, especially if it's about the future" (unverifiably attributed to Niels Bohr).

## *Basic Architectural Design Principles*

The core design principles explored in much of this book that were applicable to small-scale object design are still dominant principles at the large-scale architectural level:

- low coupling
- high cohesion
- protected variation (interfaces, indirection, service lookup, and so forth)

However, the granularity of the components is larger—it is low coupling between applications, subsystems, or process rather than between small objects.

Furthermore, at this larger scale, there are more or different mechanisms to achieve qualities such as low coupling and protected variation. For example, consider this technical memo:

**Technical Memo Issue: Adaptability—Third-Party Services**

**Solution Summary: Protected Variation using interfaces and Adapters Factors**

- Support many and changeable third-party services (tax calculators, credit authorization, inventory, ...)

**Solution**

Achieve protected variation as follows: Analyze several commercial tax calculator products (and so forth for the other product categories) and construct common interfaces for the lowest common denominators of functionality. Then use Indirection via the Adapter pattern. That is, create a resource Adapter object that implements the interface and acts as connection and translator to a particular back-end tax calculator.

See also the *Reliability—Recovery from Remote Service Failure* technical memo for the location transparency aspects of this solution.

**Motivation**

Simple. Cheaper, and faster communication than using a messaging service (see alternatives), and in any event a messaging service can't be used to directly connect to the external credit authorization service.

**Unresolved Issues**

Will the lowest common denominator interfaces create an unforeseen problem, such as too limited? **Alternatives Considered**

Apply indirection by using a messaging or publish subscribe service (e.g., a JMS implementation) between the client and tax calculator, with adapters. But not directly usable with a credit authorizes costly (for reliable ones), and more reliability in message delivery than is practically needed.

The point is that at the architectural level, there are usually new mechanisms to achieve protected variation (and other goals), often in collaboration with third-party components, such as using a Java Messaging Service (JMS) or EBJ server.

## Separation of Concerns and Localization of Impact

Another basic principle applied during architectural analysis is to achieve a **separation of concerns.** It is also applicable at the scale of small objects, but achieves prominence during architectural analysis.

**Cross-cutting concerns** are those with a wide application or influence in the system, such as data persistence or security. One *could* design persistence support in the NextGen application such that each object (that contained application logic code) itself also communicated with a database to save its data. This

would weave the concern of persistence in with the concern of application logic, in the source code of the classes—so too with security. Cohesion drops and coupling rises.

In contrast, designing for a separation of concerns factors out persistence support and security support into separate "things" (there are very different mechanisms for this separation). An object with application logic just has application logic, not persistence or security logic. Similarly, a persistence subsystem focuses on the concern of persistence, not security. A security subsystem doesn't do persistence.

Separation of concerns is a large-scale way of thinking about low coupling and high cohesion at an architectural level. It also applies to small-scale objects, because its absence results in in-cohesive objects that have multiple areas of responsibility. But it is especially an architectural issue because the concerns are broad, and the solutions involve major, fundamental design choices.

There are at least three large-scale techniques to achieve a separation of concerns:

1. Modularize the concern into a separate component (for example, subsystem) and invoke its services.
    - This is the most common approach. For example, in the NextGen system, the persistence support could be factored into a subsystem called the *persistence service.* Via a facade, it can offer a public interface of services to other components. Layered architectures also illustrate this separation of concerns.
2. Use decorators.
    - This is the second most common approach; first popularized in the Microsoft Transaction Service, and afterwards with EJB servers. In this approach, the concern (such as security) is decorated onto other objects with a Decorator object that wraps the inner object and interposes the service. The Decorator is called a **container** in EJB terminology. For example, in the NextGen POS system, security control to remote services such as the HR system can be achieved with an EJB container that adds security checks in the outer Decorator, around the application logic of the inner object.
3. Use post-compilers and aspect-oriented technologies.
    - For example, with EJB entity beans one can add persistence support to classes such as *Sale.* One specifies in a property descriptor file the persistence characteristics of the *Sale* class. Then, a post-compiler (by which I mean another compiler that executes after the "regular" compiler) will add the necessary persistence support in a modified *Sale* class (modifying just the bytecode) or subclass. The developer continues to see the original class as a "clean" appli-cation-logic-only class. Another variation is **aspect-oriented** technologies such as AspectJ (www.aspectj.org), which similarly

support post-compilation weaving in of cross-cutting concerns into the code, in a manner that is transparent to the developer. These approaches maintain the illusion of separation during development work, and weave in the concern before execution.

### *Promotion of Architectural Patterns*

An exploration of architectural patterns and how they could apply (or misapply) to the NextGen case study is out of scope in this introductory text. However, a few pointers:

Probably the most common mechanism to achieve low coupling, protected variation, and a separation of concerns at the architectural level is the Layers pattern, which has been introduced a previous chapter. This is an example of the most common separation technique—modularizing concerns into separate components or layers.

There is a large and growing body of written architectural patterns. Studying these is the fastest way I know of to learn architectural solutions. Please see the recommended readings.

## 32.6 Summary of Themes in Architectural Analysis

One theme to note is that "architectural" concerns are especially related to nonfunctional requirements, and include an awareness of the business or market context of the application. At the same time, the functional requirements (for example, processing sales) cannot be ignored; they provide the context within which these concerns must be resolved. Further, identification of their variability is architecturally significant.

A second theme is that architectural concerns involve system-level, large-scale, and broad problems whose resolution usually involves large-scale or fundamental design decisions; for example, the choice of—or even use of—an application server.

A third theme in architectural analysis is interdependencies and trade-offs. For example, improved security may affect performance or usability, and most choices affect cost.

A fourth theme in architecture analysis is the generation and evaluation of alternative solutions. A skilled architect can offer design solutions that involve building new software, and also suggest solutions (or partial solutions) using commercial or publicly available software and hardware. For example, recovery in a remote server of the NextGen POS can be achieved through designing and programming "watchdog" processes, or perhaps through clustering, replication, and failover services offered by some operating system and hardware components. Good architects know third-party hardware and software products.

The opening definition of architectural concerns provides the framework for how to think about the subject of architecture: identifying the issues with large-scale or system-level implications, and resolving them.

> Architectural analysis is concerned with the identification and resolution of the system's non-functional (e.g., quality) requirements, in the context of the functional requirements.

## 32.7 Architectural Analysis within the

## UP *Caution: Waterfall Architectural*

### *Analysis*

Architectural analysis methods and books often implicitly encourage waterfall-style extensive architectural design decisions before implementation. In iterative development and UP, apply these ideas in the context of small steps, feedback, and adaptation, rather than attempting to fully resolve the architecture before programming. Tackle implementation of the riskiest or most difficult solutions in early iterations, and adjust the architectural solutions based on feedback and growing insight.

### *Architectural Information in the UP Artifacts*

- The architectural factors (for example, in a factor table) are recorded in the Supplementary Specification.
- The architectural decisions are recorded in the SAD. This includes the technical memos and descriptions of the architectural views.

### *The SAD and Its Architectural Views*

In addition to the UML package, class, and interaction diagrams, another key artifact in the UP Design Model is the SAD. It describes the big ideas in the architecture, including the decisions of architectural analysis. Practically, it is a *learning aid* for developers who need to understand the essential ideas of the system.

When someone joins the team, a project coach can say, "Welcome to the NextGen project! Please go to the project website and read the ten page SAD in order to get an introduction to the major ideas." During a later release, when new people work on the system, the SAD is a key learning aid.

Therefore, it should be written with this audience and goal in mind: What do I need to say (and draw in the UML) that will quickly help someone understand the major ideas in this system?

The essence of the SAD is a summary of the architectural decisions (such as with technical memos) and the UP architectural views.

## Architectural Views in the SAD

Having an architecture is one thing; describing it is something else.

In [Kruchten95], the influential idea of describing an architecture with multiple views was promoted. The essential idea of an **architectural view** is this:

*Architectural View*

A view of the system architecture from a given perspective; focuses primarily on structure, modularity, essential components, and the main control flows. [RUP].

An important aspect of the view missing from this RUP definition is the *motivation.* That is, an architectural view should explain why the architecture is the way it is.

An architectural view is a window onto the system from a particular perspective that emphasizes the key noteworthy information or ideas, and ignores the rest.

An architectural view is a tool of communication, education, or thought; it is expressed in text and UML diagrams.

In the UP, six views of the architecture are suggested (more are allowed, such as a security view).[3] All are optional, but documenting at least the logical, process, use case, and deployment views is recommended. The six views are:

1. **Logical**
   - o Conceptual organization of the software in terms of the most important layers, subsystems, packages, frameworks, classes, and interfaces. Also summarizes the functionality of the major software elements, such as each subsystem.
   - o Shows outstanding use-case realization scenarios (as interaction diagrams) that illustrate key aspects of the system.
   - o A view onto the UP Design Model, visualized with UML package, class, and interaction diagrams.

3. Early versions of the UP described the "4+1" views as defined in [Kruchten95], which evolved into the six views.

**2. Process**

- o Processes and threads. Their responsibilities, collaborations, and the allocation of logical elements (layers, subsystems, classes, ...) to them.
- o A view onto the UP Design Model, visualized with UML class and interaction diagrams, using the UML process and thread notation.

**3. Deployment**

- o Physical deployment of processes and components to processing nodes, and the physical network configuration between nodes.
- o A view onto the UP Deployment Model, visualized with UML deployment diagrams. Normally, the "view" is simply the entire model rather than a subset, as all of it is noteworthy. See Chapter 38 for the UML deployment diagram notation.

**4. Data**

- o Overview of the persistent data schema, the schema mapping from objects to persistent data (usually in a relational database), the mechanism of mapping from objects to a database, database stored procedures and triggers.
- o A view onto the UP Data Model, visualized with UML class diagrams used to describe a data model.

**5. Use case**

- o Summary of the most architecturally significant use cases and their non-functional requirements. That is, those use cases that, by their implementation, illustrate significant architectural coverage or that exercise many architectural elements. For example, the *Process Sale* use case, when fully implemented, has these qualities.
- o A view onto the UP Use-Case Model, expressed in text and visualized with UML use case diagrams.

**6. Implementation**

- o First, a definition of the Implementation Model: In contrast to the other UP models, which are text and diagrams, this "model" *is* the actual source code, executables, and so forth. It has two parts: 1) deliverables, and 2) things that create deliverables (such as source code and graphics). The Implementation Model is all of this stuff, including web pages, DLLs, executables, source code, and so forth, and their organization—such as source code in Java packages, and bytecode organized into JAR files.
- o The implementation view is a summary description of the noteworthy organization of deliverables and the things that create deliverables (such as the source code).

- A view onto the UP Implementation Model, expressed in text and visualized with UML package and component diagrams.

For example, the NextGen package and interaction diagrams shown in Chapter 30 on layering and logical architecture show the big ideas of the logical structure of the software architecture. In the SAD, the architect will create a section called *Logical View*, insert those UML diagrams, and add some written commentary on what each package and layer is for, and the motivation behind the logical design. Likewise with the process and deployment views.

A key idea of the architectural views—which concretely are text and diagrams—is that they do *not* describe *all* of the system from some perspective, but only outstanding ideas from that perspective. A view is, if you will, the "one minute elevator" description: What are the most important things you would say in one minute in an elevator to a colleague on this perspective?

Architectural views may be created:

- after the system is built, as a summary and learning aid for future developers
- at the end of certain iteration milestones (such as the end of elaboration) to serve as a learning aid for the current development team, and new members
- speculatively, during early iterations, as an aid in creative design work, recognizing that the original view will change as design and implementation proceeds

## Sample Structure of a SAD

**Software Architecture Document Architectural**

**Representation**

*(Summary of how the architecture will be described in this document, such as using by technical memos and the architectural views. This is useful for someone unfamiliar with the idea of technical memos or views. Note that not all views are necessary.)*

**Architectural Factors and Decisions**

*(Reference to the Supplementary Specification to view the Factor Table. Also, the set of technical memos the summarize the decisions.)*

**Logical View**

*(UML package diagrams, and class diagrams of major elements. Commentary on the large scale structure and functionality of major components.)*

**Process View**

*(UML class and interaction diagrams illustrating the processes and threads of the system. Group this by threads and processes that interact. Comment on how the interprocess communication works (e.g., by Java RMI).*

#### Use-Case View

*(Brief summary of the most architecturally significant use cases. UML interaction diagrams for some architectural significant use-case realizations, or scenarios, with commentary on the diagrams explaining how they illustrate the major architectural elements.)*

#### Deployment View

*(UML deployment diagrams showing the nodes and allocation of processes and components. Commentary on the networking.)*

## *Phases*

**Inception**—If it is unclear if it is technically possible to satisfy the architecturally significant requirements, the team may implement an **architectural proof-of-concept** (POC) to determine feasibility. In the UP, its creation and assessment is called **Architectural Synthesis.** This is distinct from plain old small POC programming experiments for isolated technical questions. An architectural POC lightly covers *many* of the architecturally significant requirements to assess their *combined* feasibility.

**Elaboration**—A major goal of this phase is to implement the core risky architectural elements, thus most architectural analysis is completed during elaboration. It is normally expected that the majority of factor table, technical memo, and SAD content can be completed by the end of elaboration.

**Transition**—Although ideally the architecturally significant factors and decisions were resolved long before transition, the SAD will need a review and possible revision at the end of this phase to ensure it accurately describes the final deployed system.

**Subsequent evolution cycles**—Before the design of new versions, it is common to revisit architectural factors and decisions. For example, the decision in version 1.0 to create a single remote tax calculator service, rather than one duplicated on each POS node, could have been motivated by cost (to avoid multiple licenses). But perhaps in the future the cost of tax calculators is reduced, and thus, for fault tolerance or performance reasons, the architecture is changed to use multiple local tax calculators.

## 32.8 Further Readings

There is a growing body of architecture-related patterns, and general software architecture advice. Suggestions:

- *Pattern-Oriented Software Architecture*, both volumes.
- *Software Architecture in Practice* [BCK98].
- *Pattern Languages of Program Design*, all volumes. Each volume has a sec
tion on architecture-related patterns.
- Online Web articles on architectural patterns (such as J2EE architectures),
available at Sun, IBM, and other websites.
- Online Web articles on architecture available at the Carnegie Mellon Uni
versity Software Engineering Institute (SEI), which has long been a center
of architecture investigation (www.sei.cmu.edu).