Larman Applying UML and Patterns, 3rd Ed.

Extracts from Chapter 13: "Logical Architecture and UML Package Diagrams" (pp 197–204)

Introduction

The design of a typical OO system is based on several architectural layers, such as the UI layer, an application logic (or "domain") layer and so forth. This chapter briefly explores a logical layered architecture and the related UML notation.

13.2 What is the Logical Architecture? And Layers?

The **logical architecture** is the large-scale organization of the software classes into packages (or namespaces), subsystems, and layers. It's called the *logical* architecture because there's no decision about how these elements are deployed across different operating system processes or across physical computers in a network (these latter decisions are part of the **deployment architecture**).

A **layer** is a very coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system. Also, layers are organized such that "higher" layers (such as the UI layer) call upon services of "lower" layers, but not normally vice versa. Typically layers in an OO system Include:

- **User Interface**.
- **Application Logic and Domain Objects**—software objects representing domain concepts (for example, a software class *Sale*) that fulfil application requirements, such as calculating a sale total.
- **Technical Services—**general purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging. These services are usually application-independent and reusable across several systems.

In a **strict layered architecture**, a layer only calls upon the services of the layer directly below it. This design is common in network protocol stacks, but not in information systems, which usually have a **relaxed layered architecture**, in which a higher layer calls upon several lower layers. For example, the UI layer may call upon its directly subordinate application logic layer, and also upon elements of a lower technical service layer, for logging and so forth.

A logical architecture doesn't have to be organized in layers. But it's very common, and hence, introduced at this time.

13.4 What is Software Architecture?

I touched on the logical and deployment architectures, so now is a good time to introduce a definition for **software architecture**. Here's one:

> 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 behaviour as specified in the collaborations among those elements, the composition of these structural and behavioural elements into progressively larger subsystems, and the architectural style that guides this organization—these elements and their interfaces, their collaborations, and their composition. [BRJ99=Booch, Rumbaugh, Jacobson, 1999, "The UML User Guide"]

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

13.5 Applying UML: Package Diagrams

UML package diagrams are often used to illustrate the logical architecture of a system—the layers, subsystems, packages (in the Java sense), etc. A layer can be modelled as a UML package; for example, the UI layer modelled as a package named UI.

A UML package diagram provides a way to group elements. A UML package can group anything: classes, other packages, use cases, and so on. Nesting packages is very common. A UML package is a more general concept than simply a Java package or .NET namespace, though a UML package can represent those—and more.

The package name may be placed on the tab if the package shows inner members, or on the main folder, if not.

It is common to want to show dependency (a coupling) between packages so that developers can see the large-scale coupling in the system. The UML **dependency line** is used for this, a dashed arrowed line with the arrow pointing towards the depended-on package.

A UML package represents a namespace so that, for example, a *Date* class may be defined in two packages. If you need to provide **fully-qualified names**, the UML notation is, for example, *java*::*util*::*Date* in the case that there was an outer package named “java” with a nested package named “util” with a *Date* class.

The UML provides alternate notations to illustrate outer and inner nested packages. Sometimes it is awkward to draw an outer package box around inner packages. Alternatives are shown in Figure 13.3.

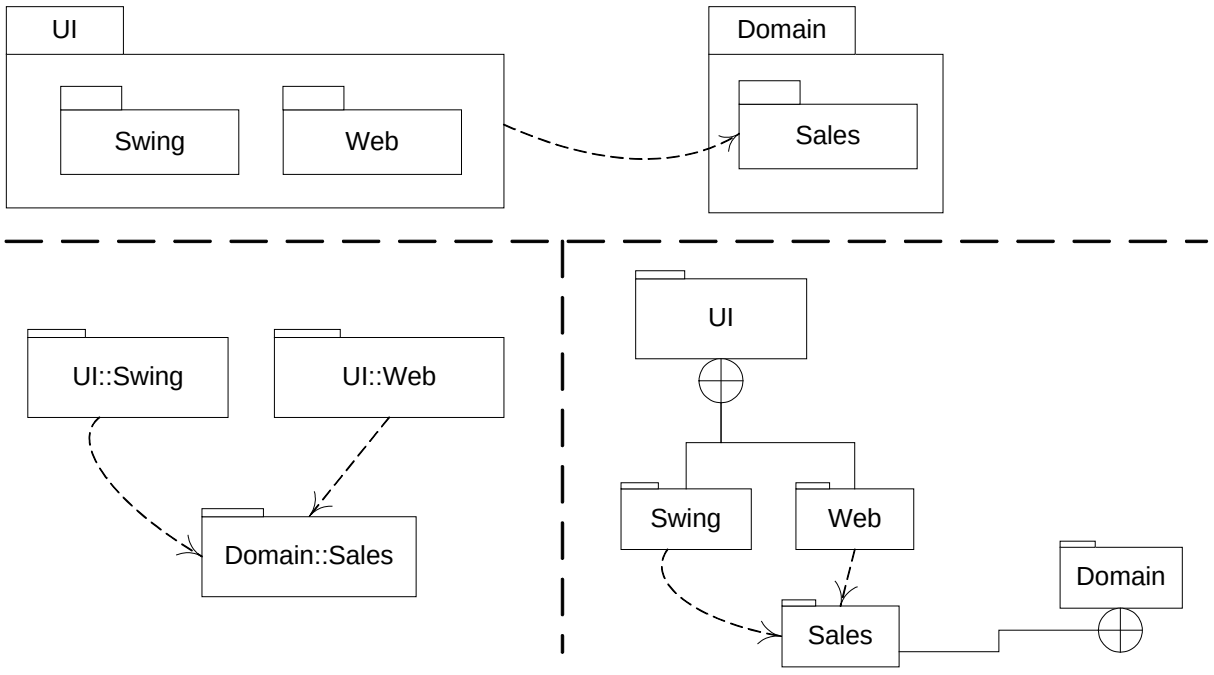

Figure 13.3 Alternate UML approaches to show package nesting, using embedded packages, UML fully-qualified names, and the circle-cross symbol.

13.6 Guideline: Design with Layers

The essential ideas of using layers [BMRSS96= Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad & Michael Stal. Pattern-Oriented Software Architecture: A System of Patterns, 1996] 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.

Some more design issues are covered later, starting on p. 559. The idea is described as the **Layers pattern** in [BMRSS96] and produces a **layered architecture**. It has been applied and written about so often as a pattern that the *Pattern Almanac 2000* [Rising00 = Linda Rising. The Pattern Almanac 2000, 2000] lists over 100 patterns that are variants of or related to the Layers pattern.

Using layers helps address several 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, so it cannot be reused with a different interface or distributed to another processing node.
- Potentially general technical services or business logic is intertwined with more application-specific logic, so it cannot be reused, distributed to another node, or easily replaced with a different implementation.
- There is high coupling across different areas of concern. It is thus difficult to divide the work along clear boundaries for different developers.

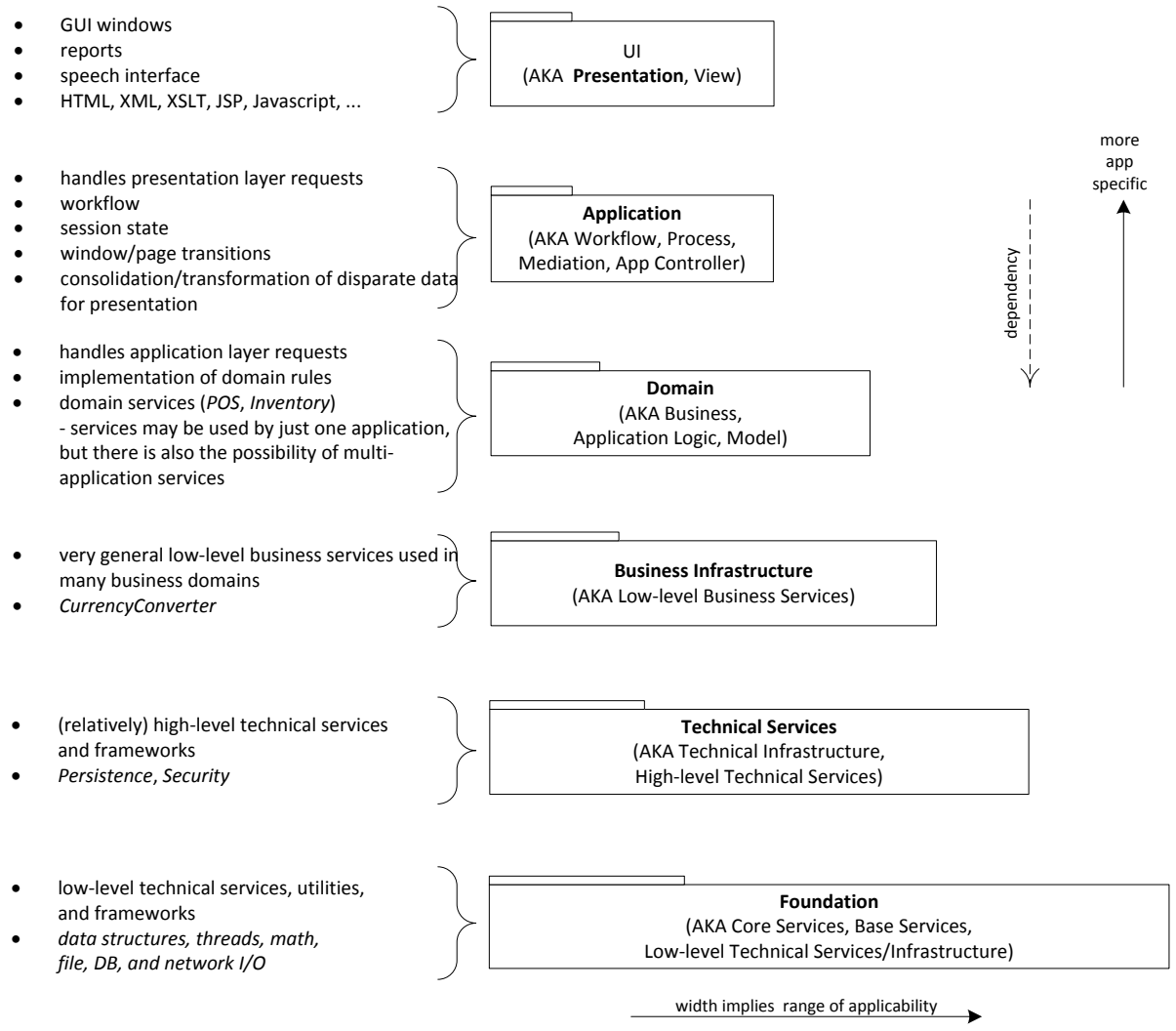

Figure 13.4. Common layers in an information system logical architecture.

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 13.4

Benefits of Using Layers

- 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 UI, 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