APPLYING UML AND PATTERNS

An Introduction to Object-Oriented Analysis and Design and the Unified Process

SECOND EDITION

Free Book for
Everyone

"People often ask me which is the best book to introduce them to the world of OO design. Ever since I came across it, *Applying UML and Patterns* has been my unreserved choice."
—Martin Fowler, author, *UML Distilled* and *Refactoring*

CRAIG LARMAN

Foreword by Philippe Kruchten

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.

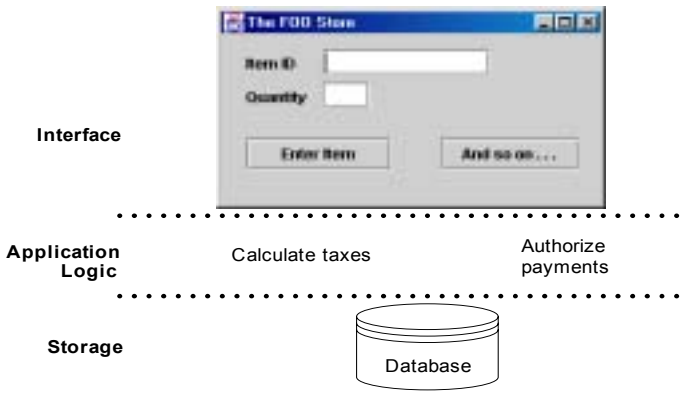

Figure 30.14 Classic view of a three-tier architecture.

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.

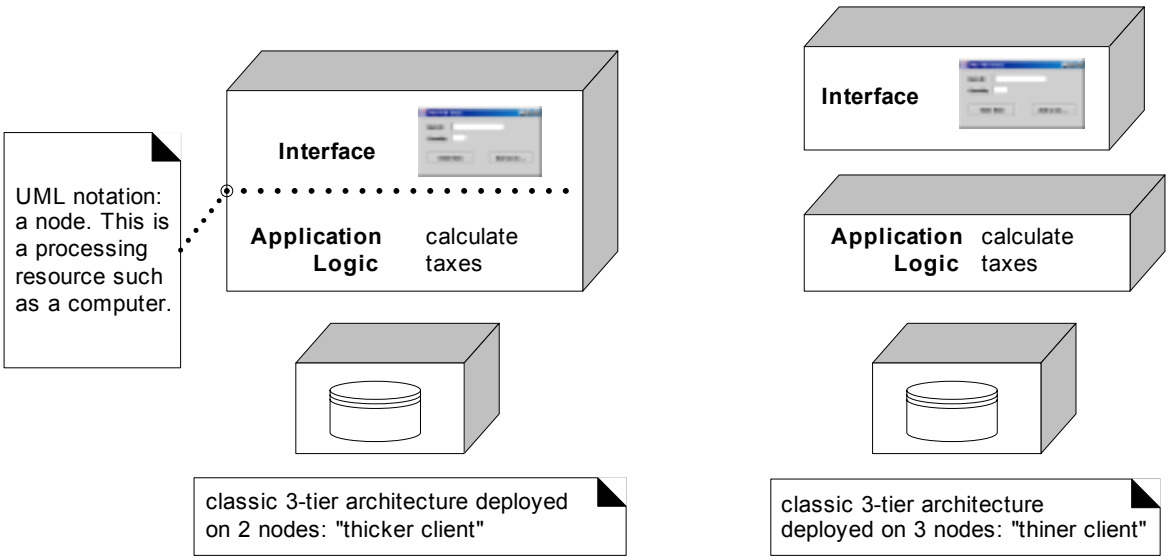

Figure 30.15 A three-tier logical division deployed in two physical architectures.

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.

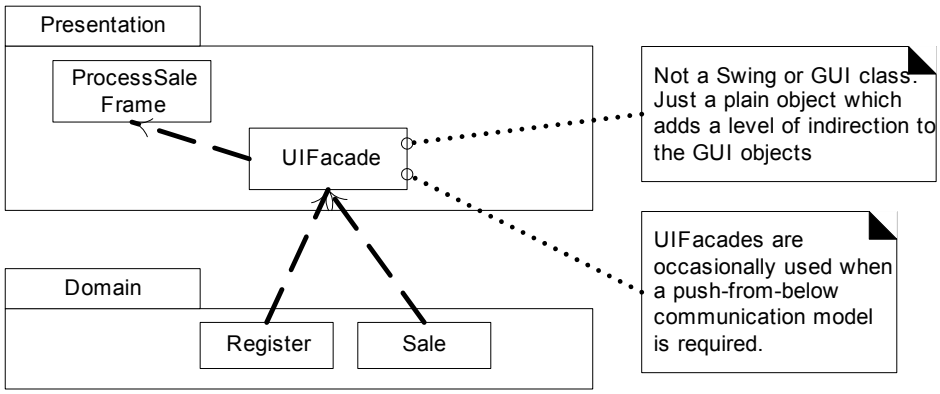

Figure 30.16 A Presentation layer UIFacade is occasionally used for push-from-below designs.

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 Presenta tion 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.