

# Layering Strategies

by [Peter Eeles](#)

Rational Services Organization, UK

*For system architects, all techniques for decomposing software systems address two main concerns:*

- 1. Most systems are too complex to comprehend in their entirety.*
- 2. Different audiences require different perspectives of a system.*

*Layering is a decomposition technique that has been adopted in numerous software systems and espoused in many texts as well as the Rational Unified Process® (RUP®). It is, however, often misunderstood and incorrectly applied. This article clarifies what is meant by layering, and discusses the impact of applying different layering strategies.*



## What Is "Layering"?

The term *layer* refers to the application of an architectural pattern generally known as the *Layers Pattern*, which is described in a number of texts<sup>1</sup> and also in the RUP. A *pattern* represents a solution to a common problem that exists in a particular context. Table 1 provides an overview of the Layers Pattern.

**Table 1: Overview of the Layers Pattern**

### Layers Pattern

#### Context

A system that requires decomposition.

#### Problem

A system that is too complex to comprehend in its entirety.

A system that is difficult to maintain.

A system whose least stable elements are not isolated.

A system whose most reusable elements are difficult to identify.

A system that is to be built by different teams, possibly with different skills.

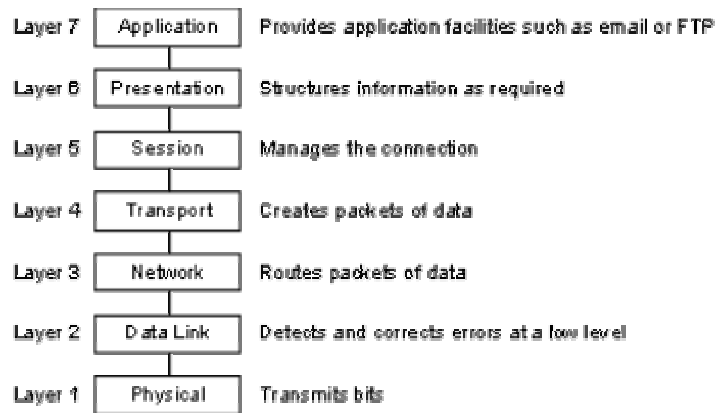
#### Solution

Structure the system into layers.

## Layering Basics

One of the most familiar models for layering is the OSI 7 Layer Model (Figure 1), defined by the International Organization for Standardization (ISO).<sup>2</sup> This model defines a set of networking protocols: Each layer focuses on a specific aspect of communication and builds upon the facilities of the layer below it. The model also uses a responsibility-based layering strategy that associates each layer with a particular

responsibility. Layering strategies can be based on other system characteristics as well, such as reuse. We will explore both responsibility-based and reuse-based strategies in more detail later.

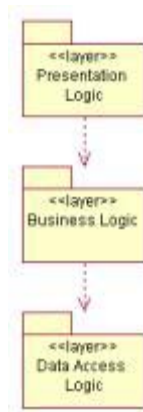


**Figure 1: OSI 7 Layer Model (Responsibility-Based Layering)**

First, however, let's look at Figure 2, which shows another, simple model for responsibility-based layering. Note that:

- The *presentation logic layer* contains elements responsible for providing some form of communication with a human being, such as an element in the user interface.
- The *business logic layer* contains elements responsible for performing some kind of business processing and the application of business rules.
- The *data access logic layer* contains elements responsible for providing access to an information source, such as a relational database.

Layers can be modeled in a number of ways, as we will see later on. Here, we represent a layer using a UML (Unified Modeling Language) Package with the stereotype <<layer>>.



**Figure 2: A Simple Model for Responsibility-Based Layering**

The layers in this kind of model are often called *tiers*, a familiar concept in distributed systems development, which deals with *2-tier*, *3-tier*, and *n-tier* systems.

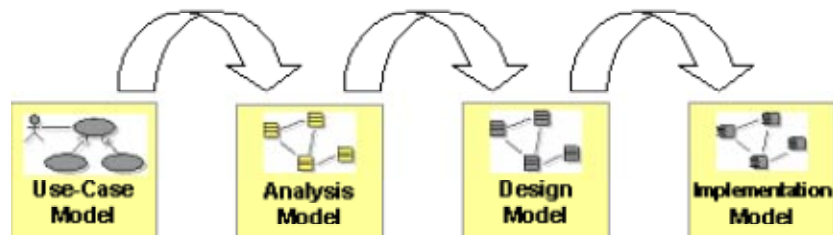
An important aspect of the model in Figure 2 is the *direction of dependencies*. The dependencies shown imply a certain rule that is a characteristic of layered systems: An element in a particular layer may only access elements in that same layer or layers below it.<sup>3</sup> In the example shown here, elements in the *Business Logic layer* may not access elements in the *Presentation Logic layer*. Also, elements in the *Data Access Logic layer* may not access elements in the *Business Logic layer*. This structure is often referred to as a directed acyclic graph (DAG). It is *directed* in that the dependencies are unidirectional, and *acyclic* in that the path of dependencies is never circular.

When defining a layering strategy, it is important to be precise about the meaning of each layer so that system architects and designers will correctly place elements in the appropriate layer. Incorrect placement will diminish the value of applying the strategy in the first place. As we discuss layering strategies in more detail below, we will also provide general guidance on what each layer means.

## Modeling Layers

For an architect, communicating any layering strategy effectively requires a set of specific *models* (and UML elements). A *model* represents a complete description of a system from a particular perspective. Figure 3 shows an example of four models that represent different perspectives of the system under consideration:

- *Use-Case Model*: captures system *requirements*.
- *Analysis Model*: captures system *requirements analysis*.
- *Design Model*: captures system *design*.
- *Implementation Model*: captures system *implementation*.



**Figure 3: Four Models Representing Gradual System Refinement**

Additional models include:

- *Deployment Model*: captures the *distribution* aspects of a system.
- *Data Model*: captures the *persistent* aspects of a system.

## Layering Strategies

Now that we have a basic understanding of what layering is and what is required to represent it, let's look at two specific layering strategies in more detail. As we saw above, layering can be based on a number of characteristics, including responsibility and reuse. This section discusses strategies based on these characteristics, as well as ways to depict them.

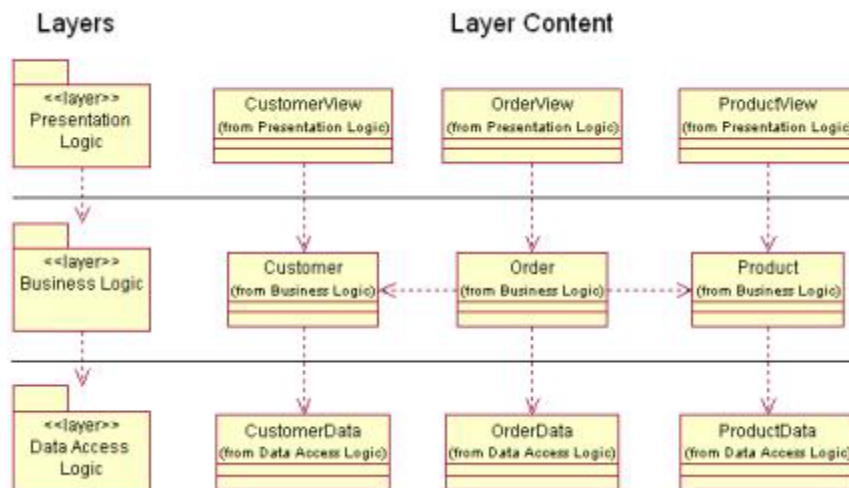
### Responsibility-Based Layering

Responsibility-based layering is probably the most commonly used layering strategy. Because it isolates various system responsibilities from one another, this strategy can improve both system development and maintenance. As we saw in Figure 2, a system can be layered based upon the following responsibilities:

- Presentation logic
- Business logic
- Data access logic

Each of these responsibilities can be represented as a layer, as shown in Figure 4, which also shows sample content for each layer. Here we consider three concepts in an order processing system: Customer, Order, and Product. As an example, the *Customer* concept comprises the:

- *CustomerView class*: responsible for the *presentation logic* associated with a customer, such as the rendering of a customer in the user interface.
- *Customer class*: responsible for the *business logic* associated with a customer, such as validation of customer details.
- *CustomerData class*: responsible for the *data access logic* associated with a customer, such as making the state of a customer persistent.



**Figure 4: Layers and Content for Responsibility-Based Layering**

Despite its widespread use, this layering strategy has generated some myths that either discourage people from using it or confuse those who do. Let's look at some of these myths and attempt to clear up the confusion.

### **Myth 1: Layers and Tiers Are Different**

This particular "myth" is a frequent source of confusion. In fact, a tier *is* a layer, albeit a layer associated with a particular strategy: responsibility-based layering. Certainly, the numerous ways in which tiers are organized, as shown in Table 2, may cause some confusion when attempting to understand the relationship between tiers and layers.

**Table 2: Tier Definitions**

**Application  
Layers (Tiers)**

**2-tier**

combined presentation logic and business logic  
data access logic

**3-tier**

presentation logic  
business logic  
data access logic

**n-tier**

presentation logic  
business logic (*distributed*)  
data access logic

**Myth 2: Layers (Tiers) Imply a Physical Distribution**

Another common misconception is that the logical layering implies a physical distribution. Consider a 3-tier layering. Various elements will reside in a particular layer, but each layer can itself be applied in a number of ways, as shown in Table 3. Note that the names in the left column (e.g., "thin client") are those commonly used to characterize a particular physical distribution.

**Table 3: Application of 3-Tier Layering**

**Application  
Layers**

**Client side**

**Server side**

**Single system**

presentation logic  
business logic  
data access logic

**Thin client**

presentation logic  
business logic  
data access logic

**Fat client**

presentation logic  
business logic  
data access logic

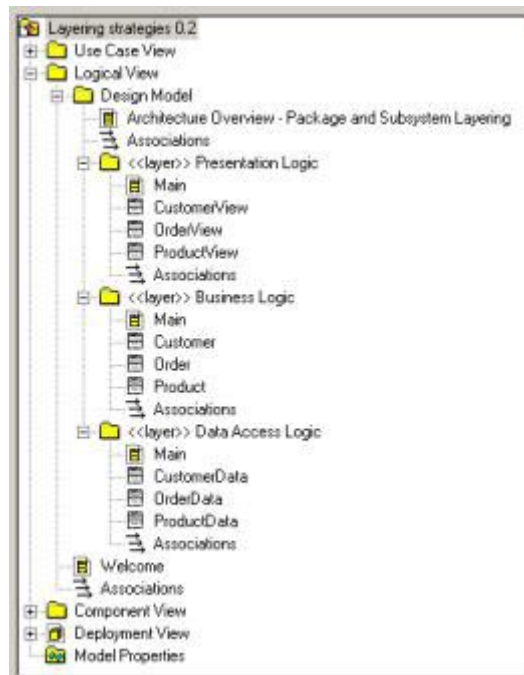
It is also true that a single system may employ more than one physical distribution strategy; certain elements would characterize a "thin client" distribution, while others would represent a "fat client" distribution. The choice is typically made based upon nonfunctional requirements such as performance.

**Modeling Responsibility-Based Layers**

Responsibility-based layering typically affects the *design model*, *implementation model*, and *deployment model*. The *design model* is typically structured using one of two approaches.

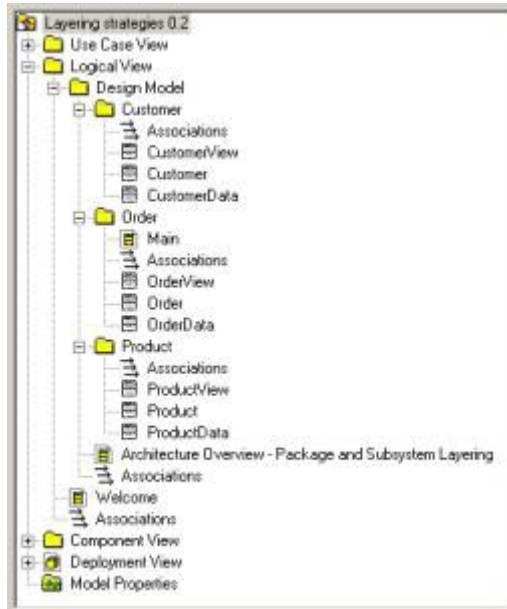
The **first approach** shows the elements "contained" within the layer. The result is implied in Figure 5, a Rational Rose® browser screenshot, which shows:

- *Presentation classes* (CustomerView, OrderView, and ProductView) residing within a Presentation Logic package.
- *Business logic classes* (Customer, Order, and Product) residing within a Business Logic package.
- *Data access logic classes* (CustomerData, OrderData, and ProductData) residing within a Data Access Logic package.



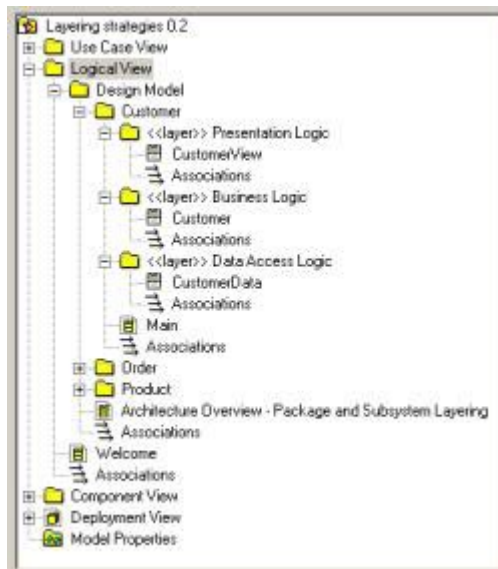
**Figure 5: Rational Rose Showing Elements Contained Within Layers in a Design Model**

The **second approach** focuses on the *business component* (in this case, Customer, Order, and Product); the primary elements of concern are the domain-related concepts supported by the system. The concept *Customer*, for example, may have associated elements of presentation logic, business logic, and data access logic.<sup>4</sup> This way of thinking results in the model structure shown in Figure 6. In this example, the layering is *implied* by the element names. For example, all *View* classes (such as *CustomerView*) imply a presentation logic layer, and all *Data* classes (such as *CustomerData*) imply a data access logic layer. Unqualified class names (such as *Customer*) imply a business logic layer.



**Figure 6: Rational Rose Showing *Implicit* Layering Within Each Business Component Package of a Design Model**

The layering could also be represented explicitly within each package representing a business component, as shown in Figure 7. *This structuring is preferable when a number of elements are involved in each layer of a given business component.* Although only the Customer business component package has been expanded in this example, the Order and Product packages would have a similar structure.

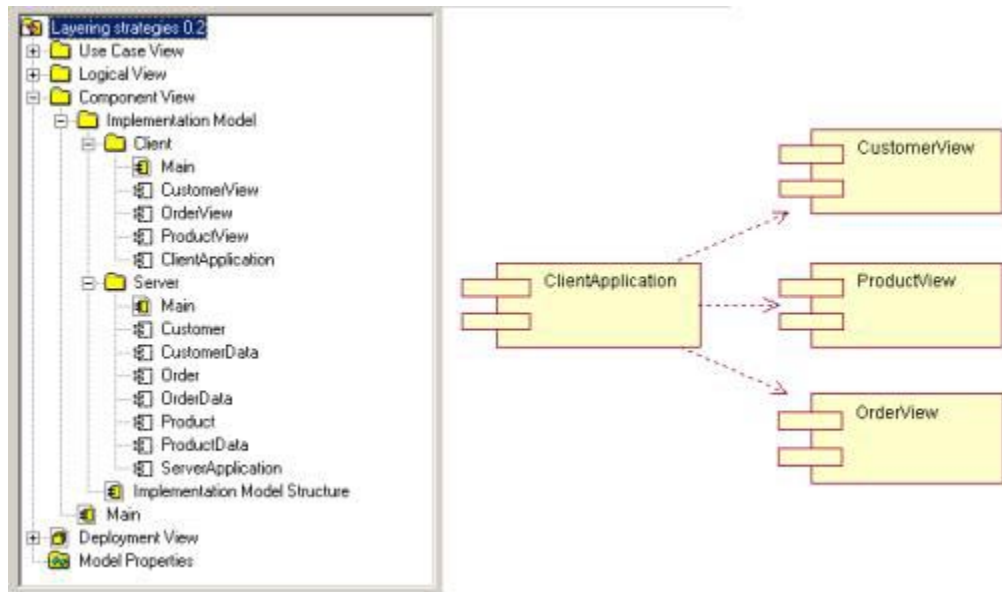


**Figure 7: Rational Rose Showing *Explicit* Layering Within a Business Component Package of a Design Model**

A responsibility-based layering strategy typically affects the *implementation model* as well as the design model, when there is a need to physically partition the elements that implement each responsibility. For example, consider a system with a "thin client"

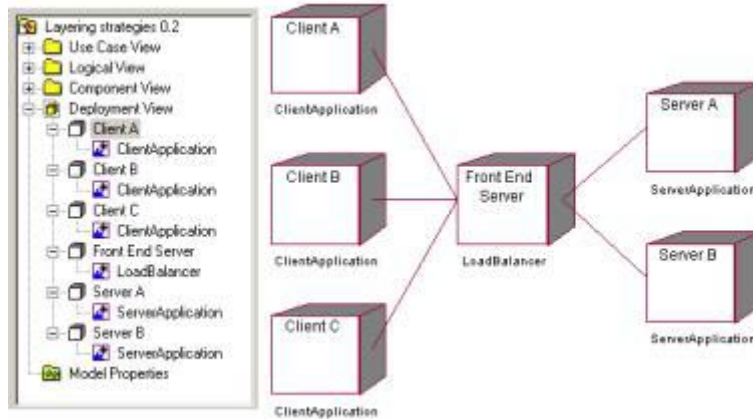
physical distribution: It is useful to identify the implementation units required to support execution on the client and server, respectively. In this example, the elements in the *presentation logic layer* reside in an application that is deployed on a client, and all elements in the *business logic layer* and *data logic layer* reside in another application, which is deployed on a server.

This scenario implies an *implementation model* like the one shown in Figure 8. Here, a Rational Rose browser image and a component diagram show the elements of the application deployed on the client. There happens to be a one-to-one mapping between a class in the design model and a UML component in the implementation model. Note, however, that mapping typically depends on the implementation technology used.



**Figure 8: Rational Rose and Component Diagram Showing *Implicit* Layering in an Implementation Model**

Similarly, a responsibility-based layering strategy affects the *deployment model* when there is a need to describe the physical distribution of the responsibilities. Referring to Figure 9 and the example above, we can see that six nodes have been defined. Each of the three *Client nodes* houses a *ClientApplication* process. The *FrontEndServer node* houses a *LoadBalancer* process that is responsible for distributing client requests to one of two *Server nodes*. Each *Server node* houses a *ServerApplication* process.



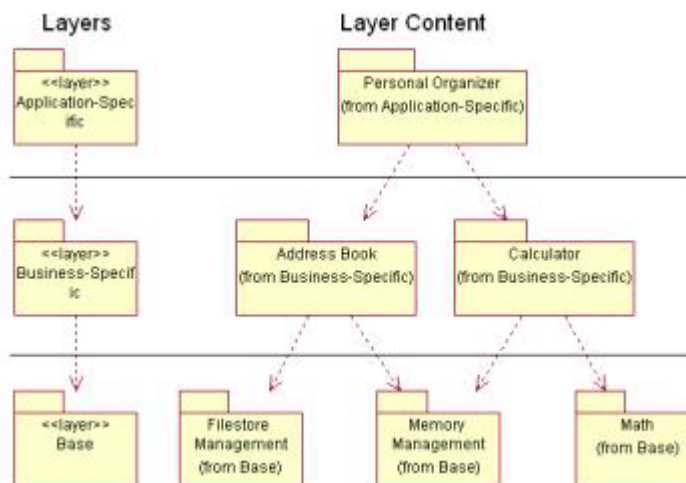
**Figure 9: Rational Rose and Diagram Describing Physical Distribution of Responsibilities in a Deployment Model**

### Reuse-Based Layering

Another commonly used layering strategy, based on reuse, is particularly relevant to organizations that have an identifiable goal to reuse components. This strategy explicitly groups components according to their level of reuse, and makes their degree of reusability highly visible.

The example in Figure 10<sup>5</sup> shows three layers:

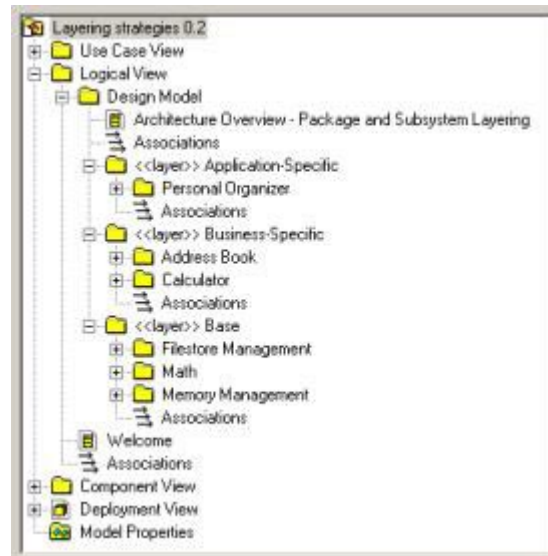
- The *Base layer* contains elements (e.g., Math) that might apply across organizations. Such elements will be reused widely.
- The *Business-Specific layer* contains elements that apply to a particular organization but are application-independent (e.g., Address Book). Such elements will be reused in applications within the same organization.
- The *Application-Specific layer* contains elements that apply to a particular application, or project (such as Personal Organizer). These elements are the least reusable.



**Figure 10: Example of Reuse-Based Layering**

### Modeling Reuse-Based Layers

A reuse-based strategy primarily influences the *design model*. The structure of a design model that incorporates reuse-based layering is easy to depict, as shown in Figure 11 (which reflects the example in Figure 10).



**Figure 11: Rational Rose Design Model Incorporating Reuse-Based Layering**

### Multidimensional Layering

This article has focused on the two most widely used layering strategies, but other related strategies might organize layering by characteristics such as security, ownership, and skill sets. In addition, the strategies we have examined can also be combined to create new layering strategies. Figure 12, for example, includes two reuse-based layers from the previous example (application-specific and business-specific) and three responsibility-based layers or tiers (presentation logic, business logic, and data access logic).

The dependencies in a reuse-based layering strategy typically result from dependencies between elements in business logic layers, as implied in Figure 12, which shows dependency between PersonalOrganizer and AddressBook.

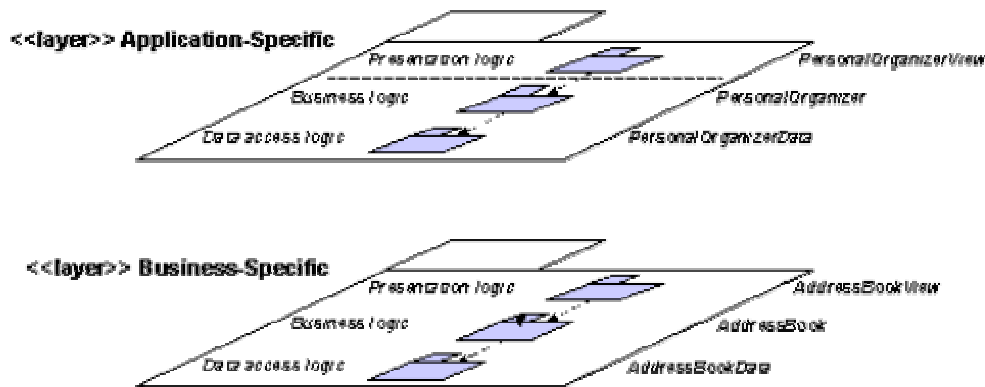


Figure 12: A Multidimensional Layering Approach

### Modeling Multidimensional Layers

In Figure 13 we can see the multidimensional aspects of layering within a two-dimensional *design model*, which also incorporates the *business component* concept.

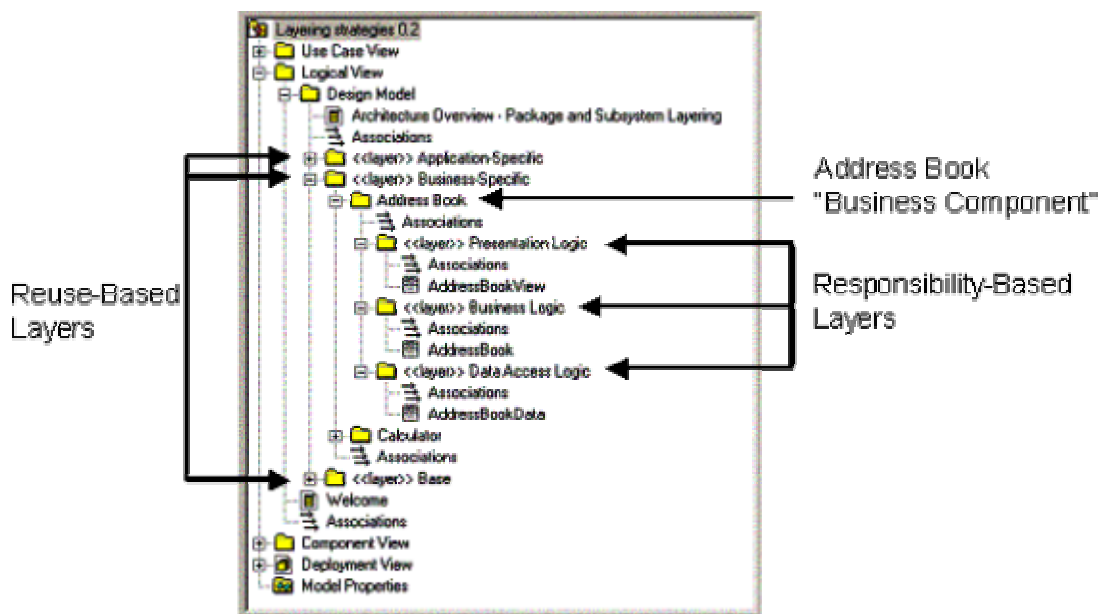


Figure 13: Rational Rose Showing Multidimensional Layering in a Design Model

Before you can adopt a multidimensional layering strategy, you must identify a primary strategy; ours is based upon reuse in this example. The design model is first organized based on this strategy, yielding the layers *Application-Specific*, *Business-Specific*, and *Base*. Each layer is then further organized by the elements that reside on each. Figure 13 shows that the *Business-Specific* layer contains *Address Book* and *Calculator*, for example. Each of these elements is then further organized based upon a secondary strategy: responsibility-based layering. For example, the *Address Book* package contains the three layers *Presentation Logic*, *Business Logic*, and *Data Access Logic*. Each of these layers, in turn, contains any elements that reside on that layer: the *Presentation Logic* layer contains the *AddressBookView* class; the *Business Logic*

*layer* contains the *AddressBook* class; and the *Data Access Logic layer* contains the *AddressBookData* class.

## An Important Choice

One of the most important decisions an architect must make is choosing an appropriate layering strategy, since that strategy will have a major influence on the structure of the system models. Even more important, the right layering strategy can directly support significant business benefits such as maintainability and reuse. If the architect chooses a responsibility-based layering strategy, for example, and isolates the different responsibilities of the system from one another, then the system the project team develops is likely to be more easily maintainable than it would be otherwise. If the architect adopts a reuse-based layering strategy, then reusable system elements can be clearly identified.

## Acknowledgments

The author would like to acknowledge the contributions of Kelli Houston, Wojtek Kozaczynski, Philippe Kruchten, Bran Selic, and Catherine Southwood (all of Rational Software) for their insightful comments on early drafts of this paper.

---

## References

Frank Buschmann et al., *A System of Patterns*. John Wiley & Sons, 1996.

Jeri Edwards, *3-Tier Client/Server at Work*. John Wiley & Sons, 1999.

Peter Eeles and Oliver Sims, *Building Business Objects*. John Wiley & Sons, 1998.

Peter Herzum and Oliver Sims, *The Business Component Factory*. John Wiley & Sons, 2000.

Ivar Jacobson et al., *Software Reuse*. Addison-Wesley, 1997.

John Vlissides, James Coplien, and Norman Kerth. *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

---

## Notes

<sup>1</sup> These include Frank Buschmann et al., *A System of Patterns*, John Wiley & Sons, 1996; Peter Herzum and Oliver Sims, *The Business Component Factory*, John Wiley & Sons, 2000; and John Vlissides, James Coplien, and Norman Kerth, *Pattern Languages of Program Design 2*, Addison-Wesley, 1996.

<sup>2</sup> The abbreviation *ISO* would suggest a name such as *International Standards Organization*, but, in fact, *ISO* is not an acronym. The following text is taken from the

ISO Web site: "ISO is a word, derived from the Greek *isos*, meaning *equal*, which is the root of the prefix *iso-* that occurs in a host of terms, such as *isometric* (of equal measure or dimensions) and *isonomy* (equality of laws, or of people before the law). From *equal* to *standard*, the line of thinking that led to the choice of *ISO* as the name of the organization is easy to follow. In addition, the name *ISO* is used around the world to denote the organization, thus avoiding the plethora of acronyms resulting from the translation of *International Organization for Standardization* into the different national languages of members, e.g., *IOS* in English, *OIN* in French (from *Organisation Internationale de Normalisation*). Whatever the country, the short form of the organization's name is always *ISO*."

<sup>3</sup> Although an event notification may result in a message from an element in one layer being sent to an element in an upper layer, there is no explicit dependency in this direction.

<sup>4</sup> This concept of a business component is discussed further in both Peter Eeles and Oliver Sims, *Building Business Objects*, John Wiley & Sons, 1998; and Peter Herzum and Oliver Sims, *The Business Component Factory*, John Wiley & Sons, 2000.

<sup>5</sup> Derived from a strategy described in Ivar Jacobson et al., *Software Reuse*. Addison-Wesley, 1997.

---