# Layering Strategies


## Peter Eeles


Rational Software White Paper


TP 199, 08/01


Rational®
the software development company

# Table of Contents

## Abstract

A number of techniques exist for decomposing software systems. Layering is one example and is described in this paper. Such techniques address two main concerns: most systems are too complex to comprehend in their entirety and different perspectives of a system are required for different audiences.

Layering has been adopted in numerous software systems, and is espoused in many texts and also in the Rational Unified Process (RUP). However, layering is often misunderstood and incorrectly applied. This paper clarifies what layering means and discusses the impact of applying different layering strategies.

## What is "Layering"?

Let's start by defining what we mean by "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 ([Buschmann], [Herzum], [PloP2]), and also in the RUP. A *pattern* represents a solution to a common problem that exists in a particular context. An overview of the Layers pattern is given in Table 1.

**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<br>A system that is difficult to maintain<br>A system whose least stable elements are not isolated<br>A system whose most reusable elements are difficult to identify<br>A system that is to be built by different teams, possibly with different skills |
| Solution | Structure the system into layers |

One of the most familiar examples of layering is the OSI 7-layer model, defined by the International Standardization Organization (ISO). This model, shown in Figure 1, 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 OSI 7-layer model uses a responsibility-based layering strategy: each layer has a particular responsibility. This strategy is described in detail later in this paper.

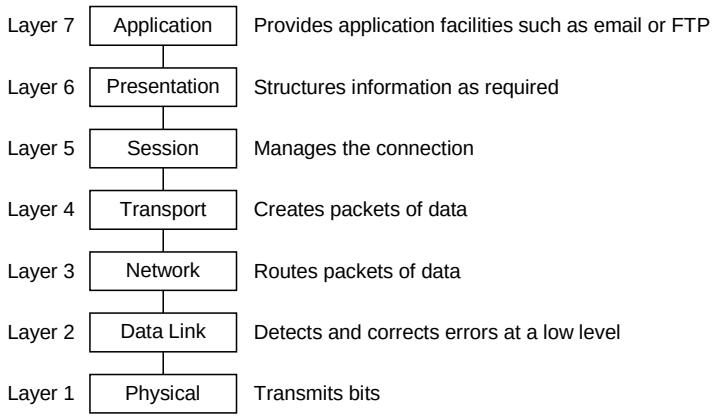


**Figure 1: OSI 7-Layer Model (Responsibility-based Layering)**

Figure 2 shows another example of responsibility-based layering.

- The *Presentation Logic layer* contains elements responsible for providing some form of rendering to 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 applying the business rules.
- The *Data Access Logic layer* contains elements responsible for providing access to an information source, such as a relational database.

It should be noted that layers can be modeled in a number of ways, as described later in this paper. For now, we will explicitly represent a layer using a UML package with the stereotype «layer».

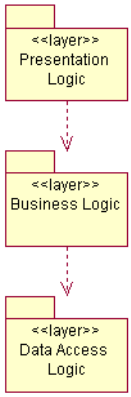


**Figure 2: Responsibility-based Layering**

The layers shown in this particular example of responsibility-based layering are often called “tiers” and are a familiar concept in distributed systems development where *2-tier*, *3-tier*, and *n-tier* systems are found.

An important aspect of Figure 2 is the *direction of dependencies* shown, since it implies a certain rule that is a characteristic of layered systems — an element in a particular layer may only access elements in the same layer, or in layers below it[1]. In the example given 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 a path of dependencies is never circular.

On a more specific note, it is important to be precise about the meaning of each layer when defining a layering strategy so that elements are correctly placed in the appropriate layer. Failure to correctly assign an element to the appropriate layer will diminish the value of applying the strategy in the first place. As each layering strategy is discussed in more detail, some general guidance is given on the meaning of each of the layers.

## Modeling Layers

As we investigate different layering strategies, it will become clear that it is appropriate to communicate each strategy using specific *models* (and therefore specific 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:

[1] Although an event notification may result in a message from an element in one layer being sent to an element in an upper layer, no explicit dependency in this direction exists.

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

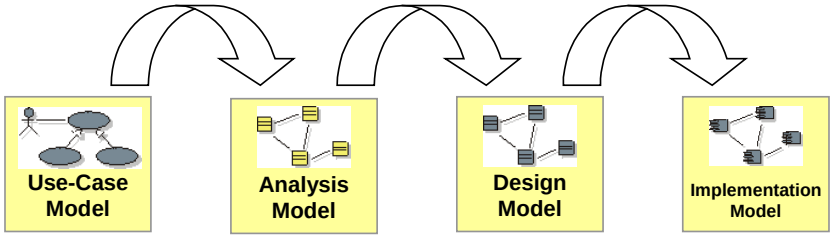


**Figure 3: Four Models Representing Gradual Refinement**

Additional models include:

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

## *Layering Strategies*

Layering can be based on a number of characteristics. This section discusses layering based on the following characteristics:

- responsibility
- reuse

The representation of each strategy will be considered as each strategy is discussed in detail.

### Responsibility-based Layering

Probably the most commonly used layering strategy is one based upon responsibility. This particular strategy can improve the development and maintenance of a system since various system responsibilities are isolated from one another. As an example (see 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 displays some 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 following:

- *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

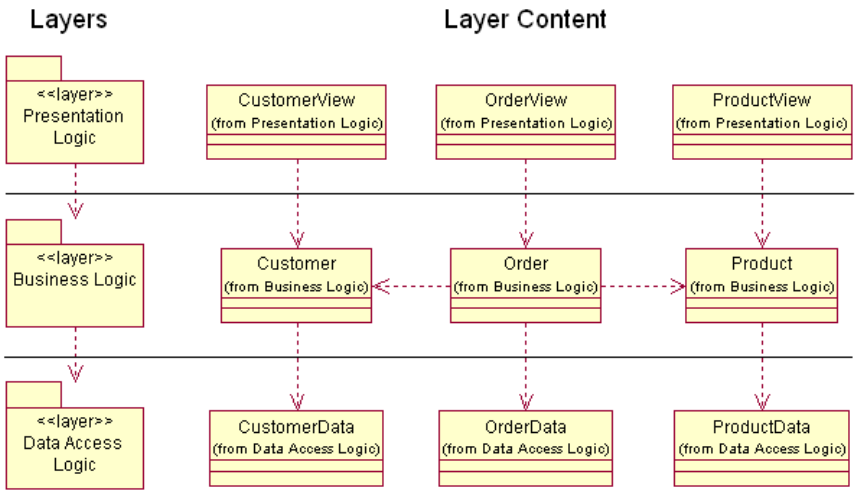


**Figure 4: Layers and Content for a Responsibility-based Layering**

We'll now consider some "myths" regarding this particular layering strategy.

### *Myth 1: Layers and tiers are different*

This particular myth is a commonly encountered source of confusion. The fact is that a tier *is* a layer, albeit a layer based upon a particular strategy—one of responsibility. The confusion is compounded by the fact that the concepts of tiers can be applied in numerous ways, as shown in Table 2.

**Table 2: Tier Definitions**

| Application | Layers (Tiers) |
|---|---|
| 2-tier | combined presentation logic and business logic<br>data access logic |
| 3-tier | presentation logic<br>business logic<br>data access logic |
| n-tier | presentation logic<br>business logic (*distributed*)<br>data access logic |

### *Myth 2: Layer (tiers) imply a physical distribution*

Another common misconception is that the logical layering implies a physical distribution. Consider a 3-tier layering. Even though various elements will reside in one of the layers, each layer itself can be applied in a number of ways as shown in Table 3, which uses names often used to characterize a particular physical distribution (such as “thin client”).

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

| Application | Layers | |
|---|---|---|
| | Client side | Server side |
| Single system | presentation logic<br>business logic<br>data access logic | |
| Thin client | presentation logic | business logic<br>data access logic |
| Fat client | presentation logic<br>business logic | data access logic |

It is also true to say that a single system may employ more than one physical distribution strategy, where certain elements would be classified as epitomizing a “thin client” distribution and others a “fat client” distribution. Typically, the choice is based upon nonfunctional requirements such as performance.

### *Modeling responsibility-based layers*

As we will see, the application of this strategy may influence 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 being “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

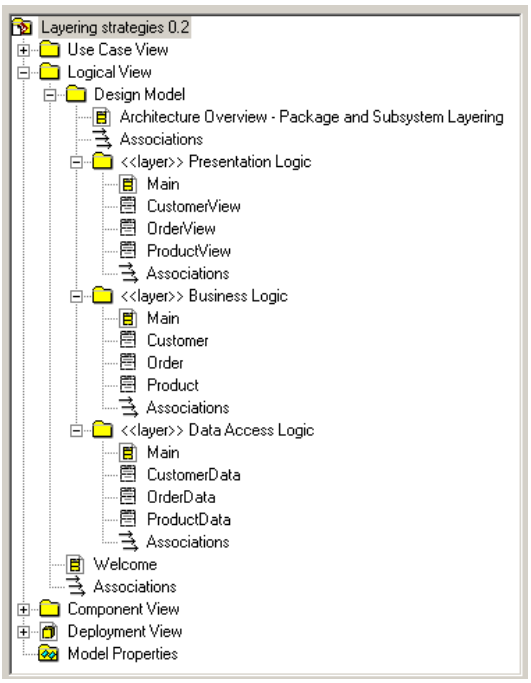


**Figure 5: Elements Contained Within Layers**

The second approach incorporates the concept of a *business component* (in this case, Customer, Order, and Product) as a first-class citizen, whereby the primary elements of concern are the domain-related concepts supported by the system. For example, the concept *Customer* may have associated elements of presentation logic, business logic, and data access logic. This concept of a business component is discussed further in [Eeles] and [Herzum]. 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.

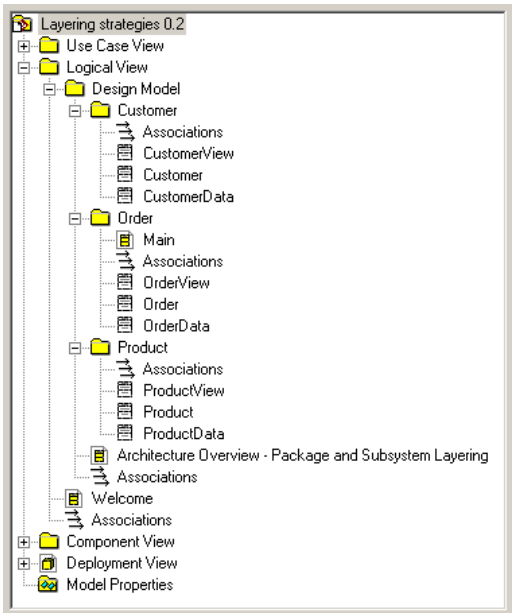


**Figure 6: *Implicit* Layering Within Each Business Component Package**

The layering could also be represented explicitly within each package representing a business component as shown in Figure 7. *This structuring is preferable when there are a number of elements 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.

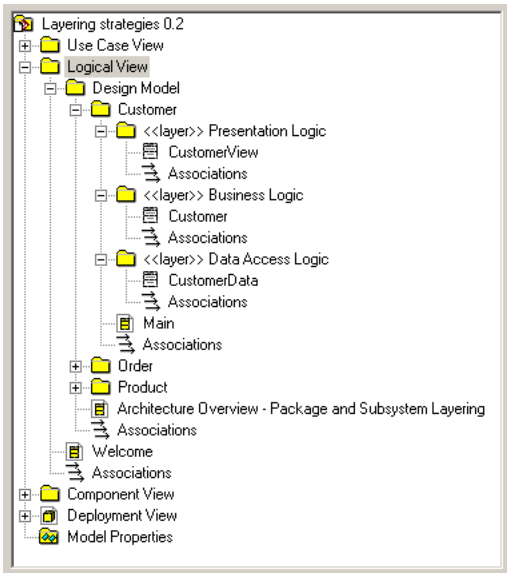


**Figure 7: *Explicit* Layering Within a Business Component Package**

A responsibility-based layering strategy typically influences the *implementation model* in addition to the design model, when there is a need to physically partition the elements that implement each responsibility. For example, consider a system that exhibits a "thin client" physical distribution: it is useful to identify the implementation units required to support execution on a client and those required to support execution on the server. In this example, the elements in the *presentation logic layer* reside in an application that is deployed on a client, and all of the elements in the *business logic layer* and *data logic layer* reside in another application that is deployed on a server.

This scenario implies an *implementation model* as shown in Figure 8, which presents a Rational Rose browser image and a component diagram displaying the elements of the application that is deployed on the client. In this example, 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 this mapping typically depends on the implementation technology used.

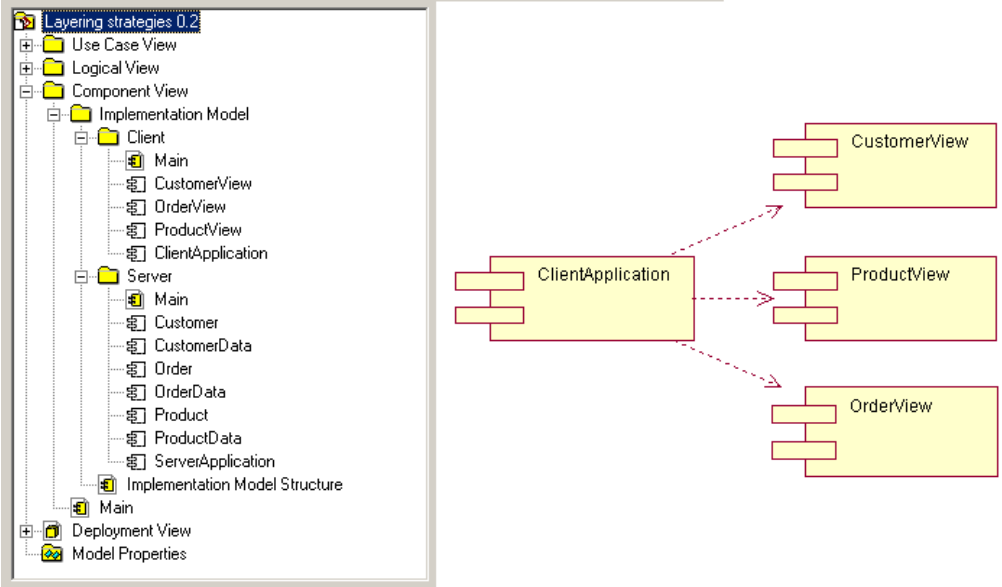


**Figure 8: *Implicit* Layering Within Implementation Model**

Similarly, a responsibility-based layering strategy can also influence the *deployment model* when there is a need to describe the physical distribution of the responsibilities. In Figure 9, and using 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.

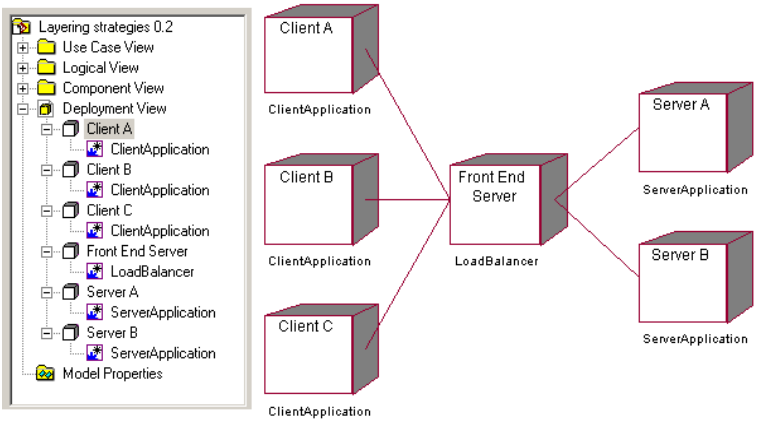


**Figure 9: Deployment Model Describing Physical Distribution of Responsibilities**

## Reuse-based Modeling

Another commonly used layering is one based upon reuse. This strategy is particularly relevant to organizations that have an identifiable goal to reuse components throughout the organization. The impact of using this layering strategy is that the reusability of components is highly visible, since components are explicitly grouped according to their level of reuse. An example layering, derived from a strategy described in [Jacobson], is shown in Figure 10. Here we see three layers: Base, Business-Specific, and Application-Specific.

- The *Base layer* contains elements that might apply across organizations (such as Math). Such elements will be reused widely.
- The *Business-Specific layer* contains those elements that apply to a particular organization, but are application-independent (such as Address Book). Such elements will be reused within applications in 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.

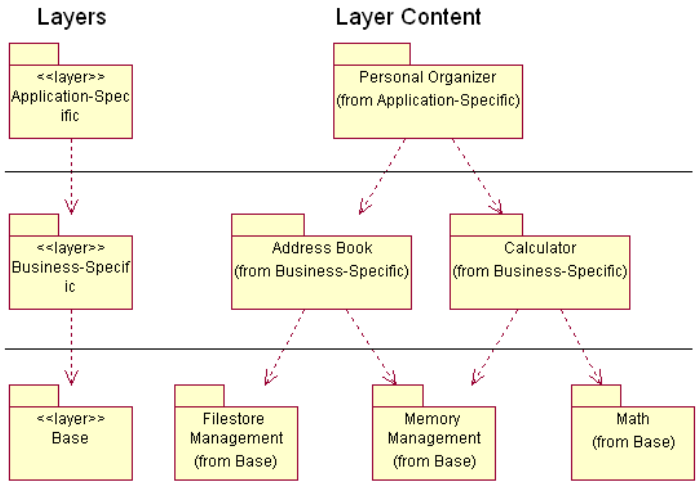


**Figure 10: Example of Reuse-based Layering**

We can derive that the elements in the Base layer are the most reusable, whereas those in the Application-Specific layer are more project specific and, therefore, less reusable.

### *Modeling reuse-based layers*

The application of a reuse strategy primarily influences the *design model*. The structure of a design model that incorporates reuse-based layering is straightforward to envisage and is shown in Figure 11, which reflects the example in Figure 10.

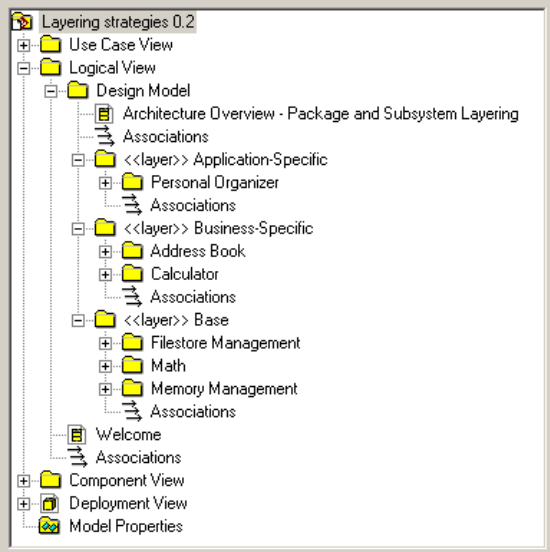


**Figure 11: Design Model Incorporating Reuse-based Layering**

## Other Layering Strategies

This paper is intended to simply give a "flavor" of the different layering strategies that exist, using the two most widely used strategies as examples. However, similar approaches could be taken for strategies that acknowledge characteristics such as security, ownership, and skill set.

## Multi-dimensional Layering

The strategies previously described can also be combined to create new layering strategies. The example in Figure 12 shows:

- two of the reuse-based layers from the previous example
  - application-specific
  - business-specific
- three responsibility-based layers (tiers)
  - presentation logic
  - business logic
  - data access logic

The dependencies present in the reuse-based layering strategy typically result from dependencies between elements in business logic layers, as implied in Figure 12, where we see the dependency between PersonalOrganizer and AddressBook.

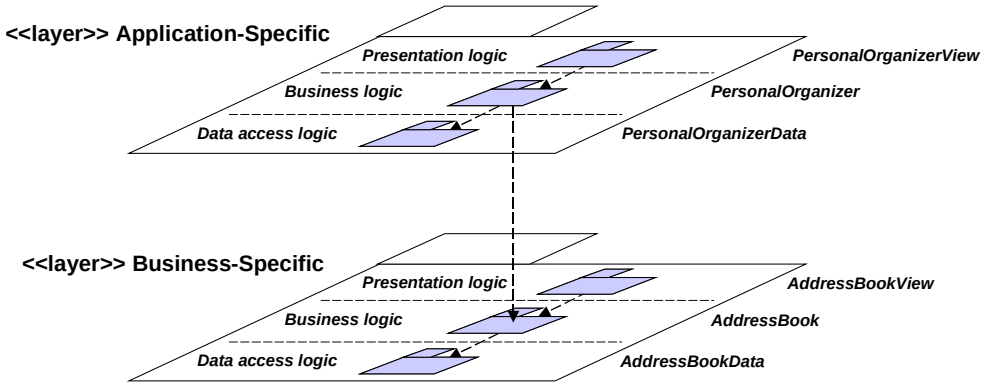


**Figure 12: Multidimensional Layering**

### *Modeling multidimensional layers*

Here we consider the representation of the multidimensional aspects of layering within a two-dimensional *design model*. We also consider a structure whereby the *business component* concept is incorporated.

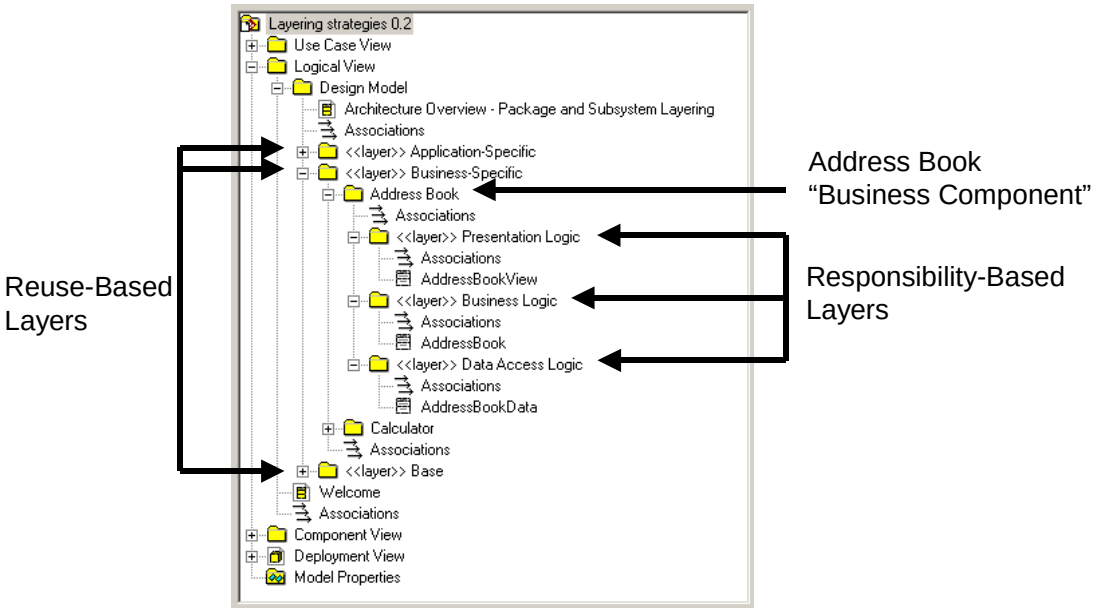


**Figure 13: Design Model Incorporating Multidimensional Layering**

Adopting a multidimensional layering strategy requires that a primary strategy be identified. In our example, the primary layering strategy is based upon reuse. The design model is first organized based on this strategy giving the layers *Application-Specific*, *Business-Specific*, and *Base*. Each of these layers is then further organized by the elements that reside on each of these layers; for example, Figure 13 shows the Business-Specific layer containing *Address Book* and *Calculator*. 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 then contains any elements that reside on this layer:

- the *presentation logic layer* contains the AddressBookView class
- the *business logic layer* contains the AddressBook class

- the *data access logic layer* contains the AddressBookData class

## Conclusion

One of the most important decisions an architect must make is choosing an appropriate layering strategy, since it will have a major influence on the structure of the models produced. Of more significance, however, is the fact that business benefits, such as maintainability and reuse, can be directly supported by the layering strategy chosen. For example, more maintainable systems are likely to be developed should the different responsibilities of the system be isolated from one another through the adoption of a responsibility-based layering strategy. Also, reusable system elements can be clearly identified using a reuse-based layering strategy.

## Acknowledgements

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.

## Bibliography

[Buschmann] Buschmann, Frank, et al. *A System of Patterns*. 1996. New York: John Wiley & Sons. ISBN 0-471-95869-7.

[Edwards] Edwards, Jeri. *3-Tier Client/Server at Work*. 1999. New York: John Wiley & Sons. ISBN 0-471-31502-8.

[Eeles] Eeles, Peter, and Oliver Sims. *Building Business Objects*. 1998. New York: John Wiley & Sons. ISBN 0-471-19176-0.

[Herzum] Herzum, Peter, and Oliver Sims. *The Business Component Factory*. 2000. New York: John Wiley & Sons.

[Jacobson] Jacobson, Ivar, et al. *Software Reuse*. 1997. Reading, Massachusetts: Addison-Wesley. ISBN 0-201-92476-5.

[PLoP2] Vlissides, John, James Coplien, and Norman Kerth. *Pattern Languages of Program Design 2*. 1996. Reading, Massachusetts: Addison-Wesley. ISBN 0-201-89527-7.

Rational®
the software development company

Dual Headquarters:

Rational Software
18880 Homestead Road
Cupertino, CA 95014
Tel: (408) 863-9900

Rational Software
20 Maguire Road
Lexington, MA 02421
Tel: (781) 676-2400

Toll-free: (800) 728-1212
E-mail: info@rational.com
Web: www.rational.com
International Locations: www.rational.com/worldwide