Modeling for enterprise initiatives with IBM Rational Unified Process

Part II: Example

by Peter Eeles
Rational Software, UK
IBM Software Group

and

Maria Ericsson
Rational Software, Sweden
IBM Software Group

This article is the second in a two-part series describing the application of the IBM Rational Unified Process,® or RUP,® and the System of Interconnected Systems pattern to initiatives such as enterprise architecting and systems engineering. Part I defined key concepts and described the relationship between a superordinate system and a number of subordinate systems in terms of certain key RUP artifacts,¹ as shown in Figure 1. In this part, we will provide an example of how these artifacts work in practice, by following the development of an order processing system. The artifacts we will consider in this example are also indicated in Figure 1.

Although we will discuss the relationships among RUP artifacts, we will not get bogged down in the detail of a particular process for deriving these artifacts. With this example, our intention is not to advocate a particular process for developing a system of systems. As we discussed in Part I, the System of Interconnected Systems Pattern can be applied to many different initiatives, and we fully expect that a number of RUP configurations will be made available to address each of the initiatives introduced in Part 1 (although they will all have a common basis). At the time of this writing, the first configuration has been made publicly available with the release of RUP SE, a configuration of RUP specifically tailored for systems engineering projects.
Superordinate system development

As we mentioned in Part I, the development of a superordinate system focuses on providing a "broad brush" perspective of a system; it tends to concentrate only on architecturally significant elements. For example, we do not typically produce a fully-detailed Use-Case Model, Analysis Model, Design Model, and so on. We generally produce "just enough" to ensure the architectural integrity of the system as a whole. Below, we will discuss the models that relate to these architecturally significant elements.

Business modeling

A software development project may undertake business modeling for a number of reasons. For example, team members may want to gain an understanding of current problems in the existing organization and identify areas for potential improvement. In our order processing system example, the team would undertake business modeling to understand the requirements of a new business and identify the business elements that will satisfy these requirements.

Business Use-Case Model

Let's begin with a superordinate Business Use-Case Model that defines the requirements of the business in our example. The various business Actors and business Use Cases in the superordinate Business Use-Case Model are shown in Figure 2. As we can see, a Customer can request information and order goods from the business. We can also see that a Vendor may manufacture products for the business, an Auditor may audit the business, and a Sponsor may review a business case.
In this article we will concentrate on the development of the Order Goods Business Use Case. The (extremely simplified) basic workflow of this Use Case is as follows:

1. The Use Case starts when the Customer initiates placement of an order for products.
2. An appropriate order is created, showing the products and quantities that the Customer purchases.
3. The relevant products are procured.
4. The appropriate quantity of each product is sent to the Customer.
5. The Customer is sent an invoice for the fulfilled order.
6. The Customer pays for the order.
7. The payment is processed.
8. The Use Case ends.

We can also describe this workflow visually, as shown in the activity diagram in Figure 3.
Business Object Model

The superordinate Business Object Model defines the organizational elements (Business Workers and Business Entities) that realize the requirements specified in the superordinate Business Use-Case Model. The Business Use-Case Realization for the basic workflow of the Order Goods Business Use Case is shown in Figure 4. In this figure we can see that:

- A Sales Clerk Business Worker performs the Create Order activity, which results in the creation of an Order Business Entity.
- A Fulfillment Clerk Business Worker performs the Procure Products Activity.
- A Fulfillment Clerk Business Worker also performs the Ship Products Activity that results in fulfillment of the Order Business Entity.
- An Accounting Clerk Business Worker performs the Invoice Customer Activity.
- An Accounting Clerk Business Worker is also responsible for performing the Process Payment Activity.
The participants in this Business Use-Case Realization are shown in Figure 5.

Figure 4: Order Goods Business Use-Case Realization (basic workflow)

Click to enlarge

The participants in this Business Use-Case Realization are shown in Figure 5.

Figure 5: Participants in the Order Goods Business Use-Case Realization (basic workflow)
Specifying requirements

If we have done business modeling, then we can refer to the Business Object Model to define what aspects of it are to be implemented. In this example, we will choose to implement all elements except the Procure Products Activity, which is performed by the Fulfillment Clerk Business Worker. The purpose of the Requirements discipline, then, is to gather, organize, and document requirements for implementing these business elements in such a way that all stakeholders (including developers) understand them.

These implementation requirements differ from the requirements specified in the Business Use-Case Model in three fundamental ways:

1. The requirements in the Business Use-Case Model represent requirements for the business. The requirements we discuss here are for implementing elements that realize the business requirements.
2. The requirements we discuss here address only a subset of the requirements specified in the Business Use-Case Model.
3. The requirements we discuss here are more detailed than the requirements specified in the Business Use-Case Model.

Use-case model

The superordinate Use-Case Model defines the requirements for the superordinate system. Figure 6 shows the Actors and Use Cases in the superordinate Use-Case Model, which are derived as follows:

- There is an Actor for each Business Worker in the superordinate Business Object Model. Keep in mind that a Business Worker may be non-human (i.e., something automated), in which case we would define a system Actor for the corresponding business Actor. For example, if the Sales Clerk Business Worker were actually automated and launched through a browser-based interface, then a Customer system Actor (rather than a Sales Clerk system Actor) would interact with the Create Order Use Case. However, there are no automated workers in our current example.
- There is a Use Case for each Business Worker "role." In other words, there is a Use Case for each Business Use Case in which a Business Worker participates (with the exception of the Procure Products Activity, which we have chosen not to implement).
- Actors and Use Cases are added to support additional system requirements that are not derived directly from the Business Object Model. In our example, Actors and Use Cases have been added to support various administrative functions, together with the bank system and fulfillment system Actors.

At this stage, we also consider the nature of each of the Actors (which, by definition, are external to the system under consideration): whether the Actor represents a person, software, or hardware. In our simple example, the bank system and fulfillment system Actors are external software systems, and all other Actors represent people. Although we may define some constraints on the interface to these Actors (such as the protocols used to access external systems) from a requirements perspective, we have made no decisions about the internal implementation of the order processing system in terms of software, hardware, workers, legacy systems,
packaged applications, reuse potential, and so on. These decisions are made as part of the Analysis and Design discipline.

Figure 6: Superordinate Use-Case Model

An initial draft description of the basic flow of the Create Order Use Case is as follows:

1. The Use Case starts when the Sales Clerk receives a request to place an order from a Customer (who, for example, has contacted the Sales Clerk by phone, email, fax, or post).
2. The Customer details are retrieved.
3. The Customer details, together with relevant details about each order item, are added to the order.
4. A request to place the order is made.
5. Stock levels are checked to ensure that the order can be fulfilled, and the order is placed.
6. The Use Case ends.

The basic flow of the Ship Products Use Case is as follows:

1. The Use Case starts when the Fulfillment Clerk is notified that an order is to be fulfilled.
2. Relevant quantities of each product specified in the order are shipped to the
Customer by the fulfillment system.

The basic flow of the Invoice Customer Use Case is as follows:

1. The Use Case starts when the Accounting Clerk is notified that an order has been fulfilled (and that the Customer should therefore be invoiced).
2. An appropriate invoice is created, based upon the order details, and sent to the Customer (by email or post).
3. The Use Case ends.

The basic flow of the Process Payment Use Case is as follows:

1. The Use Case starts when the Accounting Clerk receives payment details from a Customer (who has contacted the Accounting Clerk by phone, email, fax, or post).
2. The appropriate Customer account is debited by the bank system.
3. The order is marked as complete.
4. The Use Case ends.

The basic flows of each of the remaining Use Cases is straightforward; we will not describe them here because they represent little more than simple data manipulation.

Often, there is an emphasis on specifying nonfunctional requirements, such as performance and cost, particularly if you need to make decisions during Analysis and Design about whether subsystems are best implemented as hardware, software, or workers, in order to satisfy these nonfunctional requirements. However, for the sake of simplicity, we will not explore such nonfunctional requirements in detail in our example.

Analysis and design

Having specified the requirements, we can move on to considering the analysis and design of the solution that satisfies those requirements.

Analysis model

The superordinate Analysis Model provides a "first cut" solution of the superordinate system in terms of analysis classes. It also contains Use-Case Realizations that show how the Analysis Classes realize each of the superordinate Use Cases. Remember that the objective is not to define an extremely detailed superordinate Analysis Model. Rather, the intention is to undertake sufficient development to address the architecturally significant aspects of the Analysis Model.

Figure 7 shows, as an example, the Use-Case Realization (basic flow) of the Create Order Use Case. We can see from this figure that a number of Entity Classes have been identified: Customer, Order, and Order Item. We can also see a Create Order Form Boundary Class that allows the Sales Clerk to initiate the Create Order Use Case, and a Create Order Controller Control Class that coordinates execution of the Create Order Use Case. Finally, we see the addition of a Fulfillment System Boundary Class that represents the interface to a fulfillment system (and allows determination
Performing this exercise for all of the architecturally significant Use Cases results in the Analysis Model shown in Figure 9. Many of the Analysis Classes shown in this figure have therefore been determined by realizing other Use Cases of the superordinate Use-Case Model.
Figure 9: Superordinate analysis model

Design model

The superordinate Design Model defines the design elements that compose the solution. This model is often only detailed enough to allow us to identify, and assign responsibilities to, the design subsystems that make up our "system of systems" (and that may each be treated as a subordinate system). For example, a subsystem labeled Account Management in the superordinate Design Model may result in the development of this element as a subordinate system. However, aspects of the superordinate system may be designed in detail when these aspects are implemented as part of superordinate system development. This may occur for two reasons:

- To validate the architecture of the superordinate system from both a business and technical perspective.

and/or

- To provide a set of reusable elements to the developers of the subordinate systems.

The transformation from the Analysis Model to the Design Model is driven by various heuristics and guidelines (and the application of various architectural patterns). The transformation also acknowledges any constraints imposed on the model, such as the use of a particular implementation technology (e.g., J2EE or .NET).

In this article we apply a simple "rule" when moving from the Analysis Model to the Design Model: All Analysis Classes that belong to the same functionally related area are grouped together within a large-grained design subsystem. Figure 10 shows how we have applied this rule to identify large-grained design subsystems in the superordinate Design Model. Our decisions are also based upon an element of reuse; arguably, the Customer Management, Account Management, and Fulfillment design subsystems are more reusable than the Order Entry, Accounting, and Shipping Design subsystems.
The resulting design subsystems, along with their associated interfaces, are shown in Figure 11.

Similar to what we did for the Analysis Model, we create Use-Case Realizations in the Design Model. However, now we will show how design elements (rather than Analysis Classes) are used to realize each architecturally significant Use Case. Figure 12 shows the Use-Case Realization of the Create Order Use Case basic flow of events, in terms of our design subsystems.
Performing this exercise for all of the architecturally significant Use Cases results in the assignment of responsibilities to each of the design subsystems, as shown in Figure 13.

The specification of a design subsystem is particularly important when the design subsystem may be treated as a subordinate system in its own right. This specification essentially forms the requirements associated with the subordinate system. In RUP for Systems Engineering®, or RUP SE,® there is particular emphasis on the realization of nonfunctional as well as functional requirements when developing the superordinate Design Model.
Moving from superordinate to subordinate system development

Readers familiar with RUP will notice that developing a superordinate system simply requires applying appropriate parts of the current version of RUP. As we shall see, the development of a subordinate system also follows RUP closely, albeit with certain additional constraints (enforced by the superordinate system). Let's look at the boundary between a subordinate system and its superordinate system, since this boundary is not defined in the current version of RUP.

As we mentioned in Part I, there are many reasons for treating a superordinate system as a system of systems, especially when there is a need to manage complexity. Another reason is that different subsystems might require team members with very different skills (e.g., software engineers versus hardware engineers). Whatever the reason, it is the superordinate system that provides the architectural integrity across the subordinate systems.

Assuming that our superordinate system requires decomposition into a number of subordinate systems, we need to identify those subordinate systems, which we can do by referring to the superordinate Design Model. As we have seen, a design subsystem is an obvious modeling element to consider when identifying subordinate systems. Design subsystems are essentially encapsulated "black boxes" with well-defined interfaces. In our example, all of the design subsystems are potential candidates for subordinate systems.

For our example, we will consider the development of the Account Management design subsystem as a subordinate software system. However, it is worth noting that we might choose to implement one or more of the design subsystems using legacy systems, packaged applications, hardware, people, and so on. The rationale for choosing how to implement each design subsystem typically depends on both business factors (such as the cost of implementation) and technical factors (such as performance).

Table 1 indicates which of the superordinate design subsystems contribute to the realization of each of the superordinate Use Cases. We can determine this by examining each Use-Case Realization in the superordinate Design Model. As we can see, the Account Management design subsystem contributes to the realization of the Create Order, Invoice Customer, and Process Payment superordinate Use Cases.

| Table 1: Relationship between superordinate design subsystems and superordinate Use Cases |
Subordinate system development

As we noted, typically the development of a subordinate system focuses on implementing one of the design subsystems identified in the superordinate system. We will follow the development of the Account Management superordinate design subsystem.

Requirements

A good question at this point is, "Why do we do any requirements work at all for the subordinate systems?" Surely we've already identified requirements for the subordinate system during development of the superordinate system. However, as we have already noted, development of the superordinate system tends to concentrate only on architecturally significant aspects of the superordinate system. Typically the models for that system do not provide the level of detail required to develop each of the subordinate systems. A systematic approach to defining this detail is, of course, to use the RUP.

The superordinate system artifacts do impose constraints on the development of each subordinate system, thereby ensuring the architectural integrity of the superordinate system as a whole. We shall see that as we consider requirements and analysis and design, we must acknowledge these constraints as we provide the detail necessary to implement the subordinate system.

Use-case model

The subordinate Use-Case Model contains the Actors and Use Cases that define requirements on the subordinate system. In this article, we take a simple approach to defining these requirements: We create a single subordinate Use Case for each superordinate Use Case in which the subordinate system participates. However, in some instances it may also be appropriate (or even necessary) to create multiple Use Cases, as we shall see shortly.
Figure 14 shows the Actors and Use Cases defined for the Account Management subordinate system. Often, a subordinate Use Case derived from a superordinate Use Case is given the same name as the superordinate Use Case; this name usually makes sense in the context of the subordinate system (even if the subordinate system only addresses a subset of the associated superordinate Use Case). However, it is important to remember that these subordinate Use Cases are not, of course, the same as those in the superordinate Use-Case Model.

In Figure 14, the Actors represent the Actors in either the superordinate Use-Case Model or other subordinate systems with which this subordinate system interacts. In this example, the name of any Actor that represents another subordinate system concludes with the word System (e.g., Order Entry System).

Figure 14: Subordinate use-case model

If a subordinate Use Case represents only a segment of a superordinate Use Case, then we can derive this segment from the appropriate Use-Case Realization in the superordinate Design Model (in particular, the interaction diagram associated with the Use-Case Realization, since it indicates in which aspects of the superordinate Use Case this subordinate system participates). Figure 15 compares the Create Order superordinate Use-Case description and the Create Order subordinate Use-Case description. We can see that the Create Order subordinate Use Case description addresses only a segment of the Create Order superordinate Use-Case description, and it is also more detailed (i.e., the underlined text has no corresponding text in the superordinate Use Case).
The practice of creating a single subordinate Use Case for each subordinate Use Case in which the subordinate system participates is a very general principle; it is not practical if, for example, the subordinate system implements a disjointed set of steps in the corresponding superordinate Use Case. We can see this in Figure 16, which shows two subordinate Use Cases for the corresponding superordinate Use Case.

**Analysis and design**

We use the Analysis and Design discipline to consider the solution that will satisfy the stated requirements. In this article we do not describe the pros and cons of creating an Analysis Model as well as a Design Model. However, we will create a subordinate Analysis Model to demonstrate relationships with the elements of the superordinate Analysis Model.

**Analysis model**

Figure 17 shows the Use-Case realization of the Create Order Use Case basic flow of events. We can see that the Order, Order Item, and Bank System Classes from the
superordinate Analysis Model are used. We can also see the introduction of Create Order API, Create Order Controller, and Order Book Analysis Classes that are specific to the Account Management subordinate system. The Create Order API Class represents a programmatic interface to the Create Order functionality that this subordinate system provides. We can also see Boundary Classes representing the Fulfillment subordinate system. In this and subsequent figures, the shaded items are those that are "givens" (since they are defined by the superordinate system), and that make it easy to see where we are "plugging gaps" in the superordinate system.

Even though a Class such as the Create Order Controller has a counterpart with the same name in the superordinate Analysis Model, the two classes are different for the same reason that a subordinate Use Case is not the same as its superordinate counterpart. The Analysis Class in the subordinate system may fulfill only a subset of the responsibilities of the Analysis Class in the superordinate system (since the subordinate Use Case may implement only a subset of the flows of events of the superordinate Use Case). The Analysis Class in the subordinate system also needs to address any detail added to the Use Cases of the subordinate system.

Figure 17 also allows us to make two more observations. The first is that it is fairly easy to see, given the example, how a subordinate system essentially fills the gaps in the superordinate system (indicated by those elements that are not derived from the superordinate system) and is also constrained by the superordinate system (indicated by those elements that are derived from the superordinate system). We can also observe that, on occasion, the subordinate system development may refine the definition of any superordinate system elements. In this example, we have assigned a responsibility (Obtain Credit Authorization) to the superordinate Bank System Boundary Class, an element in the superordinate Analysis Model.

Figure 18 is a diagram of participants corresponding to the elements shown in Figure 17.
Performing this exercise for all of the Account Management Use Cases results in the Analysis Model shown in Figure 19.

**Figure 18: Participants in the Create Order Use-Case Realization (basic flow)**

**Figure 19: Analysis Model of the Accounting subordinate system**

*Click to enlarge*
Just as for the superordinate system, the transformation from the subordinate Analysis Model to the subordinate Design Model is driven by various heuristics and guidelines. Here are those applied within our Account Management subordinate system:

- A Boundary Class that represents an external subordinate system (such as the Fulfillment Boundary Class) is represented as the interface specified in the superordinate Design Model.
- A Boundary Class that represents an external system (such as the Bank System Boundary Class) is represented as a design subsystem.
- Analysis Classes that are tightly coupled (such as Order Book, Order, and Order Item) are grouped together in a design subsystem.
- All other Analysis Classes will have a one-to-one correspondence with a Design Class.

Figure 20 indicates the result of applying this guidance. Had any detailed design decisions been made within the superordinate system, then this Design Model would acknowledge those decisions.

Figure 20: Identifying design elements

The resulting initial Design Model is shown in Figure 21. We can observe that each of the Classes representing an API will realize one or more of the operations defined on the interface that is realized by the Account Management subordinate system.
Similarly to how we created Use-Case Realizations in the Analysis Model, we create Use-Case Realizations in the Design Model. But now we show how design elements (rather than analysis elements) are used to realize each architecturally significant Use Case. Figure 22 shows the Use-Case realization of the Create Order Use-Case basic flow of events.

Performing this exercise for all of the Use Cases results in the Design Model shown in Figure 23.
Summary

In Part I of this series, we described how RUP can be applied to a diverse set of initiatives, and the role of the System of Interconnected Systems Pattern in managing the complexity of such initiatives. In this part, we showed how to apply the Pattern to the development of an aspect of a system -- in this case an order processing system.

In conclusion, we want to emphasize that our intent was to describe the principles involved in applying RUP in conjunction with the System of Interconnected Systems Pattern. We did not include detailed process guidance, and again, we refer the reader to RUP SE to learn more about systems engineering, and to Jacobson et al. (see References) for more information and discussion on strategic reuse.

Acknowledgments

The authors would like to thank the following people for their help and guidance in writing this article series: Kurt Bittner, Roger Bowser, Dave Brown, Murray Cantor, Kelli Houston, Ivar Jacobson, Paula Simmonds, John Smith, and Dave West, all of IBM Rational; Christina Cooper-Bland of BACS; Jon Pidgeon of Lloyds TSB; Alan Whitfield of UK Inland Revenue; and Gary Willcocks of EDS.

References

The following references were used in preparing this paper, including Part I.


Rational Unified Process, version 2002.05. IBM Rational Software.
The Rational Edge -- October 2003 -- Modeling for enterprise initiatives with IBM Rational Unified Process


Appendix: Key RUP artifacts and their relationships

Table A-1 shows a number of key RUP artifacts (and the relationships between them) that are discussed in this article series. The purpose of this series is to describe the principles that underpin the Systems of Interconnected Systems Pattern when applied in conjunction with RUP. We have chosen, therefore, to omit certain artifacts from our descriptions, including the Business Glossary, Business Vision, Supplementary Business Specification, Business Architecture Document, Glossary, Vision, Supplementary Specification, Deployment Model, Data Model, and Software Architecture Document.

Superordinate business modeling artifacts

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Description</th>
<th>Dependent upon</th>
<th>Description of relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Use-Case Model</td>
<td>Defines the requirements of the business.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Business Object Model</td>
<td>Defines the organizational elements (Business Workers and Business Entities) that realize the requirements specified in the superordinate Business Use-Case Model.</td>
<td>Superordinate Business Use-Case Model</td>
<td>The superordinate Business Object Model contains Business Use-Case Realizations that show how the Business Workers and Business Entities of the superordinate Business Object Model fulfill the requirements specified in Use Cases of the superordinate Business Use-Case Model.</td>
</tr>
</tbody>
</table>

Superordinate requirements artifacts
### Superordinate analysis and design artifacts

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Description</th>
<th>Dependent upon</th>
<th>Description of relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Model</td>
<td>Provides a &quot;first cut&quot; solution of the superordinate system in terms of Analysis Classes. Also contains Use-Case Realizations that show how the Analysis Classes realize each of the superordinate Use Cases.</td>
<td>Superordinate Business Object Model</td>
<td>The key abstractions (Entity Analysis Classes) in the superordinate Analysis Model may derive directly from the Business Entities in the superordinate Business Object Model.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Superordinate Use-Case Model</td>
<td>The superordinate Analysis Model contains Use-Case Realizations that show how Analysis Classes fulfill the requirements specified in the superordinate Use-Case Model.</td>
</tr>
<tr>
<td>Design Model</td>
<td>Defines the design elements that compose the solution.</td>
<td>Superordinate Use-Case Model</td>
<td>The superordinate Design Model contains Use-Case Realizations that show how design elements fulfill the requirements specified in the superordinate Use-Case Model.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Superordinate Analysis Model</td>
<td>Design elements are identified from elements in the superordinate Analysis Model, using various heuristics.</td>
</tr>
</tbody>
</table>

### Superordinate implementation artifacts
## Superordinate implementation artifacts

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Description</th>
<th>Dependent upon</th>
<th>Description of relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation Model</td>
<td>Identifies the implementation elements (such as files and directories) associated with the elements of the superordinate system that are implemented.</td>
<td>Superordinate Design Model</td>
<td>Implementation elements are derived from the design elements (such as Design Classes and interfaces) in the superordinate Design Model.</td>
</tr>
</tbody>
</table>

## Superordinate test artifacts

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Description</th>
<th>Dependent upon</th>
<th>Description of relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Various</td>
<td>Superordinate test artifacts test various superordinate elements.</td>
<td>Superordinate Use-Case Model</td>
<td>Superordinate test artifacts test the requirements specified in the superordinate Use-Case Model.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Superordinate Implementation Model</td>
<td>Superordinate test artifacts test the implementation elements of the superordinate Implementation Model.</td>
</tr>
</tbody>
</table>

## Subordinate business modeling artifacts
### Subordinate requirements artifacts

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Description</th>
<th>Dependent upon</th>
<th>Description of relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Business Use-Case Model</strong></td>
<td>Defines the requirements of the business considered in the implementation of this subordinate system.</td>
<td>Superordinate Business Use-Case Model</td>
<td>A subordinate Business Use Case may derive from a superordinate Business Use Case, or a part of a superordinate Business Use Case (since a subordinate system may implement either an entire superordinate Business Use Case or a segment of a superordinate Business Use Case).</td>
</tr>
<tr>
<td><strong>Business Object Model</strong></td>
<td>Defines the organizational elements (Business Workers and Business Entities) that realize the requirements specified in the subordinate Business Use-Case Model.</td>
<td>Superordinate Business Object Model</td>
<td>The Business Workers and Business Entities in the subordinate Business Object Model include Business Workers and Business Entities from the superordinate Business Object Model.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subordinate Business Use-Case Model</td>
<td>The subordinate Business Object Model contains Business Use-Case Realizations that show how the Business Workers and Business Entities of the subordinate Business Object Model fulfill the requirements specified in Use Cases of the subordinate Business Use-Case Model.</td>
</tr>
</tbody>
</table>
Superordinate Design Model | If a subordinate system has a dependency on another subordinate system (represented as a dependency from one design subsystem to the interface of another design subsystem in the superordinate Design Model), then that subordinate system is represented as an Actor in the subordinate Use-Case Model (since it is considered to be an external system).

Subordinate Business Object Model | The subordinate Use-Case Model identifies the requirements on those aspects of the subordinate Business Object Model that are to be implemented and takes the definitions of the relevant Business Workers and Business Entities as input.

### Subordinate analysis and design artifacts

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Description</th>
<th>Dependent upon</th>
<th>Description of relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Model</td>
<td>Provides a &quot;first cut&quot; solution of the subordinate system in terms of Analysis Classes and Use-Case Realizations that show how the Analysis Classes realize each of the subordinate Use Cases.</td>
<td>Superordinate Analysis Model</td>
<td>The key abstractions in the subordinate Analysis Model include key abstractions in the superordinate Analysis Model, as well as those that are defined specifically for the subordinate system (which are, by definition, specific to the subordinate system and therefore less reusable than those &quot;imported&quot; from the superordinate Analysis Model).</td>
</tr>
<tr>
<td>Superordinate Analysis Model</td>
<td></td>
<td></td>
<td>If a subordinate system has a dependency on another subordinate system (represented as a dependency from one design subsystem to the interface of another design subsystem in the superordinate Design Model), then that subordinate system is represented as a Boundary.</td>
</tr>
<tr>
<td>Class in the subordinate Analysis Model (since it is considered to be an external system). Although the Boundary Class can be derived from the presence of an Actor in the subordinate Use-Case Model, it is likely that the superordinate Design Model contains additional information (such as the detailed responsibilities of the design subsystem that represents the &quot;external&quot; subordinate system) that is relevant to the subordinate Analysis Model.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Subordinate Business Object Model</strong></td>
<td>The key abstractions <em>(Entity Analysis Classes)</em> in the subordinate Analysis Model may derive directly from the Business Entities in the subordinate Business Object Model.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Subordinate Use-Case Model</strong></td>
<td>The subordinate Analysis Model contains Use-Case Realizations that show how Analysis Classes fulfill the requirements specified in the subordinate Use-Case Model.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Design Model</strong></td>
<td>Defines the design elements that comprise the solution.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Superordinate Design Model</strong></td>
<td>The subordinate Design Model is dependent on the superordinate Design Model, in that it must expose the appropriate interface as defined in the superordinate system. It also uses the interfaces defined for any external subordinate systems that are used, which are also defined in the superordinate system. It may also make use of any detailed design elements (such as Design Classes) that may have been defined in the superordinate Design Model.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Subordinate Use-Case Model</strong></td>
<td>The subordinate Design Model contains Use-Case Realizations that show how design elements fulfill the requirements specified in the subordinate Use-Case Model.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Subordinate implementation artifacts

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Description</th>
<th>Dependent upon</th>
<th>Description of relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation Model</td>
<td>Identifies the implementation elements in the subordinate system.</td>
<td>Superordinate Implementation Model</td>
<td>The subordinate Implementation Model may reference elements of the superordinate Implementation Model (such as an included file).</td>
</tr>
<tr>
<td>Subordinate Design Model</td>
<td></td>
<td>Implementation elements are derived from the design elements (such as Design Classes and interfaces) in the subordinate Design Model.</td>
<td></td>
</tr>
</tbody>
</table>

Subordinate test artifacts

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Description</th>
<th>Dependent upon</th>
<th>Description of relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Various</td>
<td>The subordinate test artifacts test various subordinate elements.</td>
<td>Superordinate test artifacts</td>
<td>The subordinate test artifacts support the overall test strategy defined within the test plan associated with the superordinate system.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subordinate Use-Case Model</td>
<td>The subordinate test artifacts test the requirements specified in the subordinate Use-Case Model.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subordinate Implementation Model</td>
<td>The subordinate test artifacts test the implementation elements of the subordinate Implementation Model.</td>
</tr>
</tbody>
</table>

Notes

1 For the sake of simplicity, we will not discuss certain key RUP artifacts in this article (e.g., the Supplementary Specification).

2 A business Actor is someone or something external to the business that interacts with the business. A business Use Case is a sequence of actions performed by a business that yields an observable result of value to a particular business Actor.
In conjunction with the superordinate Supplementary Specification.

The convention for representing a responsibility is to prefix the name of an operation with "/".

The superordinate Design Model is especially important when adopting a common technical architecture, since the model identifies the set of architectural mechanisms (such as persistency, security, and communication) to be used.

This is discussed in some detail in Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process, and Organization for Business Success*. Addison-Wesley, 1997.

See Jacobson et al. for a detailed discussion of the concepts application system and component system, which provide different levels of reuse.

We encourage the reader to consult Murray Cantor's IBM Rational whitepaper, "Rational Unified Process for Systems Engineering" (available at [www.rational.com/products/whitepapers/index_all.jsp](http://www.rational.com/products/whitepapers/index_all.jsp)) and the RUP SE Plug-In (available at [http://www.rational.net](http://www.rational.net); authorization required) for more information.

Such factors are considered in some detail in the RUP SE Plug-In (see Note 9).

In conjunction with the subordinate Supplementary Specification.

This is discussed in IBM Rational Unified Process, Version 2002.05.

API stands for Application Programming Interface; the Account Management subordinate system can be interacted with programmatically. In fact, this system never interacts with a person, and so does not contain any user interface elements.

Such heuristics and guidelines are often technology dependent. For simplicity, this article does not take such dependencies into account.

---

*For more information on the products or services discussed in this article, please click [here](http://www.rational.com) and follow the instructions provided. Thank you!*

*Copyright [Rational Software](http://www.rational.com) 2003 | [Privacy/Legal Information](http://www.rational.com/privacy.html)*