Model Structure Guidelines For Rational Software Modeler And Rational Software Architect (2004 Release)

White Paper

Bill Smith, Model Driven Development, IBM Rational Software

V1.0
September 8, 2004

Table of Contents

1. Introduction

Intended Audience

The paper is designed to support users of the Rational Software Architect (RSA) product who are interested in applying the guidance found in the Rational Unified Process (RUP®) to their use of RSA. If you are a user of Rational Software Modeler (RSM), you will also find the paper useful but should be aware that some sections reflect capabilities available only in RSA and not in RSM. The paper is directed primarily to the case where you are creating a new set of models in RSA. If you are new to RSA but were previously a user of Rational Rose or Rational XDE and you have imported models from those products, you might find some value in the paper as a source of guidance for restructuring your imported models.

The paper assumes that you have fairly broad knowledge of UML, as well as basic familiarity with the fundamental concepts and theory of operations of RSA, in particular modeling, transformations, and visual code editing.

Purpose

The RUP describes a set of models that represent well-defined perspectives on the problem and solution domains of systems. The utility of this set of model structures has been proven in many real-world projects, and regardless of whether you follow a formal process (such as RUP), these model structures are worth considering. This document discusses how to realize the RUP model artifacts using RSA.

As the title suggests, the project and model structures described in this paper are guidelines, not imperatives. Whether you decide to model a particular RUP artifact in RSA is a consideration of your own development process and is often a project-specific decision. One should also keep in mind that RUP is not a rigid set of process rules. It is a process *framework* within which to formulate process definitions that can range from the very formal to the very lightweight.

The way one uses UML can also range from very formal to very informal. You might choose to treat your UML models like formal architectural drawings that are to be strictly followed during construction, or you might treat your models more like sketches that suggest the broad outlines of a design, but are considered disposable once the project moves into implementation. RSA can support you at either end of the process and modeling spectra. From that standpoint, the guidelines presented in this paper are not

intended to shackle your thinking, but to help you understand how to use the features of RSA to facilitate the process that seems best to you.

Note that RSA makes it possible to use models not just as blueprints, but as the specification of a solution from which implementations can be automatically generated. This is accomplished using RSA model-to-model and model-to-code transformations. The use of RSA transformations to practice Model-Driven Development (MDD) introduces special concerns regarding model structures. If you will be using RSA models and transformations to practice Model-Driven Development, you should also consult the various RSA MDD-specific resources available on Developer Works.

Scope

This paper describes how to represent the RUP model artifacts in RSA and offers guidelines for the internal organizational structures of those artifacts. It does *not* attempt to:

- Restate the conceptual underpinnings of the RUP model artifacts, nor provide extensive descriptions of them
- Describe the process or techniques for specifying the detailed semantic or diagrammatic content of the associated RUP artifacts

For tool-neutral information on how to define, develop, and model the contents of the RUP artifacts, see RUP.

For information on tool-specific techniques for developing the content of RSA models see:

- The product documentation (tutorials, samples, online help)
- The tool mentors in the RSA-specific RUP configuration, of which this whitepaper is a part
- RSA-related resources on Developer Works

Typographic Conventions

Discussions that are of interest to users of RSA who are migrating from IBM Rational Rose or XDE are presented in sidebar fashion, within a bordered text box with shaded background:

> ***XDE/Rose***
> Discussion of interest to previous XDE or Rose users.

Organization of the Paper

The Basic Concepts and Terminology section that follows establishes a working vocabulary and provides some general information about how models are implemented in the RSA product.

Next, the RUP Model to RSA Model Mapping section discusses how RSA supports the model types defined by RUP.

Following that are several sections that provide guidance for structuring models of various types in RSA. Some of these sections discuss different ways of using a type of model depending upon how much rigor you prefer in terms of your process, modeling approach, and architectural control.

Finally, there is a discussion covering various issues associated with factoring models into multiple modeling files. This touches on strategies for managing scale and enabling sharing of models among team members in order to minimize file contention and conflict merges.

2. Basic Concepts and Terminology

Models

In RUP, a model is defined as a complete specification of a problem or solution domain from a particular perspective. A problem domain or a system may be specified by a number of models that represent different perspectives on the domain or system. RUP proposes a specific set of models:

- Business Use-Case Model
- Business Analysis Model
- Use-Case Model
- Analysis Model (may be subsumed in Design Model)
- Design Model
- Implementation Model
- Deployment Model
- Data Model

Note that RUP itself is tool-agnostic. As far as RUP is concerned, a model could be a drawing on a napkin or on a whiteboard, something in a modeling tool, or even a mental image. So from the RUP perspective, a model is a logical concept. In the context of RSA, we can discuss models in logical terms, but we can also discuss them in physical terms.

Suppose you have teams working on two applications: a team of three analysts working on a timesheet management application, and a second team of five analysts working on a call center application. Both teams are currently working to capture requirements and are using RSA for use case modeling. In RUP terms you would say that one team is building "the use-case model for the timesheet application" and the other is building "the use-case model for the call center application." But if the teams are using RSA, it is important to recognize that their models have particular physical manifestations. That is the subject of the next section.

Modeling Files

In RSA, models are persisted as files. (In Eclipse terminology, a file is considered a 'resource'[1], so if you encounter the term 'modeling resource' in this paper or in other sources, it means the same thing as 'modeling file'). In the broadest sense, RSA supports two kinds of modeling files:

- "Pre-Implementation" modeling files. These files contain conceptual UML content that does not *directly* reflect implementation artifacts. These files contain both the UML semantics of the model, and UML diagrams that depict the semantic elements.

- Implementation modeling files (Eclipse resources), which are normal implementation artifacts such as Java source files or Web pages and which reside in an Eclipse project. In this case, you can think of the project as representing the scope of the content of the implementation modeling file. The model semantics reside in the implementation artifacts themselves. Each diagram resides in its own physical file within the project. These diagrams may use UML notation, but they may also use other notations

[1] In Eclipse a resource is a file, but also has additional properties and behavior within the Eclipse environment. The Modeling Files described here are treated as 'resources' by Eclipse.

(e.g., IDEF1X or Information Engineering notations for data visualization, or a Rational proprietary notation used for designing Web tiers). The UML diagram types that are supported for depicting 3GL code artifacts include Class diagrams and (for Java only) Sequence diagrams.

The focus of this paper is how to organize the internal structures of "pre-implementation" models, and **within the paper the term "modeling file" is reserved for files that contain "pre-implementation" model content**. Guidance for organizing the contents of implementation projects can be found in other sources such as online help for Rational Software Architect, Rational Application Developer, and Rational Web Developer.

A "pre-implementation" modeling file does not necessarily contain all of the information for one model. In fact, it will often be the case that a modeling file contains only a subset of a model. For instance, in the example given above where we have a team of three working on a use-case model for a timesheet application, the team might choose to physically partition its use-case model into three modeling files so that each member of the team can work on a different subset of the use cases without contending for the same file. The final section of the paper discusses the issues associated with partitioning models and managing modeling files.

Model Types

In RUP, models are of specific types such as use-case model, analysis model, or data model. In RSA you can think of a modeling file as having a type: the type of the model (or model subset) the file contains. A modeling file's type can be established in either of two ways:

- Start with a "Blank" modeling file (see below) and then establish its type simply by how you choose to name it and what kind of content you place into it (including what UML profiles you apply to it).
- Create it based upon a pre-defined "template model" that represents a particular model type. Notice that the "type" of a modeling file is really just a convention concerning the content of the file. For example, the tool will not prevent a file that contains a use-case model from also containing the classes that realize the use cases. However, these guidelines recommend that you treat model files as being typed.

Workspaces, Projects and Project Types

Readers familiar with Eclipse, WebSphere Studio products, or Rational Application Developer will already know that files reside within Projects, that Projects can be of various types, and that Projects are (virtually) grouped and managed within Workspaces. RSA modeling files reside within projects just like other files.

For purposes of this discussion, not all of the project types that are available in Rational Software Architect, Rational Application Developer, and Rational Software Modeler need be explained in detail. In the broadest terms, we are interested in two categories of projects:

- **UML projects**
- **Implementation projects**, which include the specialized types such as Enterprise Project, EJB Project, Web Project, and C++ Project

We stated earlier that RSA supports two kinds of modeling files:

- Files that contain UML models (semantic elements plus diagrams depicting them) used for modeling at levels of abstraction above implementation (such as requirements, analysis, and design)
- Projects that contain implementation artifacts such as Java code or Web pages plus (optional) diagram files that contain diagrams depicting the implementation artifacts.

The rule for allocating models to projects is simple: **a)** place "pre-implementation" modeling files in UML projects, and **b)** implementation models take care of themselves because in essence:

[implementation model] = [implementation project]

There are some exceptions to this rule. The following UML modeling files are candidates for placement within a language-specific implementation project:

- Design 'sketch models' (which will be discussed in a later section)
- Models with sequence diagrams describing tests that will be executed against the code in the project

> ***XDE/Rose***
> The Rose and XDE theories of operation include the practice of iteratively refining the Design Model until you reach a code-equivalent level of abstraction, and then using code-model synchronization technology to keep the semantics of that model in agreement with the code itself. For instance, in XDE, implementation models exist not just as code and diagrams in projects, but also as 'code model' files that are persisted independently of the implementation artifacts and in essence represent a redundant copy of their semantics.
>
> The RSA theory of operations encourages the use of platform-neutral models at levels of abstraction higher than code (i.e., design models such as an Enterprise IT Design Model) and the use of transformations to generate code from those models. Then, at the code level of abstraction, RSA simply lets you draw UML diagrams of code, and dispenses with the approach of using a separately persisted semantic model.
>
> Note that RSA does not *prevent* you from defining UML models at a code level of abstraction and generating code from them; in fact, this type of usage is expected. But RSA does not provide technology for keeping such models synchronized with code. This type of usage generally corresponds to a very 'lightweight' variant of RUP wherein the design model is considered dispensable once code has been generated.

Concepts in Review

The following illustrations summarize the preceding discussions. The illustrations reflect the scenario described earlier, where we have two teams, one working on a call center application and another working on a timesheet management application.

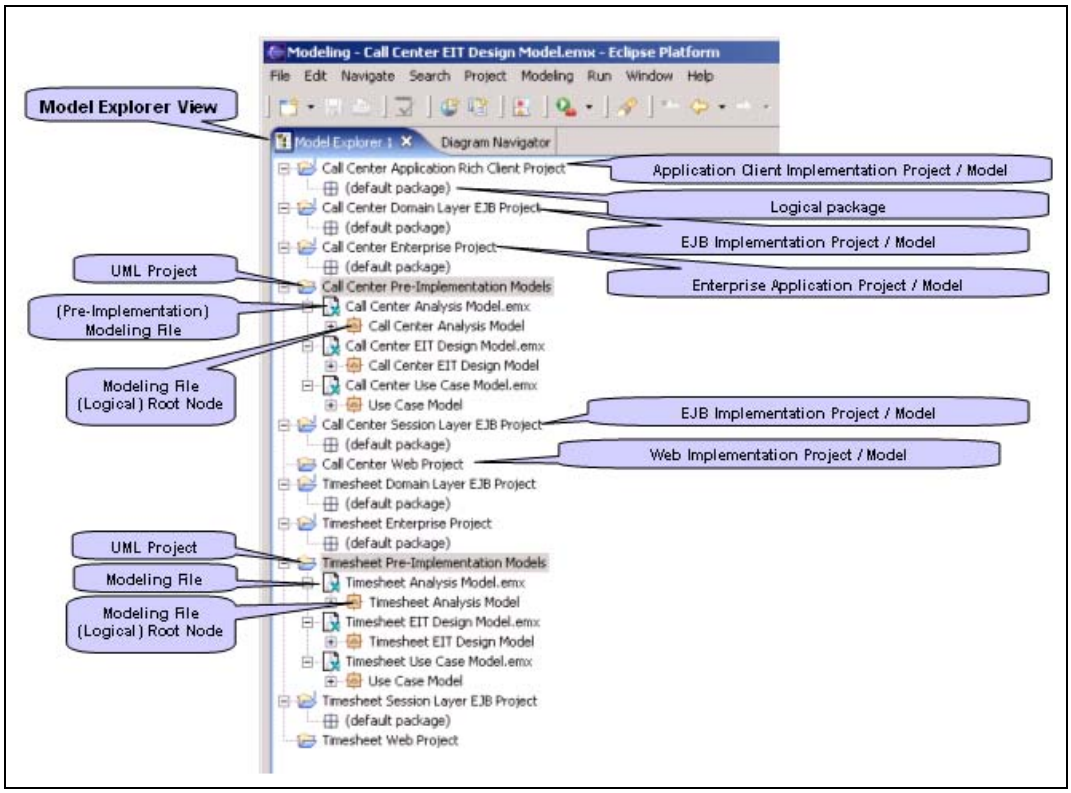


Figure 2-1

RSA provides a Model Explorer view, which provides a combined physical and logical view of models. In the Model Explorer you see the projects in your workspace depicted as top-level nodes, and within each project you see the resources that belong to that project. **Figure 2-1** depicts in the Model Explorer a collection of projects that correspond to the two applications in our scenario. The UML projects have been used for the pre-implementation models. A collection of other projects of solution-appropriate types (such as Enterprise Application projects, Web projects, and so on) has been used for the implementation models.

> ***XDE/Rose***
> In contrast to the RSA model explorer, the model explorers in Rose and XDE provided only a logical view of models. Note that the view of resources provided by the RSA Model Explorer is not the 'pure' physical view provided by the Eclipse Navigator view. While some physical resources are visible in the Model Explorer, they are mostly represented by icons that indicate *logical* views of the resources.

Figure 2-2 shows how the Timesheet use-case model might be internally organized into packages that represent some functionally cohesive subsets of the problem domain.

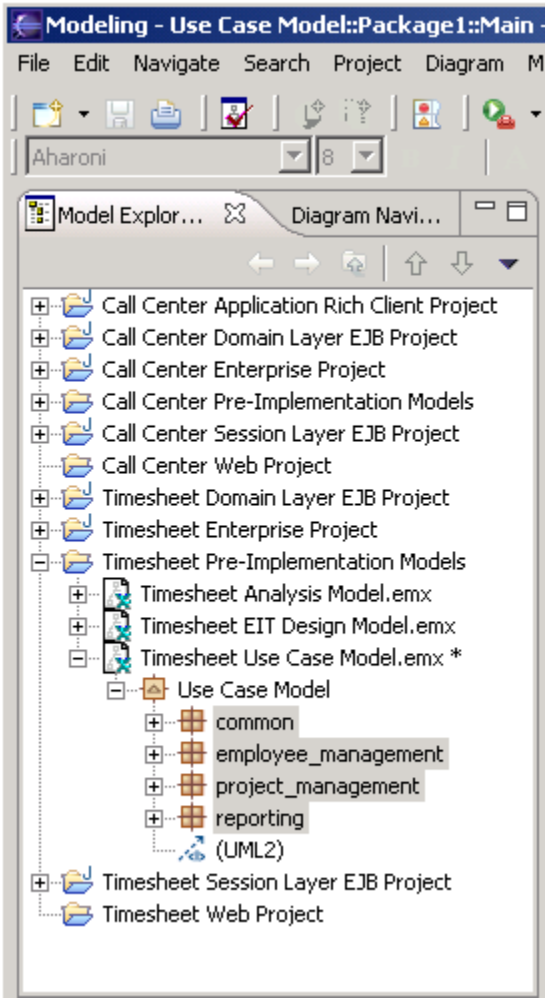


Figure 2-2

In **Figure 2-1** and **2-2** each of the pre-implementation models resides in a single modeling file. Alternatively any of them could be refactored into multiple modeling files. For instance the Timesheet use-case model could be refactored into four modeling files where each corresponds to one of the depicted subsets of the problem domain ("common", "employee_management", "project_management", and "reporting"). In that case the root node of each modeling file would be named to maintain a consistent namespace across all of the modeling files that make up the complete use-case model. For example the root nodes of the four modeling files might be "timesheet.requirements.common", "timesheet.requirements.employee_management", "timesheet.requirements.project_management" and "timesheet.requirements.reporting".

3. RUP Model to RSA Model Mapping

The following table shows how the most commonly used RUP models map to RSA model types. The mapping is generally straightforward, but it is the key to using this paper as a guide to practicing RUP with RSA. The RSA model types mentioned in the table are discussed immediately following the table. Guidelines for internal organization of the various model types, and what kinds of projects to keep them in, are provided in later sections. Those later discussions are presented in terms of the RSA model types listed here.

RUP Model	RSA Model Type
Use-Case Model	Use-Case Model
Analysis Model	Analysis Model (alternate: «analysis» packages in Design model)
Design Model	For n-tier business applications: Enterprise IT Design Model For other types of applications: Blank Model used as design model For design 'sketch': Blank model used as design 'sketch' model Optional supplement: Blank model used as Implementation Overview Model
Implementation Model	Implementation projects containing implementation artifacts and diagram files
Deployment Model	Blank model used as deployment model

RSA Model Types

Blank Model

RSA provides the option to create a "Blank Model" (File→New→UML Model→Blank Model). A "Blank Model" is a modeling file that is not based upon a model template. It has no special profiles applied, and no default content other than a single "Main" (freeform) diagram. **You can use a blank modeling file as a starting point for any type of model**. By choosing how you name it, what content you define within it, and what profiles you apply to it, you might use a blank modeling file to build a use-case model, an analysis model, a design model, a deployment model, or any other type of RUP model.

Use-Case Model

RSA provides the option to create a “Use-Case Model” file based upon a model template. The template contributes default content as depicted in ***Figure 3-1***. (It is outside the scope of this document to explain how the “building block” content and search strings are used. The templates contain instructions such that you should find them to be largely self-explanatory.)

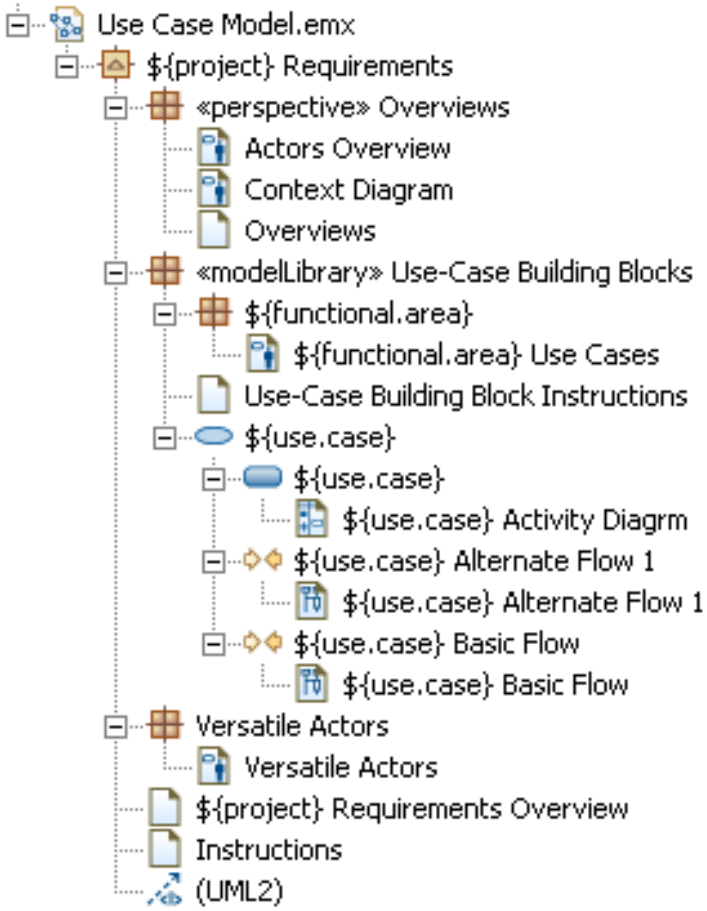


Figure 3-1

Analysis Model

RSA provides the option to create an “Analysis Model” file based upon a model template. The template contributes default content as depicted in ***Figure 3-2***. In addition, an “Analysis” profile is applied to model files created from this template:

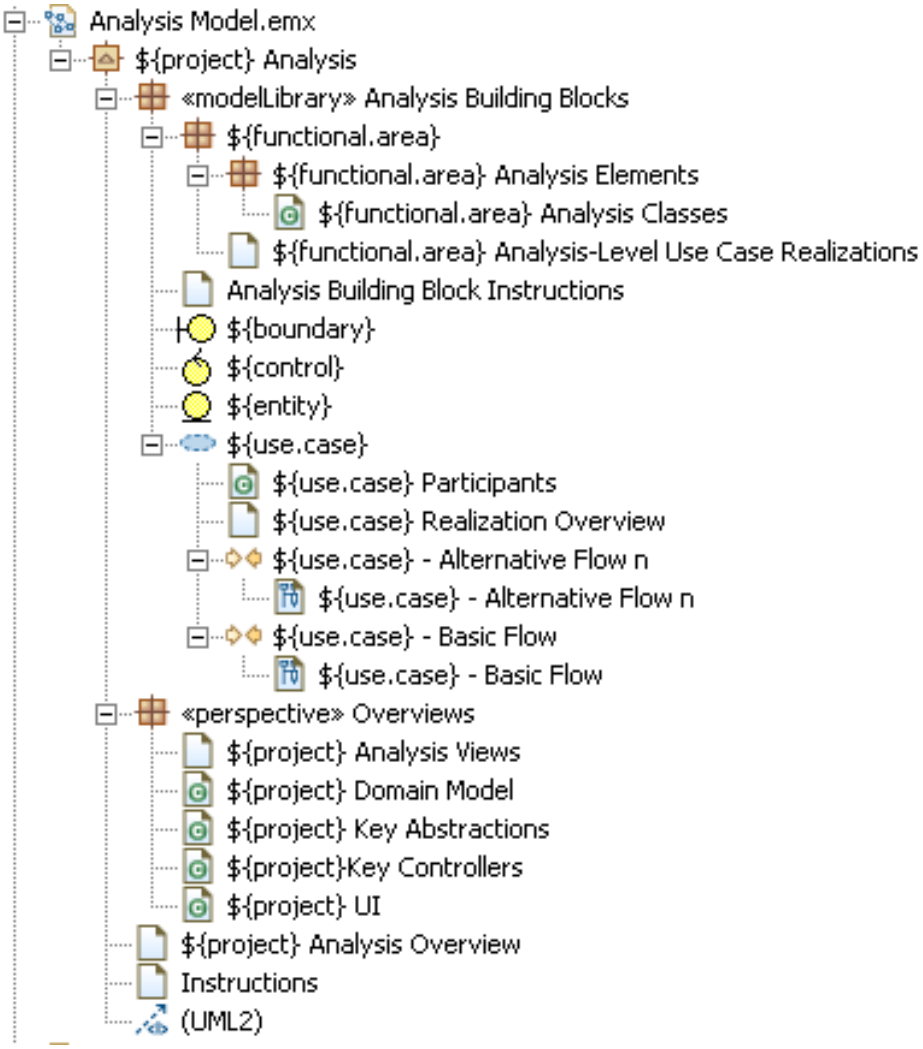


Figure 3-2

Enterprise IT Design Model

RSA provides the option to create an "Enterprise IT Design Model" (EITDM) file based upon a model template. The template provides default content as depicted in **Figure 3-3**. In addition, an "EJB Transformation" profile[2] will be applied to model files created from this template. This is the appropriate template to use for design (and optionally for analysis) when targeting business applications and using RSA code-generating transformations to support creation of such applications.

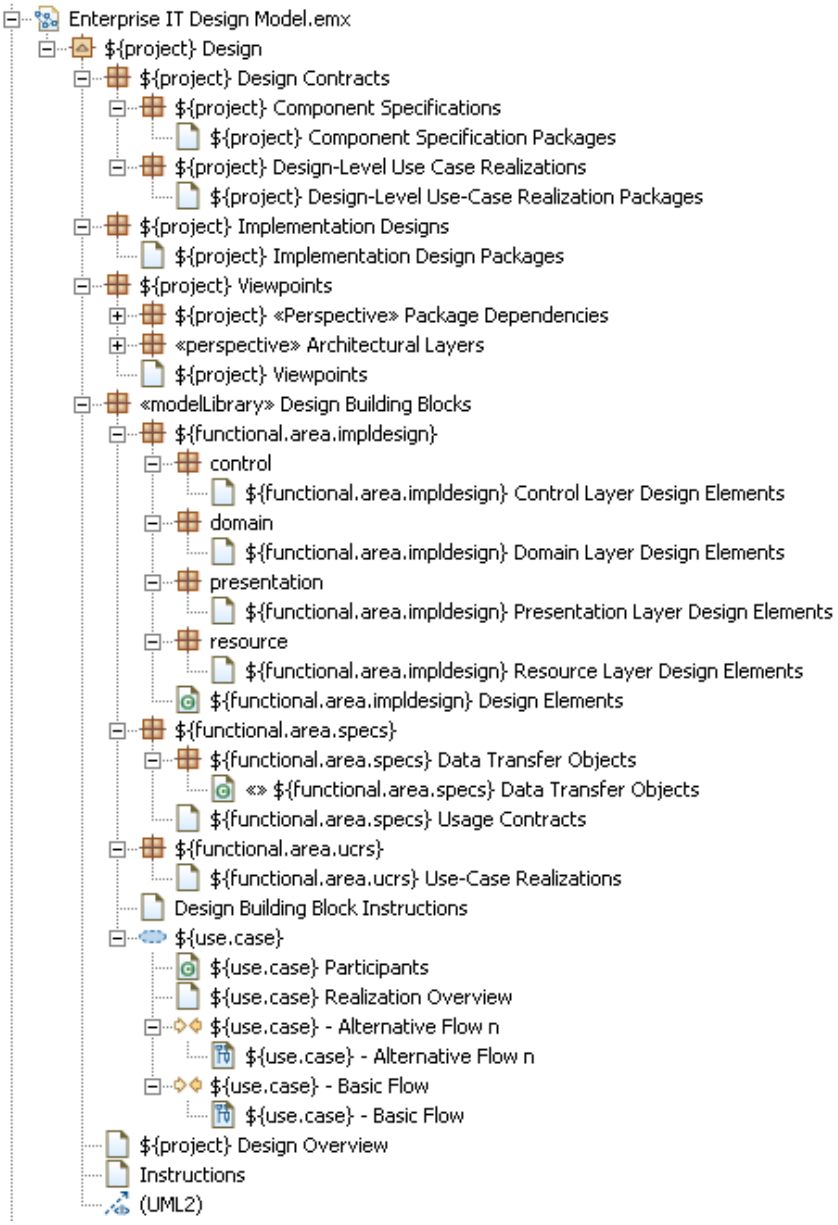


Figure 3-3

[2] The set of transformation-enabling profiles provided as part of the EIT Design Model template is likely to evolve as updates to the product are released.

Implementation Overview Model

As part of your design model, in addition you might find it useful to define an "Implementation Overview Model" to capture a high-level view of how the implementation is to be organized. The "Implementation Overview Model" would be used early in the design phase—before code has been generated or written—to represent the actual RSA projects and folders/packages in which you expect the code and related files (metadata, deployment descriptors, and so on) to reside. You could also use this model to show the anticipated dependencies among those projects and packages, which can prove helpful in identifying system build requirements. The Implementation Overview Model can also be a place to keep informal conceptual diagrams of the solution architecture.

Implementation Model

As previously stated, in RSA an implementation model consists of a project containing implementation artifacts and (optionally) diagrams that depict those artifacts[3].

"Sketch" Models

As noted in the "Basic Concepts and Terminology" section, you might choose to treat design models like formal architectural drawings that are maintained for the lifetime of the system and used to support/enforce architectural control. Or, you might treat them more like sketches that serve to suggest a design and help clarify and communicate it, but are considered disposable once the actual implementation has begun to diverge from the original design. RSA supports both approaches. Its features generally do not specifically target one approach or the other, but your choices about how to use design models certainly help determine what RSA features you will use and how you will use them. Where the distinction becomes important in the context of guidelines presented in this paper, the term "sketch model" will be used to indicate that a model is being used in the more 'disposable' manner.

[3] To create these diagrams, instead of using File→New→UML Model to create a model, you use File→New→Class Diagram to create a diagram in which you can compose 'views' of code in UML (or other) notation. Each individual diagram is persisted as a separate file with an extension of .dnx, and can be version controlled much the same as a code file. These diagrams do not contain any semantic information, just notation. All relevant semantic information resides in the code itself. When you change something such as a class name or operation signature in one of these diagrams, you are actually changing the underlying code itself. When you make such changes in code (using a text editor), the diagrams in which the changed code appears are automatically updated.

4. General Guidelines and Techniques for Organizing Internal Structures of Models

The primary tool for organizing the content of UML models is the package. UML packages serve two primary purposes:

- partitioning, organizing, and labeling model information
    - grouping elements that correspond to a specific subject matter in the problem or solution domain
    - separating different types of model information such as interfaces, implementations, diagrams, and so forth
    - grouping elements in order to define and control their dependencies on other elements
    - grouping diagrams that provide alternative views on the same model

- establishing namespaces
    - for model elements
    - for implementation artifacts generated from model elements (this might involve mappings between model and implementation language namespaces)
    - for a unit of reuse

Traditionally, RUP has proposed specific packaging strategies for various model types. Those strategies are reflected in the model type-specific sections of this paper. RSA also introduces some additional organizational tools, which are described here:

Represent Viewpoints Using «perspective» Packages

In cases where it is desirable to see elements organized in more than one way, you can create additional packages with diagrams that depict the alternate organizational schemes. This same technique can serve anywhere there is a need to represent a particular view on model content that cuts across the model's packaging scheme. RSA supports this technique by providing a «perspective» package stereotype as part of its UML 'base profile'. You can think of a «perspective» package as generally the equivalent of a RUP for Systems Engineering or IEEE 1417 "Viewpoint".

Do not place semantic elements (classes, packages, associations, and so on) within «perspective» packages. Just place diagrams within them that depict views based upon the alternate organizational concern or application viewpoint. Applying the «perspective» stereotype to a package does several things. It visually identifies that package as representing a particular viewpoint. It also supports a model validation rule that warns you when semantic elements are placed in a «perspective» package. It also serves as a designator of packages that should be bypassed by RSA transformations

Create Self-Updating Depictions of Specific Concerns Using Topic Diagrams

In contrast to 'normal' diagrams wherein you manually place the elements you wish to depict, the contents of a Topic Diagram are determined by a query that is run against existing model contents. To create a Topic Diagram, you select a 'topical' model element, then define what other elements you wish to appear in the diagram based upon the types of relationship(s) they have to the topical element. As the semantic content of the model changes, the Topic Diagrams adjust accordingly.

Examine Models Via Browse Diagrams

Browse Diagrams are not *specifically* a tool for model organization. Their purpose is to facilitate discovery and understanding of model content without having to manually compose diagrams. But in the context of model organization it is good to be aware of them since they might reduce your need to compose persisted diagrams. That, in turn, could reduce the size and complexity of your models, leaving them easier to organize.

Browse diagrams are a bit like Topic diagrams, but with the key difference is that Browse Diagrams are never persisted, they are always generated on-the-fly. To produce a Browse Diagram, you select a model element (from a diagram or the Model Explorer), and use the context menu to “Explore in Browse Diagram”. This will produce a diagram depicting the selected element as the ‘focal point’ with related elements presented in a radial layout around the focal point. Of course, you can then select one of the related elements in that Browse Diagram, and make it the focal point of another Browse Diagram, and continue in this manner as long as you like.

Inter-Diagram Navigation

In RSA there are two mechanisms for inter-diagram navigation:

- It is possible to drag a diagram node from the Model Explorer to some other ‘host’ diagram. Then you can double-click the resultant icon on the host diagram to open the referenced diagram.

- Whenever you create a new UML package in a model, a “Main” diagram (freeform diagram) is automatically created. By default, this “Main” diagram is created as the ‘default’ diagram of the package. You can rename the diagram to something other than “Main”, and it will still be treated as the ‘default’. You can also select a different diagram in the package and make it the ‘default’ diagram for that package. The purpose of the ‘default’ diagram is this: if you place the package itself onto some other ‘host’ diagram, you can then double-click the package, which will open its default diagram.

These mechanisms support the following organizational **guidelines**, which can apply to models of any type:

1. Compose the Main diagram (or other default diagram) of each modeling file to depict:
    a. each top-level package in the modeling file
    b. the diagram icons for any other diagrams that reside in the root package of the modeling file (in other words, do not depict the icon for the default diagram itself)

2. Compose the Main diagram (or other default diagram) of each top-level package to depict:
    a. the packages that it directly contains
    b. the diagram icons for any other diagrams that it directly contains

3. Repeat this pattern for each successively lower level of packages.

5. Guidelines for Internal Organization of Use-Case Model

NOTE: In this section and in the other model type-specific sections that follow, the guidelines will be illustrated using examples taken from the Auction showcase example that is included with RSA. Study the example to see additional organizational details and the content of diagrams.

Use-Case Model High-Level Organization

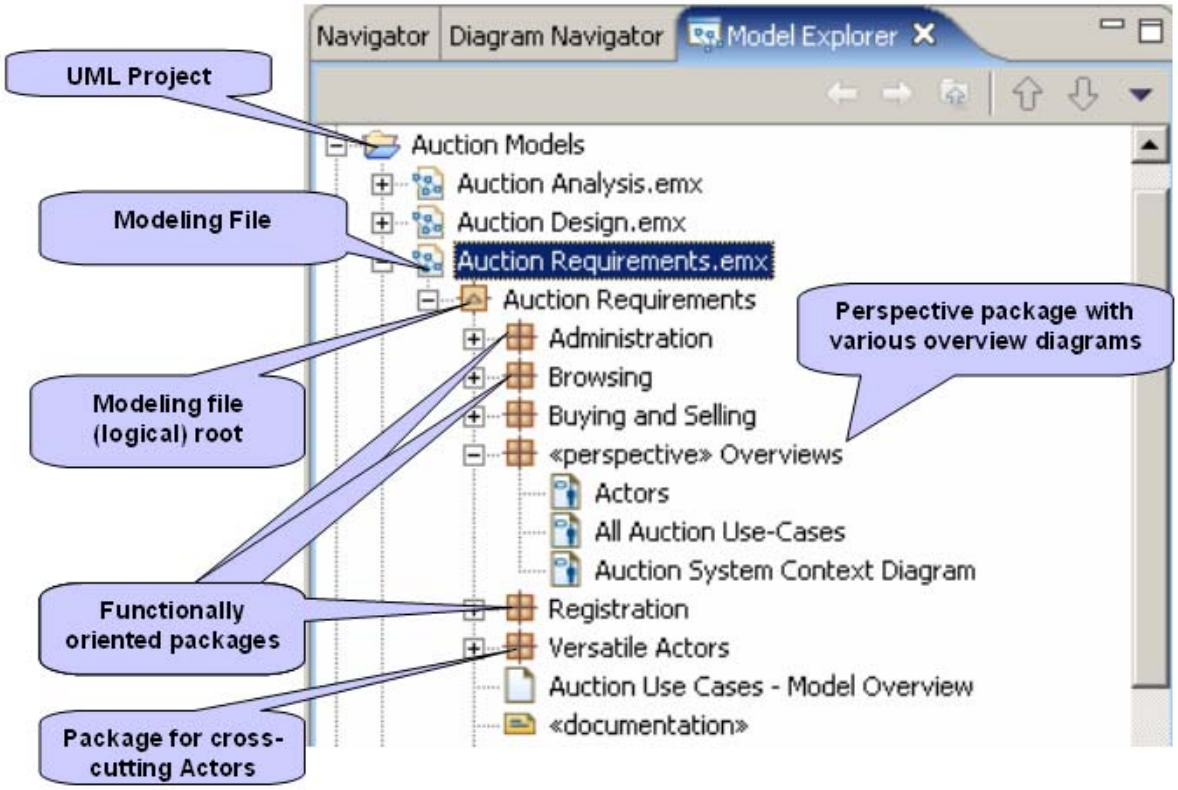


Figure 5-1

Figure 5-1 illustrates the following guidelines for structuring use-case models:

1. Use top-level packages to establish functionally-oriented groupings. **Rationale**:

    - This generally maps well to division-of-labor concerns when a team of people will be working on the use-case model. And it sets you up well in case you later decide to break the use-case model into multiple modeling files due to file contention having become an issue (you can just create a separate modeling file per top-level package).

    - Compared to other organizational approaches, this will generally map better to the organization of the eventual implementation. This is important if you will be using transformations to seed each successive lower level of abstraction. Specifically, if you will be generating seed content into an analysis model based upon the use-case model, you want the packaging structure of the use-case model to map well to the desired packaging structure of the target analysis model.

In turn, you want the packaging structure of the analysis model to map well to the design model, and the packaging structure of the design model to map well to the set of projects that will comprise the implementation. The simpler these mappings, the less work will be required to configure the transformations from one abstraction level to the next.

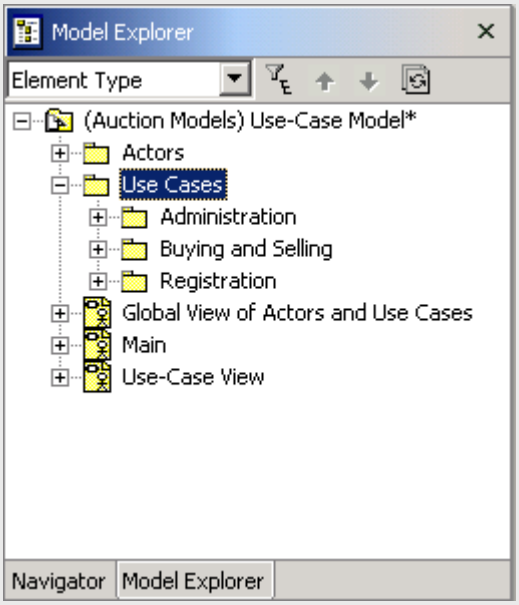
2. Use another top-level package to capture 'broadly empowered' or 'versatile' Actors.

3. Use diagrams in «perspective» packages to capture high-level, cross-cutting views of the use cases. **Rationale**:

    - Provide cross-cutting views and views of 'architecturally significant' use cases while keeping the semantic elements of the model organized into functionally-oriented groupings.

XDE/Rose

RSA guidance somewhat revises the traditional guidance for high-level organization of the use-case model, which was to create a package for actors and another for use cases. Then, as the size and complexity of the model demanded, you would use lower-level packages to establish functionally oriented groupings as shown in this XDE-based example:



Use-Case Model Content

It is outside the scope of this document to serve as a detailed tutorial on how to write good use cases or the dos and don'ts of good use case modeling. However, here is a brief discussion of what could be included in a use-case model in addition to the actors and the use cases.

- **Recommended:** Create a 'main' diagram at the model root that depicts the other packages of the model and supports drill-down into those packages and their respective 'main' diagrams.

- **Recommended:** In each use case package, include a diagram that depicts the package's use cases, any relationships among them, and the actors that participate in them. (If the number of cases is large, more than one diagram may be appropriate.)

- **Recommended:** Describe each use case's main and alternate flows in its Documentation field[4]. (See **Figure 5-2**.)

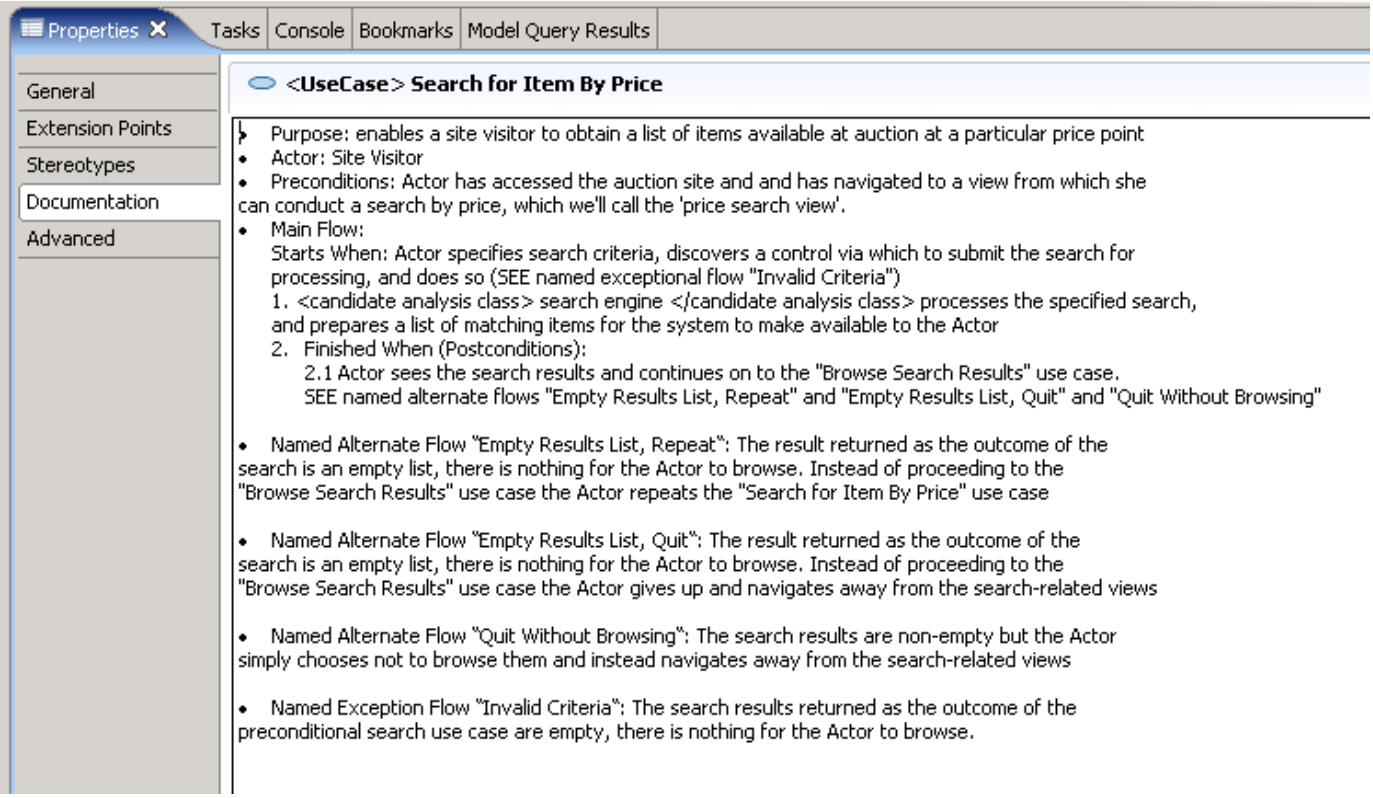


Figure 5-2

- **Optional:** When the complexity of a use case warrants it, add an Activity diagram and compose it to reflect the overall activity flows of the use case. (See **Figure 5-3**.) **Rationale:** This helps show the conditions that correspond to each of the (main and alternate/exceptional) flows and helps ensure that all the various flows ultimately re-converge. (Adding an Activity diagram in RSM/RSA will result in an Activity being automatically added to the use case, with the diagram under the Activity.)

- **Optional:** Model a 'black box' realization of each of the named (main, alternate, and exceptional) flows of the use case; add a collaboration occurrence to the use case; add to it an interaction instance corresponding to the main flow of the use case plus an interaction instance for each of the named alternate and exceptional flows; compose a sequence diagram (or alternatively a communication diagram) for each interaction instance. These use-case collaboration instances should not be confused with analysis-level use-case realizations (as described in the analysis model) or with design-level use-case realizations (as described in the design model). Those are "white box" realizations of the use case and describe interactions among the internal elements of a solution. The collaboration occurrences proposed here for the use-case model are strictly "black box" interactions

[4] The formatting depicted in the use case description example was accomplished by creating the textual 'template' for a use case description using an RTF-capable editor, then copying and pasting the template into the use case description field.

between actors and the system. (See **Figure 5-3**.) **Rationale:** This provides non-technical stakeholders with a high-level picture of how the users of the system will interact with the system. It might also help you identify the various views (screens or pages) that will be required as part of the implementation. It also formally establishes the naming of the use case's various flows (scenarios) by assigning those names to semantic model elements (that is, to the collaboration occurrences).

> ***XDE/Rose***
> In UML 1.x, you would have used "Collaboration Instance" instead of "Collaboration Occurrence" for this purpose.

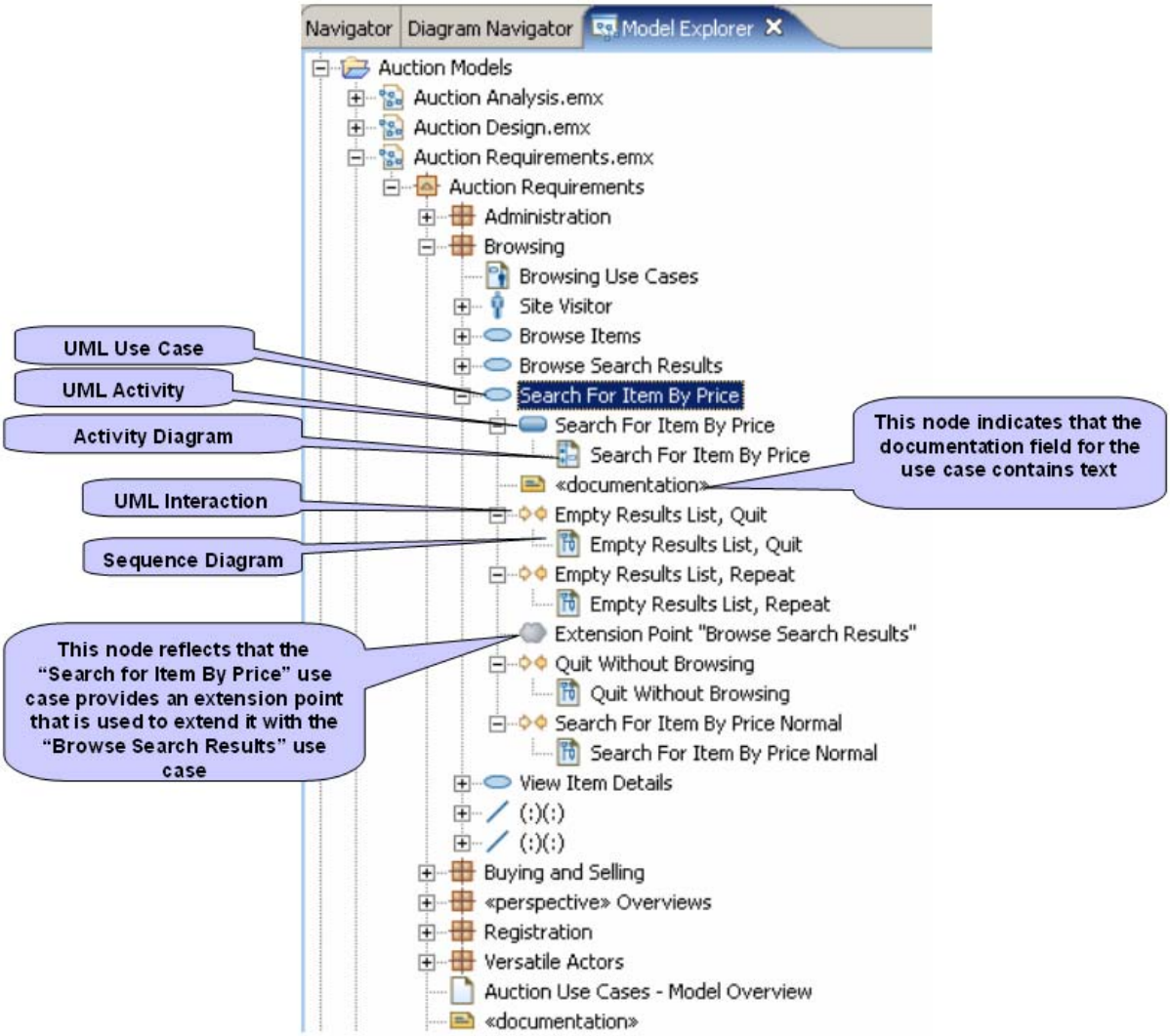


Figure 5-3

- **Optional:** If you are following the RUP guidance to identify 'architecturally significant" views of your architecture, and particularly if you will be maintaining a Software Architecture Document, add a top-level «perspective» package to contain use case diagrams that depict the architecturally significant use cases. You might want to name the package "Use-Case View of Architecture".

6. Guidelines for Internal Organization of Analysis Model

The analysis model represents a 'first-cut' at a solution. It is a stepping-stone to get from requirements to final design, focusing on capturing information about the business domain and showing candidate solution elements at a high level of abstraction that is close to the business. It is where analysis classes and analysis level use-case realizations reside. It is through the process of modeling use-case realizations (primarily using sequence diagrams) that you begin to *discover* what classes are needed for the solution—in particular, these will be classes that correspond to the lifelines you discover you need in the sequence diagrams. There are also some rules-of-thumb that can be applied to suggest analysis model content based upon the content of the use-case model. These will be touched upon later in this section.

In RUP, whether or not an analysis model should be maintained independently of the design model is a project-specific decision, one you make on the basis of whether you believe the value of maintaining the separate analysis model will warrant the time invested. If a separate analysis model is created, but not maintained, then the analysis classes will be moved into the design model and refined. Or perhaps the analysis model gradually evolves to become the design model[5]. In product-specific terms, here are some options you might consider:

1. Create an analysis model that resides in a modeling file (or set of files) based on the Analysis Model template. Then use a manual process or automated transformations to create refined versions of the analysis elements in a second model file (or set of files) based on the Enterprise IT Design Model template, and then dispose of the analysis modeling files. This leaves you the option of maintaining the separate analysis model on going, or discarding it.
2. Do analysis-level modeling in a modeling file (or set of files) based on the Enterprise IT Design Model template, to which you apply the Analysis Profile. This way you can start modeling the use-case realizations using analysis classes, and then over time refine them so that design interfaces take on the roles in the behaviors.
3. A hybrid of the second and third options is to maintain an analysis model of sorts, within the same modeling file(s) as the design model. To do this you would segregate the analysis content into packages to which you apply the keyword «analysis». This affords the opportunity to retain analysis-level artifacts within the same modeling files as their more refined design-level counterparts.

A concern to be aware of when using RSA transformations to generate implementations is that those transformations can in many cases accept analysis-level elements as their inputs, saving you some of the steps of manually refining those elements into design elements. When using RSA in this manner, options 2 or 3 above are preferred. The standard code-generating transformations packaged, as part of RSA will bypass model packages that have the «analysis» keyword.

[5] RUP in fact calls out the option of creating analysis classes and analysis-level use-case realizations in the design model and then evolving them directly into their design forms from there. . Under that approach, as the design model is "discovered" you could create packages along the way in which you preserve some of the "pure analysis" perspectives.

Analysis Model High-Level Organization

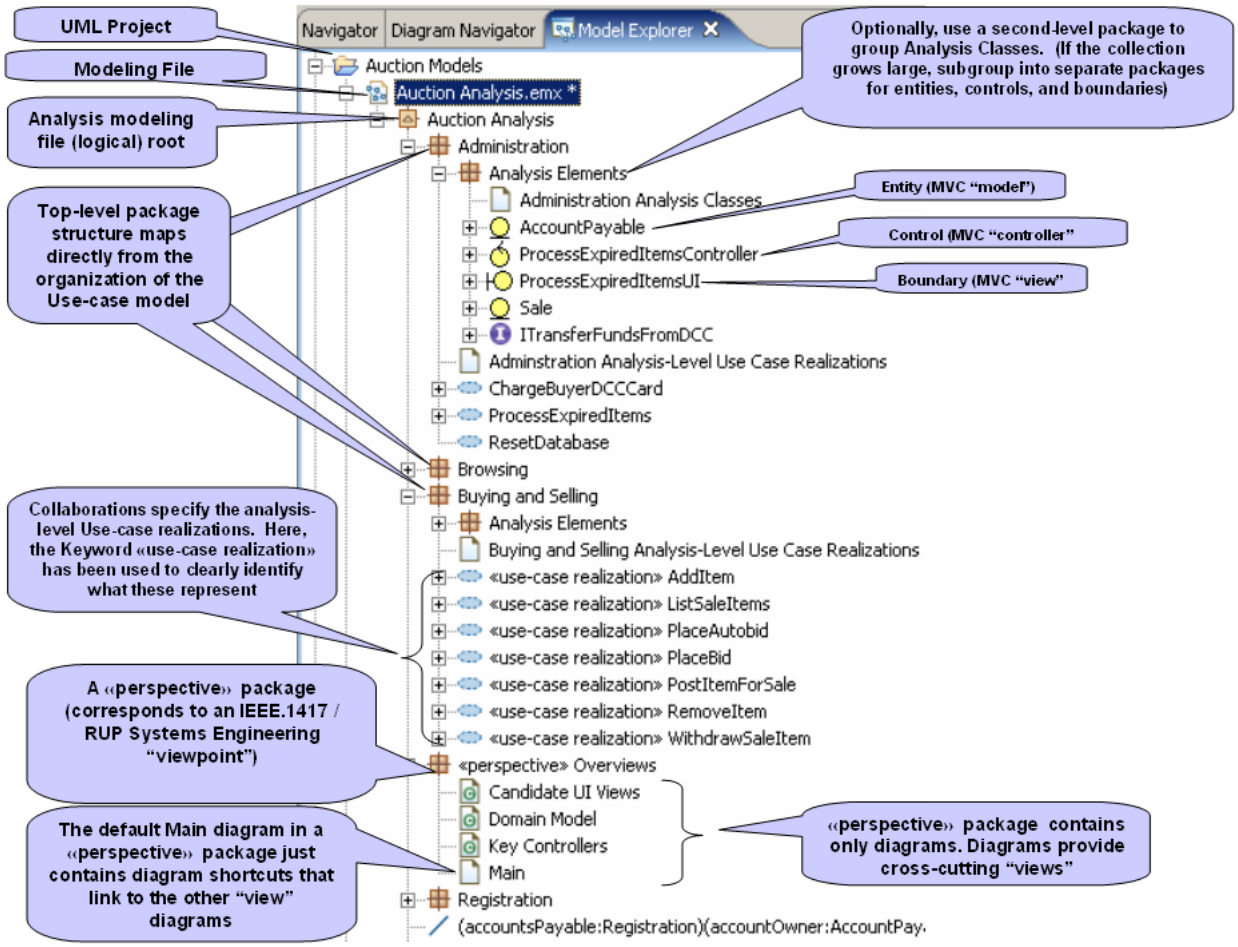


Figure 6-1

Figure 6-1 illustrates the following guidelines for structuring analysis models:

1. Use top-level packages to establish functionally oriented groupings for analysis classes. **Rationale**: same rationale as for use case model.

2. Optionally, within the top-level packages, use sub-packages to collect and organize the analysis classes.

3. Use diagrams in «perspective» packages to capture alternative, high-level, or cross-cutting views of the analysis elements. **Rationale**: Provide different perspectives for different stakeholders while keeping the semantic elements of the model organized into functionally oriented groupings.

A slight variation of this approach is depicted in **Figure 6-2** that shows the use of a top-level package to segregate the use-case realizations from the analysis classes. Within that top-level package is a set of functionally oriented sub-packages that matches the set of top-level packages. Isolating the use-case realizations in this way enables refactoring of the package structure that contains the analysis classes, without *necessarily* affecting the organization of the use-case realizations. (Particularly if the analysis model will evolve to design *in situ*, it is likely that the package organization for classes will evolve such that it no longer matches that which was originally used for the use cases.)

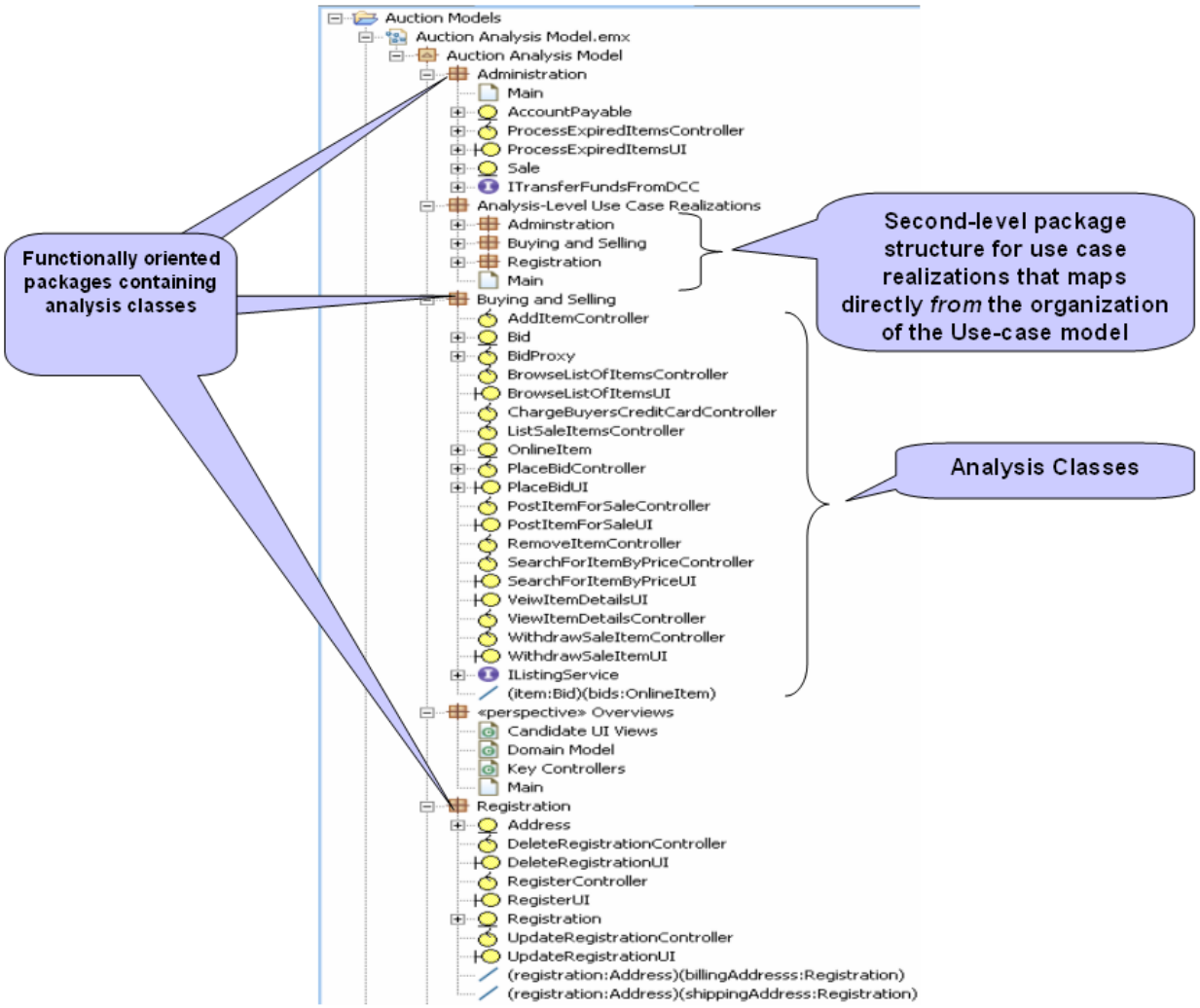


Figure 6-2

Depending upon your situation, there could be reason to introduce the use of a naming convention that anticipates merging and re-use of model content created by multiple independent groups, even including groups in different (partner) businesses. If this is a concern, consider using an inverted Internet domain namespace convention as depicted in **Figure 6-3**. Note that this is probably not a large concern for analysis modeling *per se*, but if you are taking the approach of letting your analysis model evolve into your design model *in situ,* and you anticipate re-use or business integration at the design level, you might want to plan ahead. Another potential advantage of adopting this approach: because it might map well to the organization of code generated from the analysis/design, it might simplify subsequent configuration of the code-generating transformations.

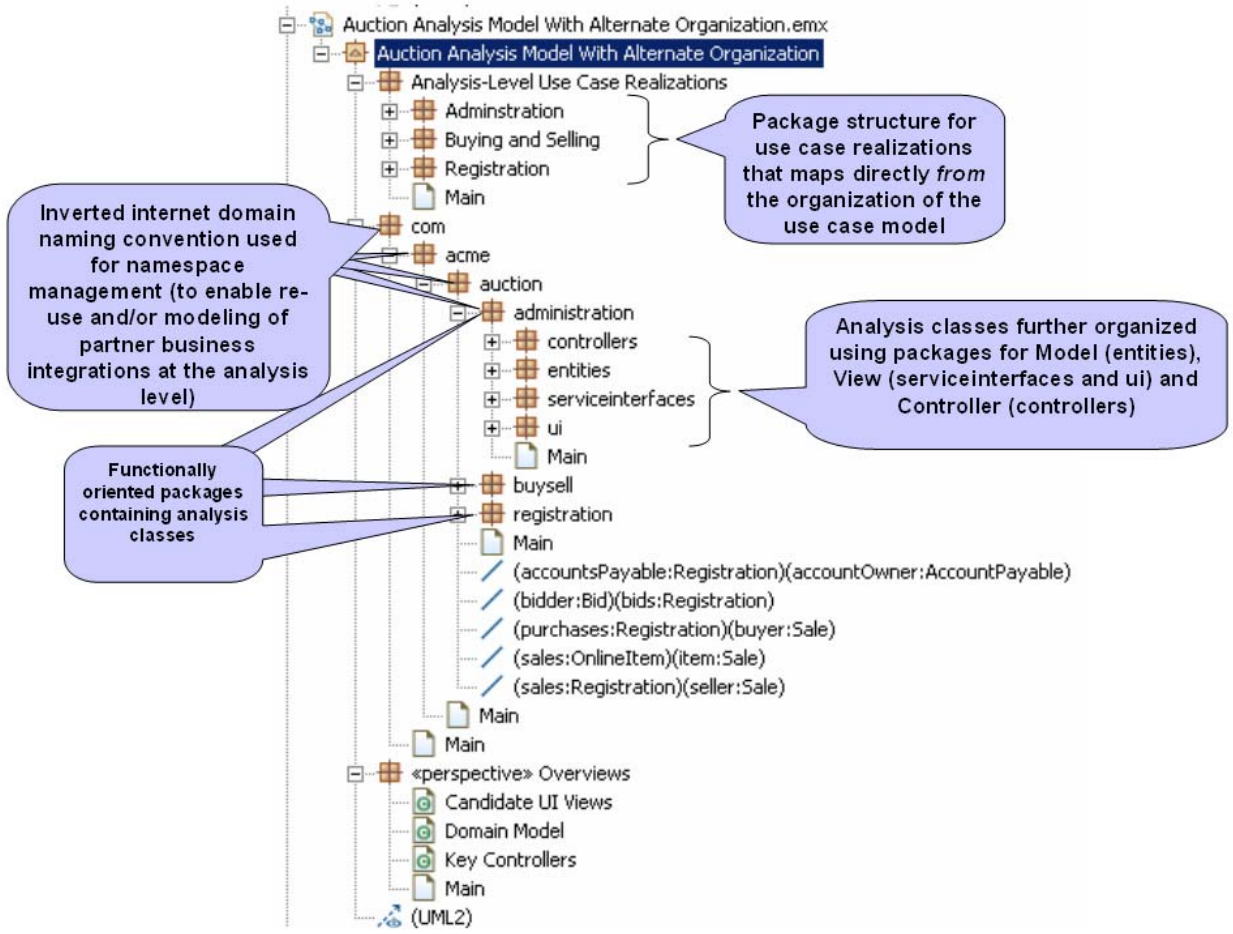


Figure 6-3

Analysis Model Content

There are a number of ways to discover what the analysis classes are. One way is to start drawing the sequence diagrams that suggest use-case realizations. As you do, you will discover what lifelines you need, and generally each lifeline will correspond to a likely analysis class. When you discover classes this way, you might create them in the use-case realization packages of the analysis model but you should not leave them there. You should 'refactor' the model to move the analysis classes into functionally oriented packages as described earlier in the guidelines for high-level organization of the analysis model (see ***Figure 6-1***).

Another useful approach for discovering analysis classes: 'seed' the analysis model with classes based upon these rules-of-thumb:

- For each use-case (in the use-case model) add a «control» class to the analysis model. «control» classes represent the business logic associated with the use case. (Later, in design, they will also map to concerns such as session management.)

- For each actor/use-case relationship (in the use case model) add a «boundary» class to the analysis model. The «boundary» classes represent interfaces between the solution and a human actor or between the solution and some external system. The «boundary» classes that correspond to a human actor are likely to eventually map to one or more user interface artifacts in the design and implementation. The «boundary» classes that correspond to an external system might eventually map to some sort of adapter layer in the design and implementation.

- Through a process such as CRC card analysis, or word analysis of use case descriptions, identify additional «control» classes (verbs) and «entity» classes (nouns).

When you use this seeding approach to identify analysis classes, you can place the classes directly into functionally oriented packages as described earlier in the guidelines for high-level organization of the analysis model (see ***Figure 6-1***).

However you go about discovery of analysis classes, you are almost sure to recognize that changes to your original functional package organization are needed.

Optional: Use second-level packages within the analysis class packages to further organize the content of those packages (see **Figure 6-4**).

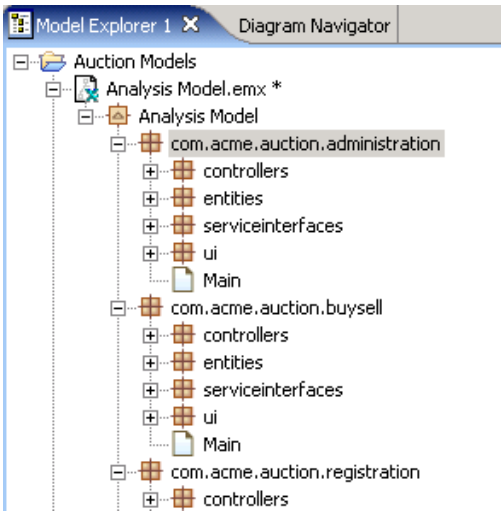


Figure 6-4

Recommended: The analysis model should contain analysis-level use-case realizations**,** which describe how the use-cases are performed in terms of the analysis classes. Each of the analysis use-case realizations (represented by a UML Collaboration) realizes a use case in the use-case model and has the same name as that use case. See **Figure 6-5**. For each named use-case flow[6] that you feel should be modeled as an analysis-level realization, add a sequence diagram (which will automatically add an owning Interaction). **Figure 6-6** shows the types of semantic content that will be added to the model as you create sequence diagrams. (Note that you can filter any UML element type from the Model Explorer view, and so hide much of the 'clutter' depicted in **Figure 6-6**.)

[6] As previously established in the Use Case Model

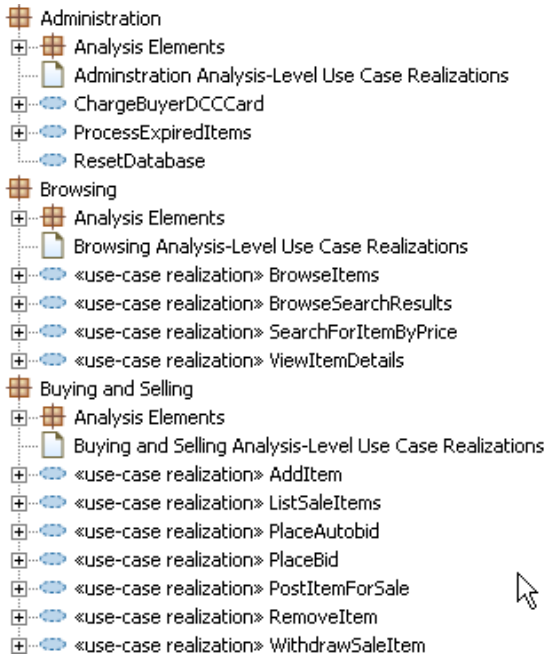


Figure 6-5

Optional: Once you have created the sequence diagram for a use case flow, you can select its owning UML Interaction in the Model Explorer and add a Communication Diagram to it. The new Communication Diagram will be automatically populated with the analysis class instances that participated in the sequence diagram.

Recommended: Create a Realization dependency relationship from each use-case realization (UML Collaboration) and the corresponding use case from the use-case model (see **Figure 6-6**). Because you can use features such as Topic Diagrams and Traceability Analysis to understand the traceability relationships in your model, you don't really need to retain permanent diagrams to depict the traceability relationships, so it is recommended that you create the relationships using some sort of 'throw-away' diagram, for example:

- Add a free form diagram to the Collaboration.
- Drag the Collaboration onto it.
- Drag the use case onto it.
- Draw the dependency relationship.
- Finally, (in the Model Explorer) delete the diagram from the Collaboration.

Recommended: Include a "Participants" diagram for each use-case realization, to show the analysis classes that participate in the realization (that is, the analysis classes whose instances appear on the interaction diagrams that describe the realization of the use case) and the relationships among those classes that support the collaboration described in the interaction diagrams. See **Figure 6-6**.

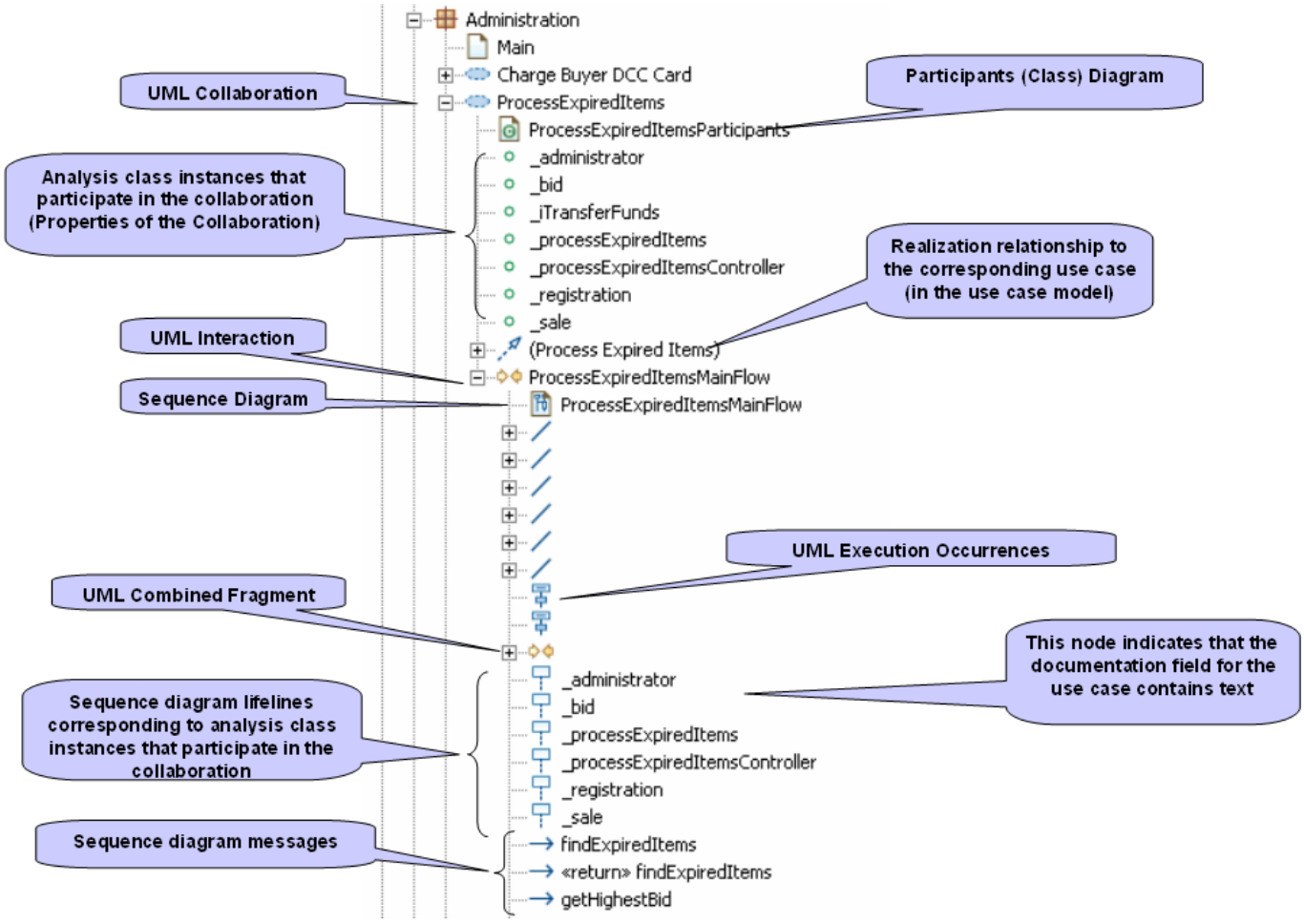
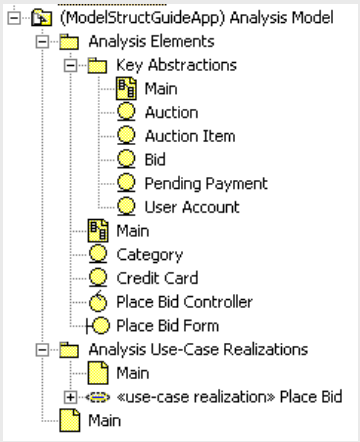


Figure 6-6

XDE/Rose

The traditionally recommended structure for the **Analysis Model** as shown below is modified for RSA to place emphasis on a functionally oriented package organization for analysis classes. Note also that the use of a Key Abstractions package (which would compromise an otherwise functionally oriented packaging approach) is replaced by use of a Key Abstractions diagram (or diagrams) in a «perspective» package.



7. Guidelines for Internal Organization of Design Model

Design Model High-Level Organization

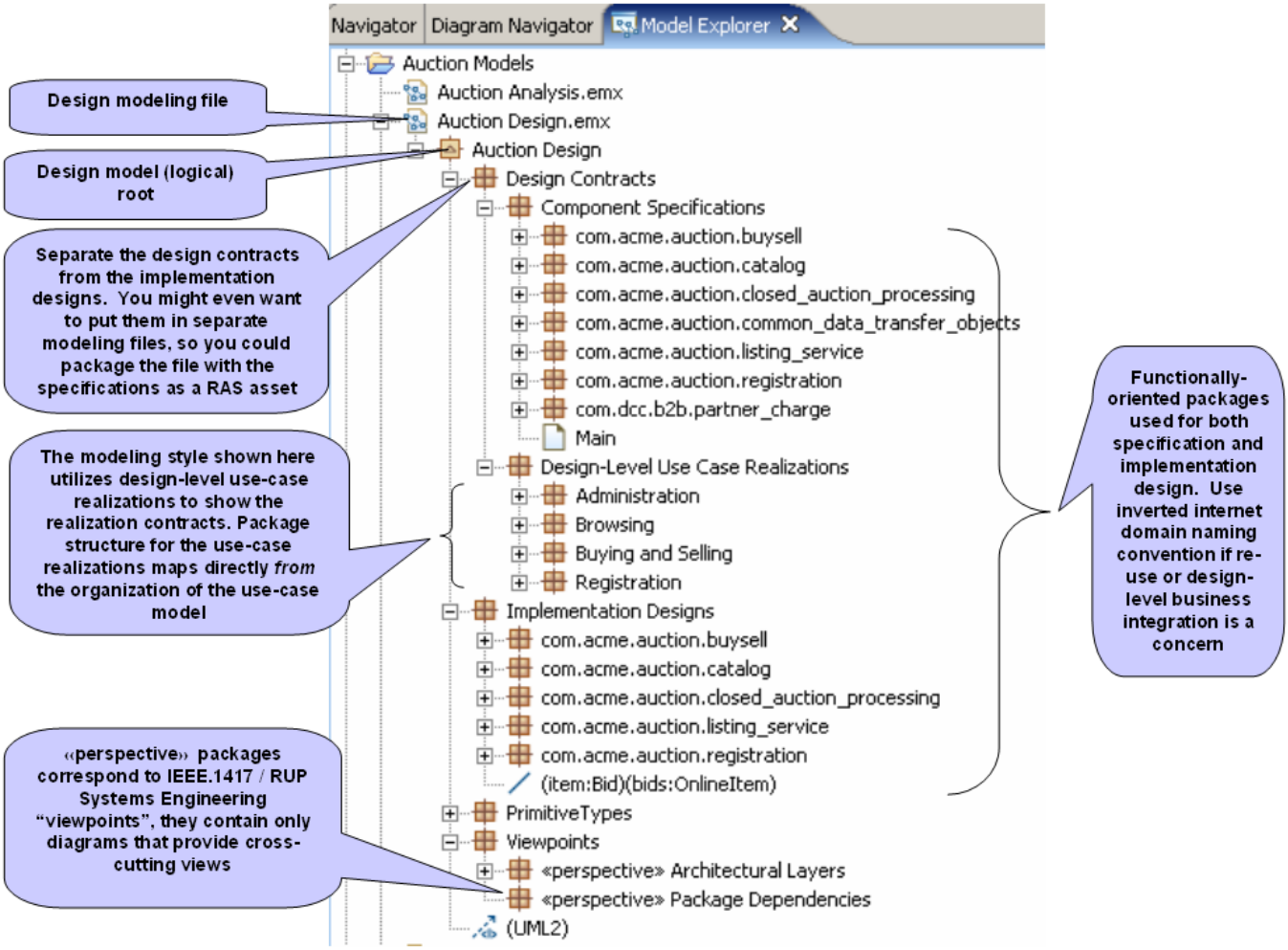


Figure 7-1

Figure 7-1 illustrates the following guidelines for structuring design models:

1. Separate specifications from implementation designs. The illustration shows the use of top-level "Design Contracts" and "Implementation Designs" packages to accomplish this.

2. Use lower-level packages to establish functionally oriented groupings. You might, for instance, start with the organization that you used during analysis, and let it evolve as you make decisions about how the analysis classes map to actual design classes, components, and services. (Any initial organizational scheme is likely to evolve during design—see further discussion below).

   A word about subsystems might be in order at this point. In versions of UML prior to 2, a subsystem is a specialized type of package. In UML 2, a subsystem is a specialized type of component, and a

Component can contain packages. In UML 2, «subsystem» Components are viable organizational/namespace alternatives to packages, yet UML2 is vague about appropriate use of subsystem versus package. Suggestion: Use Packages at levels of granularity such as the design subsystems of a particular application, and reserve Subsystems to represent entire applications (for example, CRM or SCM) in enterprise-wide views of architecture.

> ***XDE/Rose***
> At the time of this writing, it was expected that Rose and XDE model import tools would offer the option to map UML 1.x subsystems to either UML2 Subsystems or to packages with the «subsystem» keyword applied.

3. It is likely that the organization of design elements will evolve away from how the use-cases of the system are organized (in the use-case model and perhaps in the analysis model, if a separate analysis model is being maintained). Use packages to further sub-divide the design contracts into the design element specifications (the usage contracts), and the design-level use-case realizations (the realization contracts), and maintain a package sub-structure for the use-case realizations that continues to mirror the organization of the use-cases themselves.

4. *Consider* using architectural layers as the basis for the second-level organizational scheme for the elements that make up the specifications and implementation designs of the functional areas (and see further discussion below).

5. Within the UML components and packages that group semantic model elements, place diagrams that provide views specific to that grouping. This guideline pertains whether that grouping is based upon functionally oriented subsets of the business domain, upon an architectural layer, or what-have-you. Make the 'default' diagram have the same name as the package or component itself, and compose it to show an overview of the contents of the package. This keeps some diagrams close to what they depict, making it easier to navigate and understand the model.

6. You might want to introduce the use of an inverted Internet domain namespace in the design model. **Rationale**:
   - Basically the same reasons that doing so is important with respect to language-specific implementations:
     a. scenarios involving integration work where there are multiple model-driven applications involved (especially with partner companies)
     b. re-use scenarios
   - This will likely simplify subsequent configuration of transformations to implementation (source-to-destination location and name mapping).

7. *Consider* using package names that will be valid in the target implementation platform(s), to avoid the burden and potential confusion of namespace mapping. (For the most part, this simply means "do not use spaces or punctuation other than underscores in the names".)

8. Use lowercase for package names to make them easier to distinguish from class names in a package.

9. *Consider* using different names for Interfaces and the Components or Classes that realize them. Either use ILoan and Loan, or Loan and LoanImpl for interface and implementation names. This is not actually necessary in the model, but is often a good idea in generated code, so this is another area where you can spare yourself some subsequent transformation configuration work.

10. In the following scenario, any of the analysis-level content from which code is not meant to be generated should be segregated within packages stereotyped as «analysis»[7].

    - A) You have to chosen bypass the use of a separate analysis model, and to populate the design model with analysis-level content and maintain that content at the analysis level of abstraction while also creating design-level content in the same model, and

    - B) You will be driving model-to-code transformations from the EIT Design Model.

11. Use diagrams in «perspective» packages to capture high-level, cross-cutting views of the design elements. **Rationale**: Provide cross-cutting views, views of 'architecturally significant' content, and views that appeal to different types of stakeholders while keeping the semantic elements of the model organized into functionally oriented groupings.

It is important to recognize that the packaging structures of design models will evolve over time. Ultimately, the organization should correspond to how you structure your architecture into components and services. This approach to the *end game* organization of the design will then generally afford the best potential for packaging re-usable assets and the most straightforward mapping from the design to the set of projects and folders that will hold implementation artifacts (code, metadata, documentation) generated from the design.

However, the *initial* organization should correspond more or less directly to the organizational approach you used for the use-case model and then revised during analysis[8]. In fact (as described in the prior section "Guidelines for Internal Organization of the Analysis Model"), you can elect to let your analysis model evolve into design in-place. In other words, the initial organization of the design will tend to group together cohesive and loosely coupled sets of business concerns, and isolate cross-cutting or re-usable elements. This approach to initial organization proves effective because:

- If you hope to use transformations that generate design model content from analysis or use-case model content, the source package-to-destination package mappings will be simple and straightforward.

- An initial organizational approach based on functional cohesion and loose coupling of the packages clearly stands the best chance of mapping to the final component-oriented organization, meaning it should reduce the amount of refactoring you have to do as part of the design process.

- Loose coupling of packages has the potential to improve team workflows, and to facilitate re-use in cases where the design is factored into multiple modeling files.

Alternate approaches are of course possible and in some cases advisable as an *end game* organization:

- If you are targeting J2EE-based Web applications including EJBs, the organization of the design might anticipate the conventions of RSA and Rational Application Developer regarding J2EE

[7] Such packages will be bypassed by transformations.

[8] The packaging of the analysis classes is often refactored significantly as it is discovered, in order to better support reuse and unanticipated functional requirements.

projects.[9] In particular, you might choose to define top-level design packages that correspond to the architectural layers (presentation and business, with business sub-layered into session and domain). This obviously is not a platform-neutral approach and so is advisable only if you know that the solution you are designing will not be implemented on a platform other than J2EE.

- More generally, it is often the case where n-tier applications are being built that developer expertise and the division of labor correspond to presentation and business layers, so again you may choose to use top-level packages that correspond to those architectural layers. But be careful about organizing classes that are intended to support particular *business functions* in order to support a particular *architecture*. It makes either harder to change.

- If you find reason to use a non-component/service/subsystem-oriented organizational approach, you should still be able to map the organization of the design to a set of target projects and folders by investing some additional effort in configuring the code generation transformations. A special type of companion model referred to as a 'mapping model' can be used to define particularly complex mappings.

Design Model Content

There are no hard-and-fast rules for what should reside in the design model, but the following suggestions may prove useful.

[9] Loosely speaking: An Enterprise Project per system or application or large subsystem, and for each Enterprise Project a Web project for the presentation tier, and multiple EJB projects where the EJB projects correspond generally to components or minor subsystems, and where typically separate EJB projects are used for the session (session EJBs) and domain (entity EJBs) layer per component or subsystem. See section 9 of this paper for more information.

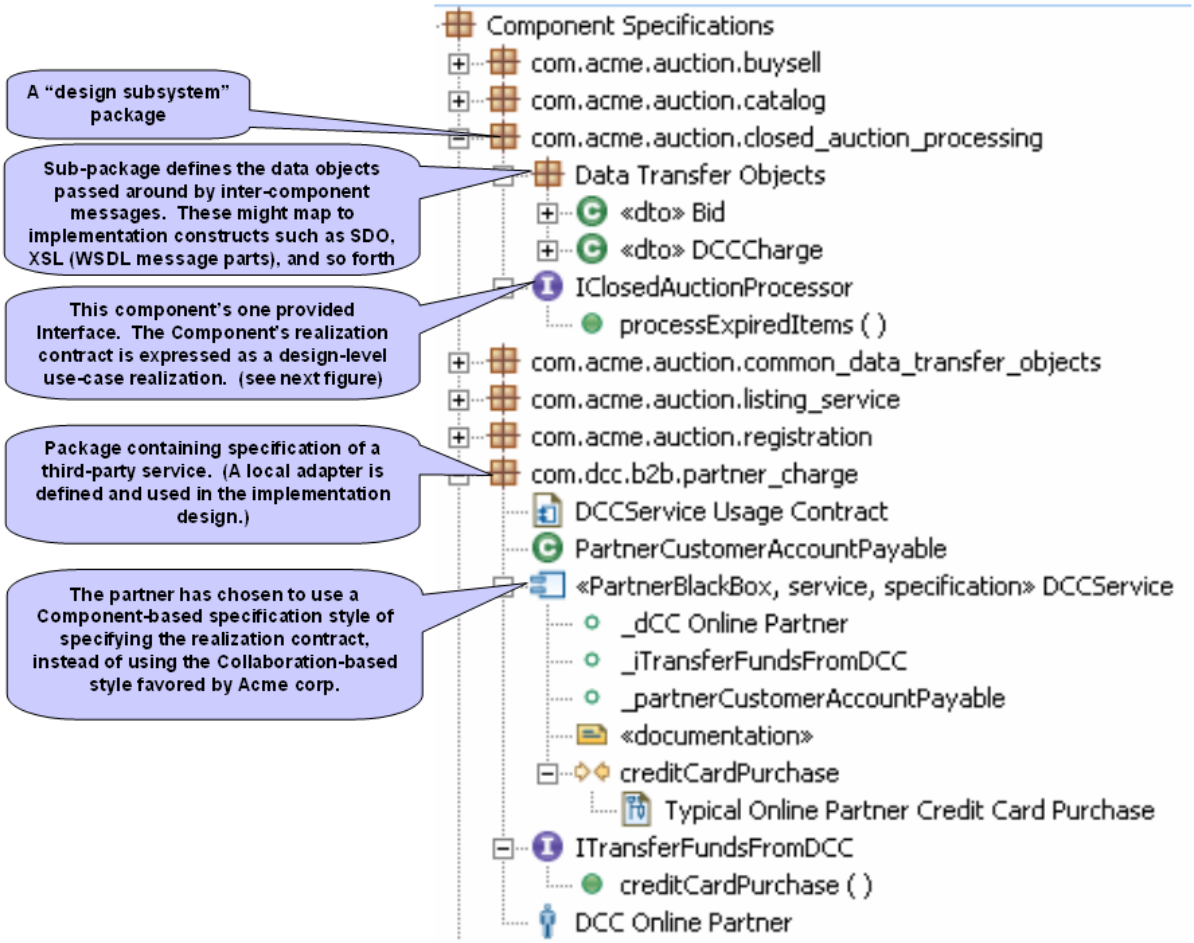

A "design subsystem" package
Sub-package defines the data objects passed around by inter-component messages. These might map to implementation constructs such as SDO, XSL (WSDL message parts), and so forth
This component's one provided Interface. The Component's realization contract is expressed as a design-level use-case realization. (see next figure)
Package containing specification of a third-party service. (A local adapter is defined and used in the implementation design.)
The partner has chosen to use a Component-based specification style of specifying the realization contract, instead of using the Collaboration-based style favored by Acme corp.
Component Specifications
com.acme.auction.buysell
com.acme.auction.catalog
com.acme.auction.closed_auction_processing
Data Transfer Objects
«dto» Bid
«dto» DCCCharge
IClosedAuctionProcessor
processExpiredItems ( )
com.acme.auction.common_data_transfer_objects
com.acme.auction.listing_service
com.acme.auction.registration
com.dcc.b2b.partner_charge
DCCService Usage Contract
PartnerCustomerAccountPayable
«PartnerBlackBox, service, specification» DCCService
_dCC Online Partner
_iTransferFundsFromDCC
_partnerCustomerAccountPayable
«documentation»
creditCardPurchase
Typical Online Partner Credit Card Purchase
ITransferFundsFromDCC
creditCardPurchase ( )
DCC Online Partner

Figure 7-2

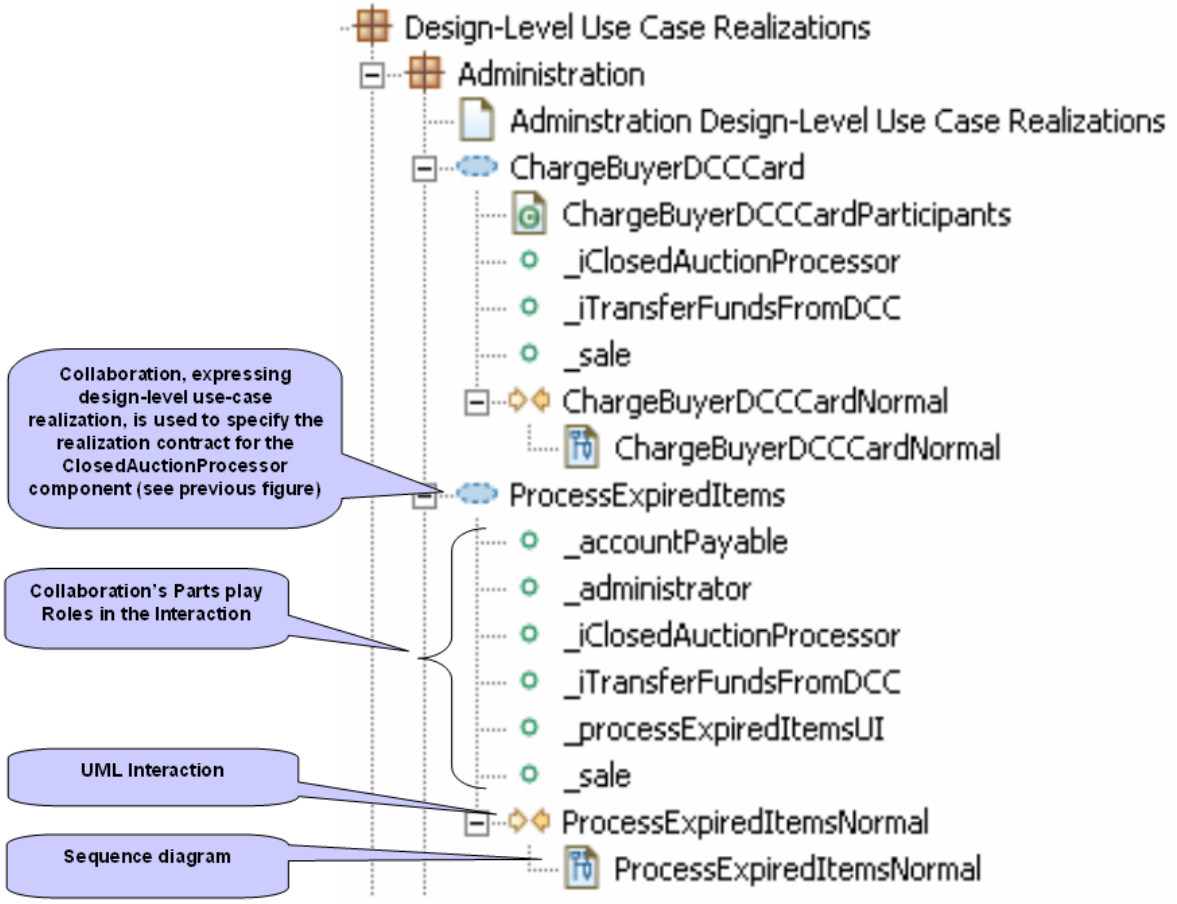


Figure 7-3

Figure 7-2 and **Figure 7-3** adhere to the organizational structure depicted in **Figure 7-1** and depict how design contracts might be specified.

- The usage contract for a "ClosedAuctionProcessor" component is expressed as a single Interface[10] (**Figure 7-2**). The corresponding realization contract is specified by a single design level use-case realization expressed as a Collaboration[11] (**Figure 7-3**) Note that whereas analysis-level use-case realizations show collaborations among analysis classes, the design-level realizations show collaborations among less abstract design elements[12]. If there is desire that the specification subset of a design model be packaged independently of the implementation design subset, then it is important that the design-level use-case realizations use only analysis or design specification elements—never implementation design elements—in their roles.

[10] Components can of course have multiple provided interfaces' it just so happens that this example only has one.

[11] Other components might participate in multiple system use-cases, so their realization contracts might reside in multiple use-case realizations. In such cases you could also include, in the same package with the component's Interface, a diagram called "{component name} Where-Used" on which you place links to the various diagrams that make up the use-case realizations for those use-cases.

[12] Another likely difference: Some of the "participant" diagrams in the design-level realizations might be Component Diagrams that depict component wiring, instead of (or in addition to) participant Class Diagrams as suggested for analysis-level use-case realizations.

- The usage and realization contracts for the third-party “DCCService” are together in one package[13]. Once again, we see that the usage contract consists of a single Interface, but in this case the realization contract is expressed using a «specification» Component (**Figure 7-2**). (Otherwise, the specification of the realization contract is just about the same, expressed using behaviors—in this case, an Interaction called “creditCardPurchase”). Another example using Components instead of Collaborations is shown in **Figure 7-4**.

- Operations are defined in interfaces, which can be realized by «specification» components (if used) or by classifiers in the Implementation Design that implement the interfaces.

- The specification of data transfer objects (which would serve as the types of parameters of the provided operations, and might map to implementation constructs such as XML schema or SDOs) can also be included as part of the usage contract. For components that are not designed to be distributable, you might or might not choose to specify data transfer objects as the specifications of the types used as operation parameters. For distributable services (for example, Web Services), it is mandatory that the service’s operations not reference objects in a local address space, so DTOs must be used.

> ***XDE/Rose***
> In prior versions of UML, guidance for use-case realizations was to use a Collaboration Instance per use-case and an interaction and sequence diagram for each of the significant flows of the realization.
>
> In Rational Software Architect, you should often be able to use just one interaction and diagram because UML2 sequence diagrams now support notations for alternate execution paths.
>
> Also, in UML 2 there is no longer a ‘Collaboration Instance’. Instead, there is ‘Collaboration Use’, which requires a Collaboration as its type. Therefore, in Rational Software Architect, use Collaborations to represent use-case realizations.

- [13] Note that the hypothetical situation here is that the DCC company supplied the Acme company with the UML specification, which Acme then incorporated into its design model. That is the type of scenario where the use of inverted Internet domain spaces could come in handy.

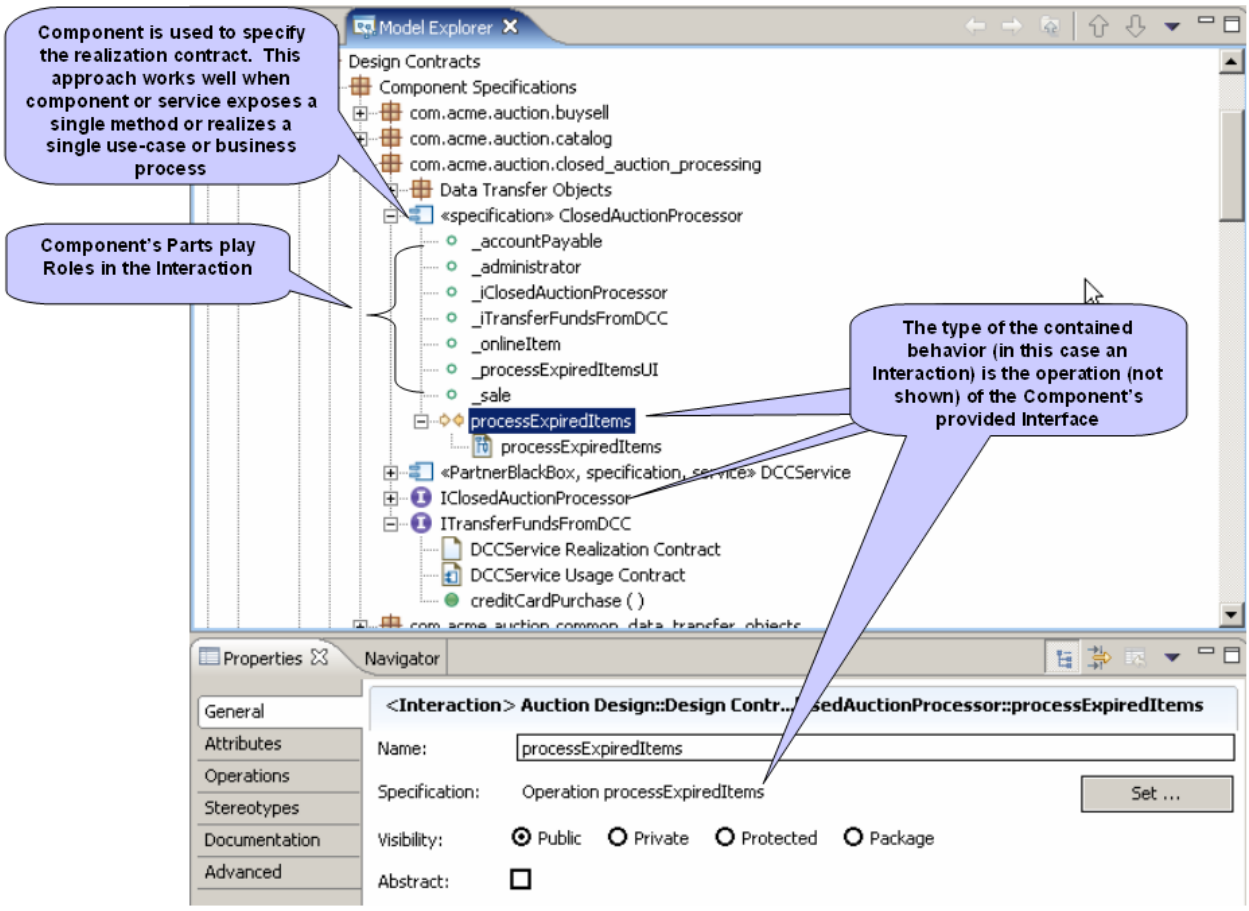


Figure 7-4

- A possible approach to specifying implementation designs is shown in **Figure 7-5**.The implementation structure is defined using simple classes that contain operations. This approach is quite typical of design models created using UML 1.x. A second possible approach that might be more in keeping with the goals of UML2 is shown in **Figure 7-6**. Here, instead of Classes we see that Components are used and that the Components do not own Operations but instead own behaviors (in this case an Interaction).

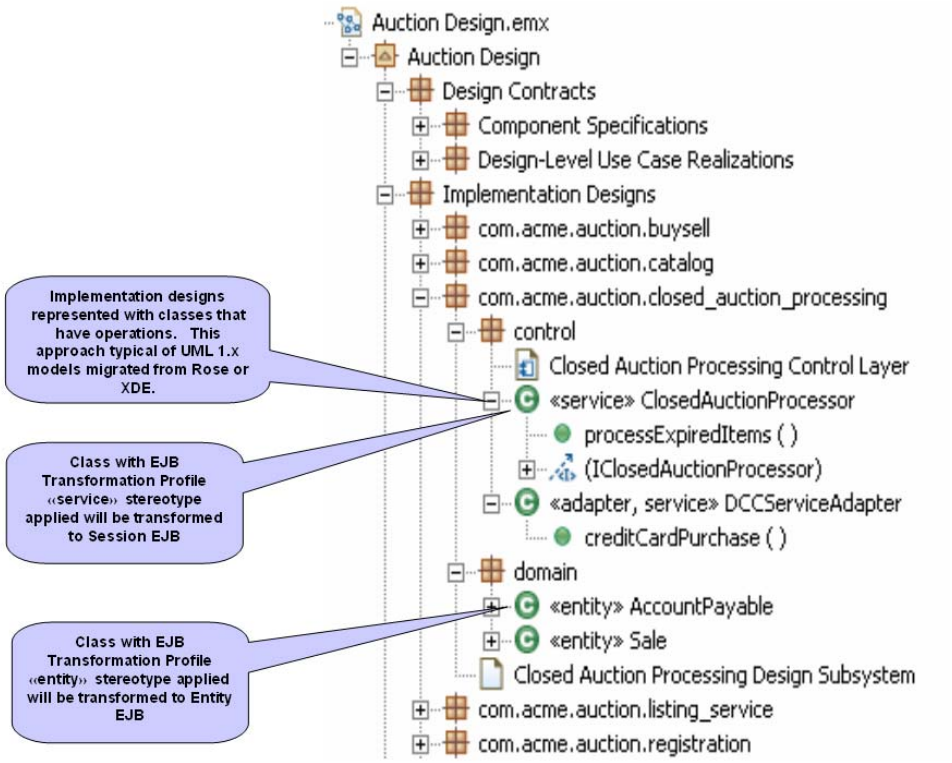


Figure 7-5

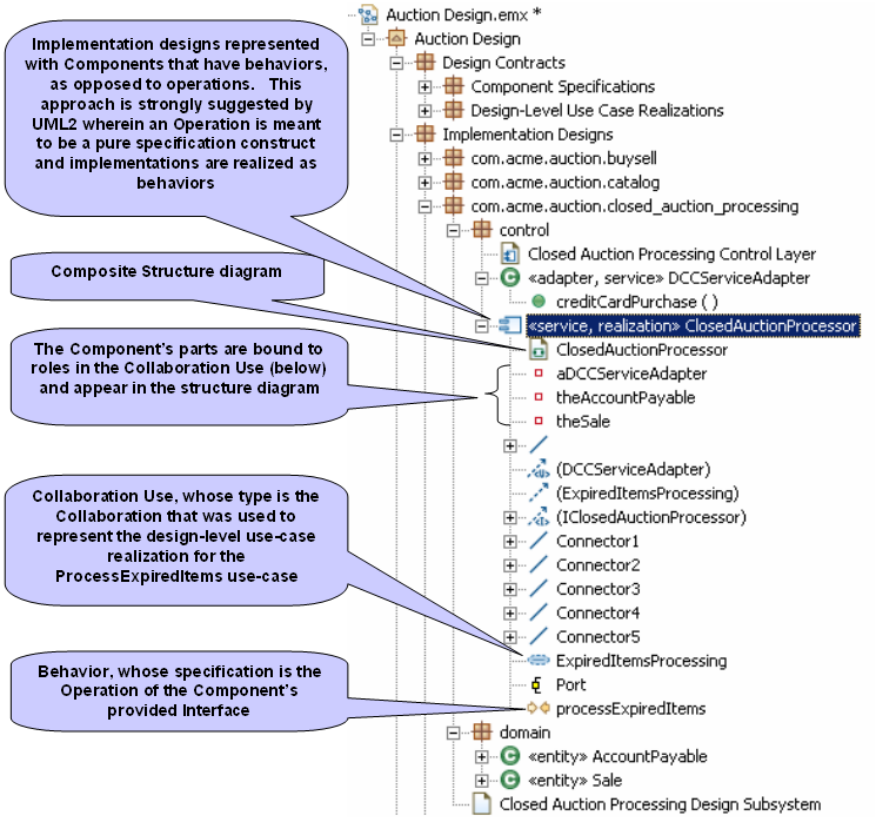


Figure 7-6

8. Guidelines for Internal Organization of Implementation Overview Model

> ***XDE/Rose***
> In the XDE Model Structure Guidelines, an Implementation Overview model was recommended as a device to provide a subsystems-level overview of the implementation. The details of each subsystem were then specified in the code model of the project that implemented the subsystem.

Strictly speaking, it should not be necessary to use an Implementation Overview model in RSA. If the design model organizational guidelines are followed, then the (end game) organization of the design model should naturally take shape around components (including the heftier «subsystem» and more distributable «service» varieties). Then, through transformations, the packages of the design can be mapped to projects. For instance, in the case of a J2EE implementation, they will map to the various Java, EJB, Web, J2EE Application, and other projects in which the implementation is developed. (And those projects in fact represent the implementation model for the solution, as noted in the Basic Concepts and Terminology section of this paper.)

However, you might still prefer to sketch out your project structure at an early stage, or you might simply prefer to see a depiction of project structures that is more visually explicit—for instance, one where the artifacts representing projects and folders are explicitly keyworded as «project» and «folder», or even «EJB Project» and «Web Project». Another consideration is that depicting finer-grained implementation artifacts (JARs, for instance) would be inappropriate for the design model (which in the Rational Software Architect theory of operation is intended to be platform-neutral). But such artifacts are perfectly acceptable for inclusion in an Implementation Overview model. Thus, there are some reasons you might want to use an Implementation Overview model. **Figure 8-1** depicts a sample Implementation Overview model

A final thought is that an Implementation Overview model might be a good place to capture informal diagrams of various aspects of the solution. **Figure 8-2** shows an informal high-concept diagram of the auction system on which most of the samples in this paper are based.

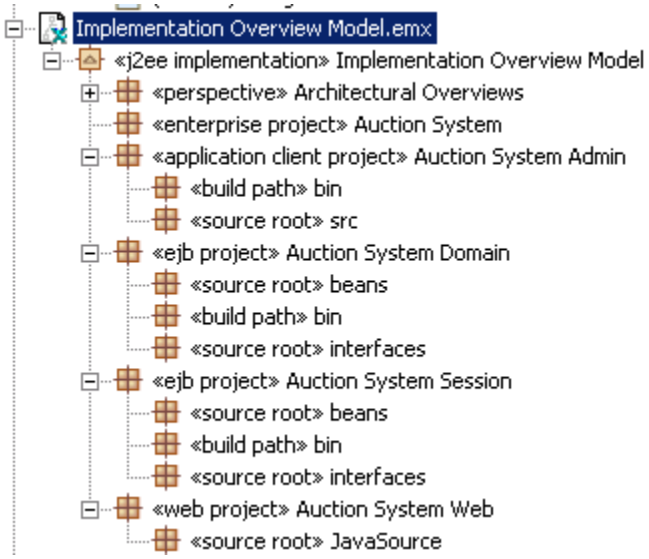


Figure 8-1

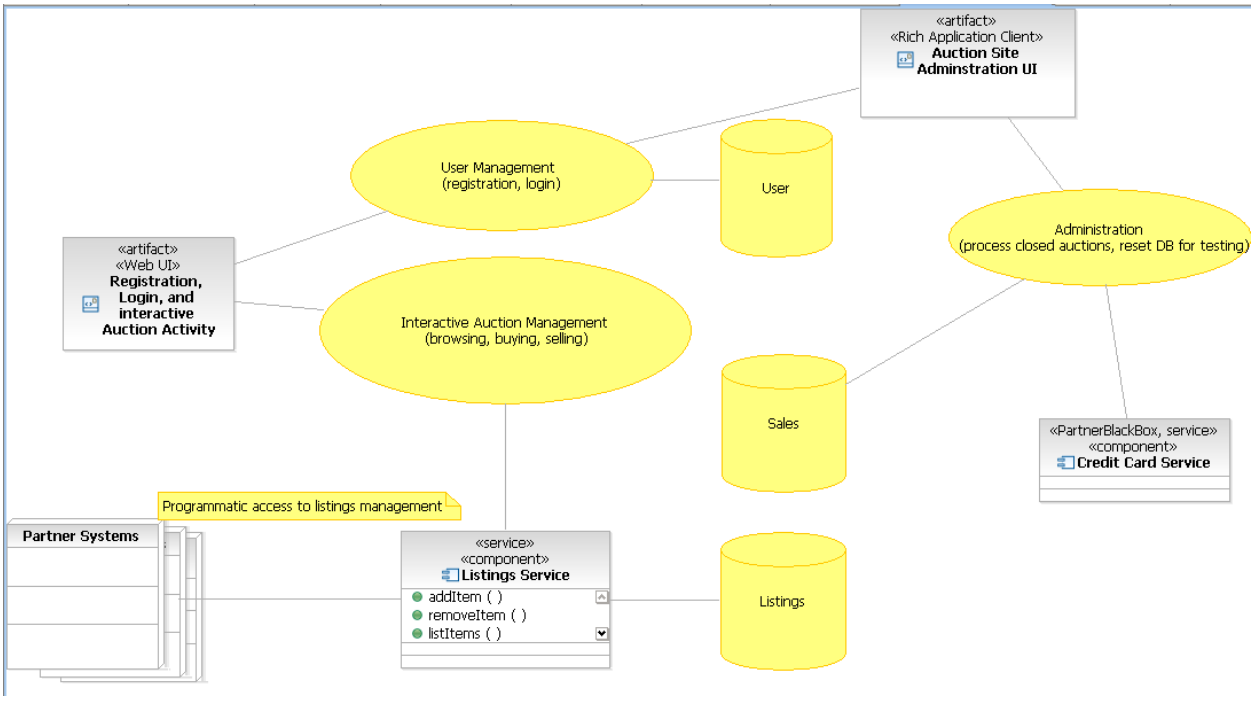


Figure 8-2

9. Guidelines for Internal Organization of Deployment Model

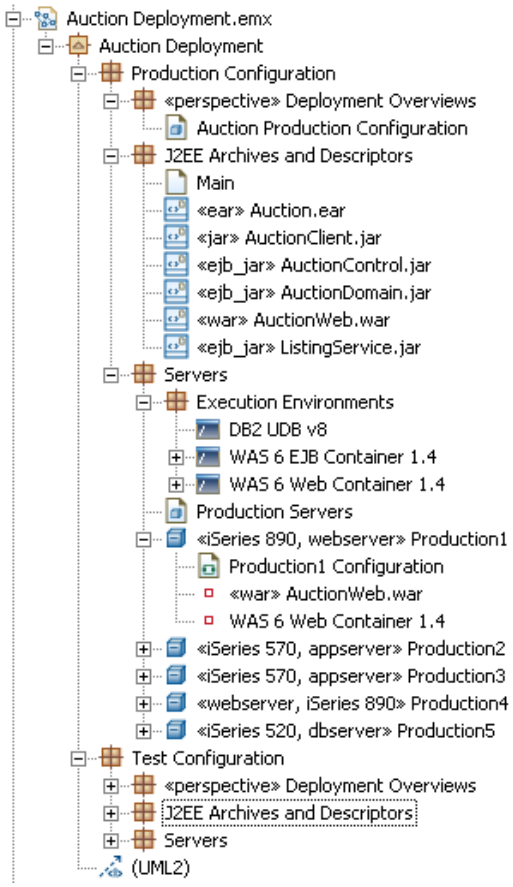


Figure 9-1

Less probably needs to be said about the deployment model than about any of the other models addressed in this paper. There should typically be very few downstream implications of your deployment modeling organization and content choices, so just do what makes sense. Still – just to get you thinking – a possible strategy and a bit of representative content are depicted above in **Figure 9-1**. In this example, note:

1. Specifications of production configurations have been separated from those of test configurations.

2. Overviews (for example, of clusters, data centers, or enterprises) are maintained in «perspective» packages.

3. A lightweight approach has been taken with regard to specializing and classifying nodes and artifacts: a combination of packaging and the use of keywords. A more sophisticated approach would be to develop a specialized UML profile that defines specialized stereotypes and properties appropriate for describing and documenting the types of resources used in your own environment.

10. Using a Modeling File to Represent the Software Architecture Document

Given its tools for organizing models, such as diagram links and support for multiple model files with cross-model references, it becomes an almost trivial matter to create a model that in effect represents the RUP Software Architecture Document and the “4+1 Views of Architecture”.

In simplest form, you might do something along the lines of **Figure 10-1**. Create a modeling file and populate it with a simple set of packages corresponding to the 4+1 Views. (The example is shown without a package for Process View, since the system in this example does not exhibit much in the way of concurrency.)

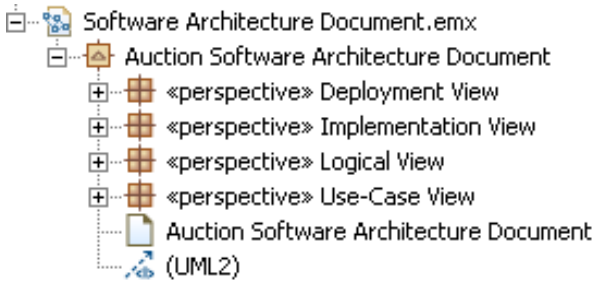


Figure 10-1

Then perhaps compose the default diagram along the lines suggested in **Figure 10-2**. You could also add additional notes or text to this diagram.

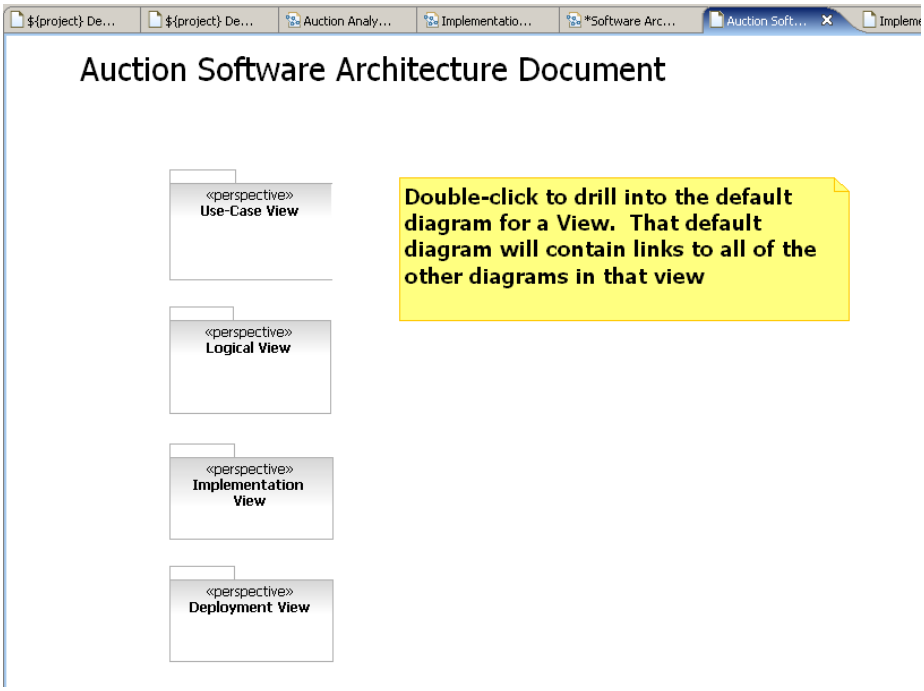


Figure 10-2

Then just create diagrams in the Software Architecture Document modeling file, using these approaches:

- Create diagrams that are composed using UML semantic elements from other modeling files, and that depict new views that were not found in those other modeling files but are needed as part of the architecture document.

- Create diagrams that are composed of geometric shapes and/or "ad-hoc" UML elements that reside in the Software Architecture Document modeling file. (Such UML elements should be just for purposes of documentation or clarification and should have no semantic significance to the actual implementation of the solution being described.)

- Create diagrams that simply contain links to the existing diagrams in other modeling files. (This technique will work well if the architecture document modeling file is to be distributed along with the other modeling files for consumption by readers. If the architecture document is instead going to be Web-published, follow one of the other approaches instead.)

11. Team Development Considerations

The "Basic Concepts and Terminology" section discussed the various models such as Use Case, Analysis, and Design, as recognized by RUP. An example was also given to illustrate the point that a "pre-implementation" model in RSM or RSA can be persisted as one or more modeling files, and also that you might maintain multiple use-case models, multiple analysis models, and so forth, where each of those models is persisted either as a single modeling file or as multiple modeling files. This section introduces some of the considerations for when and why you might choose to persist a model as multiple modeling files. A more comprehensive treatment of these issues is to be found in RSA online help.

Modeling in Teams

When multiple individuals must work in parallel on a model, you might find it more efficient to break the model into multiple modeling files (.emx files). Doing so makes it more likely that the team can do its work by checking out the modeling files for exclusive access. That in turn lowers the chances that merges will have to be performed as the result of two or more individuals modifying the same modeling file.

On the other hand, there is a trade-off. Merges, while they should be required less frequently when using multiple modeling files, will probably still have to be done on occasion. And merges process modeling *files* individually, not in groups. In other words, when a model is stored as several modeling files, the individual files are processed "out of context" of the full (logical) model. Merges work better when the merge session has access to more of the informational content of the full model—in other words, when the model is split among *fewer* modeling files.

Ultimately, partitioning models into multiple files is not as important as structuring models in such a manner as to enable multiple team members to work on them without introducing conflicting changes that must be resolved during merges. We recommend that you partition (*logical instances* of) models into multiple physical files only when it will significantly reduce the need to perform merges. Typically, you can reduce the need to perform merges only when your team members can successfully work with the individual files both *exclusively* (that is, only one team member has a file checked out at any point in time) and *in isolation* (that is, each team member can make their changes to the individual files, without also requiring access to the other files that make up the complete logical context of the model).

Two Approaches to Model Partitioning

In RSM and RSA you can make decisions about breaking logical model instances into multiple physical

files on an ad-hoc basis, by using the tools' capabilities for refactoring models. However, our recommendation is to *plan ahead.* The following paragraphs compare and contrast the two approaches.

Planned Approach: Decompose Models At the Outset

Do your best to anticipate your team's sharing needs and accordingly factor your models into logical subsets. For instance:

- Factor each use-case model into separate modeling files by looking at how functional expertise is distributed among the analysts on the team, and break things down accordingly (in essence creating a separate file wherever you might otherwise have used a package to organize the actual use cases within the model). For instance, in a healthcare provider application you might have a modeling file to contain the patient registration use cases, and another for order processing use cases. Or you might further decompose the order processing cases into separate modeling files for lab orders, radiology orders, pharmacy orders, and so forth.

- Factor each analysis model into multiple files, where again the recommended approach is to break things down according to the allocation of expertise among the team members or the logical groupings naturally found in your problem domain (two considerations which most often align).

- Factor each design model into multiple files based upon the major subsystems, services, or components of your architecture (in other words, according to how you expect the implementation work will be assigned to teams or individuals).

In each of these cases, for each model you could also maintain an additional modeling file that contains shared/common elements and diagrams that provide overviews that cross partition (subsystem/package/service) boundaries.

Additional guidance on model partitioning can be found in RSM/RSA Online Help

Ad-hoc Approach: Model Refactoring

In RSM and RSA, you can create multiple files for a model by starting out with the entire model in one file, and then later 'severing' branches of the model to create additional files. Even if you planned a partitioning strategy in advance, this approach can be useful when you recognize a new opportunity to improve team workflows.

For purposes of this discussion we are concerned with the logical content of models, so a severed branch of a model is called a "sub-model". When a sub-model is severed from its containing model, the original model maintains a 'shortcut' that points to the sub-model. The sub-model does not maintain a back-pointer to the original model. By default, the elements in the new sub-model will no longer reside in the namespace of the original parent model. However, the option is given during the severing procedure to propagate the namespace of the original model into the new sub-model. For the steps to create a sub-model by severing it from the original model, see RSM/RSA Online Help.

Cross-File References

Any two RSA modeling files can reference one another's elements. This can include references that span projects. Some of these references will represent relationships that you explicitly create, such as dependencies between use cases or associations between classes. Others are created by RSA and are in some cases hidden. For example, traceability links can be generated through the application of transformations, and those links are created as normal, visible model elements. As another example, an RSA diagram contains references that point to the semantic elements depicted in that diagram (any

semantic element can appear in multiple diagrams). The display element references are “hidden” references (that is, you will not see them in the Model Explorer).

Each cross-file reference represents a potential point of breakage in cases such as:

- One or more of the cross-referenced files cannot be properly saved (for example, due to a system crash).

- Files are moved around in the file system, without the RSM/RSA model server knowing about it.

Therefore, when planning to decompose a model into multiple files, remember that it is worth considering whether one result might be a proliferation of cross-file references. Again, organizing models to favor functional cohesion and loose coupling of organizational units (packages or modeling files) can contribute to a better team development experience.