Model Structure Guidelines For Rational Software Modeler, Rational Systems Developer, and Rational Software Architect (“Traditional RUP” Orientation)

White Paper

Bill Smith, Model Driven Development, IBM Rational Software

V7.0
March 24, 2006

Table of Contents

1. Introduction

Intended Audience

This paper is designed to support users of the Rational Software Architect (RSA), Rational Systems Developer (RSD) and Rational Software Modeler (RSM) products (collectively "**RS*x***"), especially those who are interested in applying the modeling guidance found in the Rational Unified Process (RUP®) to their use of RS*x*. If you are a user of Rational Software Modeler (RSM) you will find the paper useful but should be aware that some sections reflect capabilities available only in RSA and RSD.

The paper is focused on creating a new set of models in RS*x*. If you are new to RS*x* but have been a user of Rational Rose or Rational XDE and plan to import models from those products, you may also find the paper to be a source of guidance for restructuring your imported models.

The paper assumes that you have some knowledge of RUP and UML. It also assumes that you are familiar with the fundamental concepts and "theory of operations" of RS*x*, which you can read about in "**The New IBM Rational Design and Construction Products for Rose and XDE Users**"

Purpose

The RUP describes a set of models such as use-case models, analysis models, and design models that represent well-defined perspectives on the problem and solution domains of systems. The utility of this set of models has been proven in many real-world projects. Even if you do not follow RUP these model structures are worth considering. This paper describes how to realize them using RS*x*.

Other modeling approaches may also be considered, but model structuring guidance to support them is beyond the scope of this paper. For instance a "Business Driven Development" approach to modeling for Service Oriented Architectures could be used. It might start with a business process model – perhaps expressed as a UML 2 Activity model – and then proceed directly to specifying the contracts of a set of services that would automate tasks in the business process. That approach would suggest model structures quite different from those described in this paper.

It is important to understand that the project and model structures described in this paper are guidelines, not imperatives. Whether you decide to model a particular RUP artifact in RS*x* is a consideration of your own development process. It is often a project-specific decision. Also please recognize that RUP is not a rigid set of process rules. It is a process *framework* within which to formulate process definitions. Such process definitions can range from very formal to very lightweight.

The way you use UML modeling can also range from very formal to very informal. You might choose to treat your models like formal architectural drawings that are to be strictly followed during construction. Or you might treat your models like sketches that suggest the broad outlines of a design, but are considered disposable once the project moves into implementation. RS*x* can support you at either end of these process and modeling spectrums. The guidelines in this paper are not meant to shackle your thinking. They are meant to help you understand how to use the features of RS*x* to facilitate a process that seems best to you.

Note that RS*x* makes it possible to use models not just as blueprints, but as specifications from which significant portions of an implementation can be automatically generated. This is accomplished using model-to-model and model-to-code transformations. The use of transformations to practice Model-Driven Development (MDD) introduces some special concerns regarding model structures. If you will be using models and transformations to practice Model-Driven Development, you should also look for RS*x* MDD-specific resources on the Rational design and construction area of developerWorks®.

Scope

This paper describes how to represent the RUP model artifacts in RS*x*. It also 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 RS*x* models see

- the product documentation (tutorials, samples, online help)
- the tool mentors in the RUP configurations of which this whitepaper is a part
- RS*x* -related resources on developerWorks

Typographic Conventions

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

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

Organization of the Paper

A Basic Concepts and Terminology section will establish a working vocabulary and provide some general information about how models are implemented in the RS*x* products.

Next the RUP Model to RS*x* Model Mapping section will discuss how RS*x* supports the model types defined by RUP.

Following that are several sections that provide guidance for structuring models of various types. Some of these sections discuss different ways you can use models according to how much rigor you prefer in terms of your process, modeling approach, and architectural control.

Finally there is a discussion of the issues associated with using models in teams. It touches on strategies for managing scale and enabling sharing of models among team members in order to minimize file contention and merging.

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

RUP is tool-agnostic. As far as RUP is concerned, a model could be a drawing on a napkin or a whiteboard, something in a modeling tool, or even a mental image. From the RUP perspective a model is a logical concept.

In the context of RS*x* 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 RS*x* 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 RS*x* it is important to recognize that their models have physical manifestations. That is the subject of the next section.

Modeling Files

RS*x* 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 RS*x* supports two kinds of modeling files:

- **Pre-Implementation modeling files** are stored as individual files in the host OS file system. They have filename extensions of ".emx". These files contain
  - UML semantic elements (classes, activities, relationships, …)
  - UML diagrams that depict the UML semantic elements (and *may* also depict visual references to things in other semantic domains such as Java, C++, or DDL).

- **Implementation modeling files** are stored as Eclipse *projects* in an Eclipse workspace. The projects contain
  - implementation artifacts (Java source code, Web pages, XML-based metadata files, …)
  - diagram files that depict and directly reflect the implementation artifacts.

  The model semantics reside in the implementation artifacts themselves. For instance, the semantic model of a Java implementation is serialized and stored as a collection of Java source code files. Each diagram resides in its own physical file within the project. Diagram files can have a variety of extensions. The most common is ".dnx". The implementation modeling diagrams often use UML notation, but may also use other notations (e.g. IDEF1X or Information Engineering notations for data visualization, or IBM proprietary notations used for designing Web tiers).

The focus of this paper is how to organize the internal structures of "pre-implementation" models. **Within the remainder of the paper the term "modeling file" is reserved for "pre-implementation" modeling files** (files with .emx extensions). Guidance for organizing the contents of implementation projects may be found in other sources such as online help for Rational Software Architect, Rational Application

[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.

Developer, and Rational Web Developer Community Edition.

A modeling file does not necessarily contain all of the information for one (logical) model. In fact a modeling file will often contain only a subset of a model. In the example given above we had a team of three working on a use case model for a timesheet application. That 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 this paper discusses 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 RS*x*, modeling files are not "strongly typed" but you can follow a convention of using multiple modeling files that are "loosely typed". If you wish to follow this convention, you can establish loose typing in either of two ways:

- start with a "Blank" modeling file (see below) and establish its type by how you name it and what kind of content you place into it (including what UML profiles you apply to it)
- create a modeling file it based upon a pre-defined "template model" that represents a particular model type. The RSx products provide a default set of template models for the model types described in this paper. You can also create your own template models (refer to product Help and to forums and other resources on developerWorks).

Either way, the "type" of a modeling file in RSx is really just a matter of convention concerning the naming and 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 (which in logical terms, RUP would consider to be part of the analysis or design model).

These guidelines do suggest that you treat RSx modeling 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 grouped and managed within Workspaces. RS*x* modeling files reside within projects just like other files.

For purposes of this discussion, not all of the project types that are available in RSx and Rational Application Developer need be explained in detail. We are primarily interested in two categories of projects:

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

We stated earlier that RS*x* supports two kinds of modeling files:

- Files with .emx extensions that contain UML models used for modeling at levels of abstraction above implementation (requirements, analysis, design)
- Eclipse projects that contain implementation semantics (typically serialized as source code file artifacts), and diagrams that reflect those semantics.

The rule for allocating models to projects is simple:

a) place "pre-implementation" modeling files in UML projects

b) implementation models take care of themselves because in essence:

[implementation model] = [implementation project]

There are a couple of exceptions that make 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. So 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 RSx 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, RSD, and RAD simply let you draw diagrams of code semantics expressed in UML notations, dispensing with the approach of using a separately persisted semantic model at the implementation level of abstraction.

Note that RSx do not *prevent* you from defining UML models at a code level of abstraction and generating code from them, and in fact this type of usage is expected. But RSx do not provide technology for keeping such models auto-synchronized with code.

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.

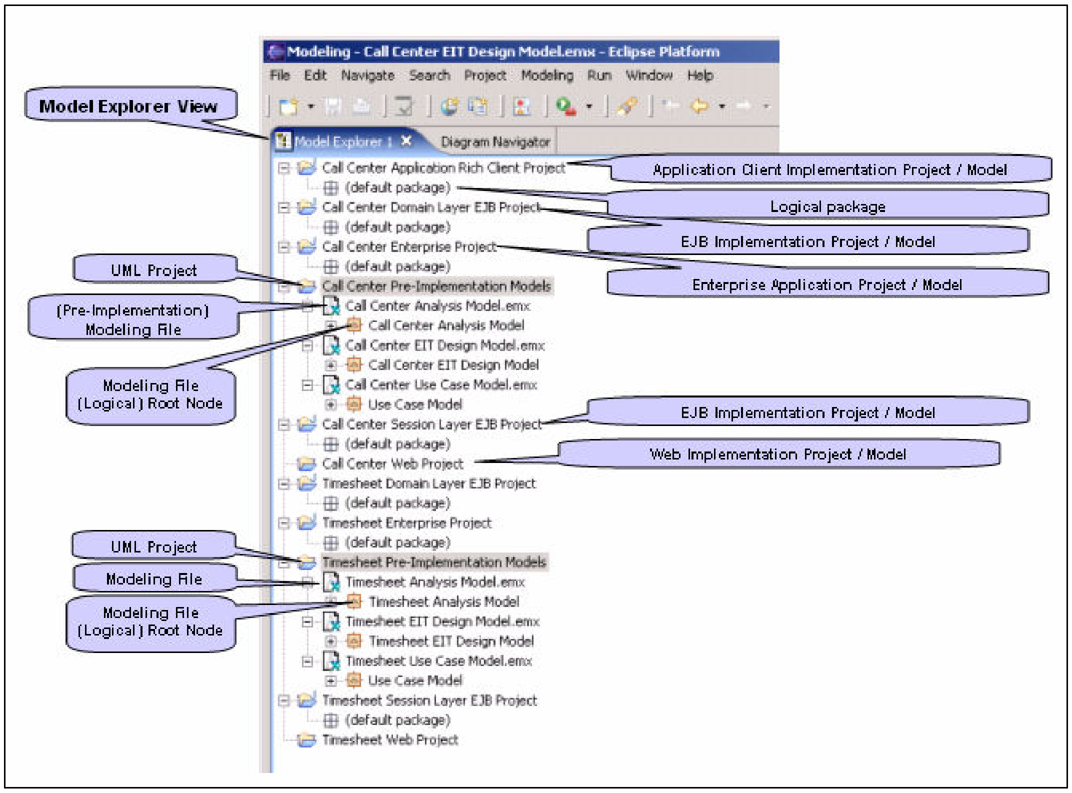

Modeling - Call Center EIT Design Model.emx - Eclipse Platform
File Edit Navigate Search Project Modeling Run Window Help
Model Explorer View
Model Explorer
Diagram Navigator
Call Center Application Rich Client Project
(default package)
Call Center Domain Layer EJB Project
(default package)
Call Center Enterprise Project
(default package)
Call Center Pre-Implementation Models
Call Center Analysis Model.emx
Call Center Analysis Model
Call Center EIT Design Model.emx
Call Center EIT Design Model
Call Center Use Case Model.emx
Use Case Model
Call Center Session Layer EJB Project
(default package)
Call Center Web Project
Timesheet Domain Layer EJB Project
(default package)
Timesheet Enterprise Project
(default package)
Timesheet Pre-Implementation Models
Timesheet Analysis Model.emx
Timesheet Analysis Model
Timesheet EIT Design Model.emx
Timesheet EIT Design Model
Timesheet Use Case Model.emx
Use Case Model
Timesheet Session Layer EJB Project
(default package)
Timesheet Web Project
Application Client Implementation Project / Model
Logical package
EJB Implementation Project / Model
Enterprise Application Project / Model
UML Project
(Pre-Implementation) Modeling File
Modeling File (Logical) Root Node
EJB Implementation Project / Model
Web Implementation Project / Model
UML Project
Modeling File
Modeling File (Logical) Root Node

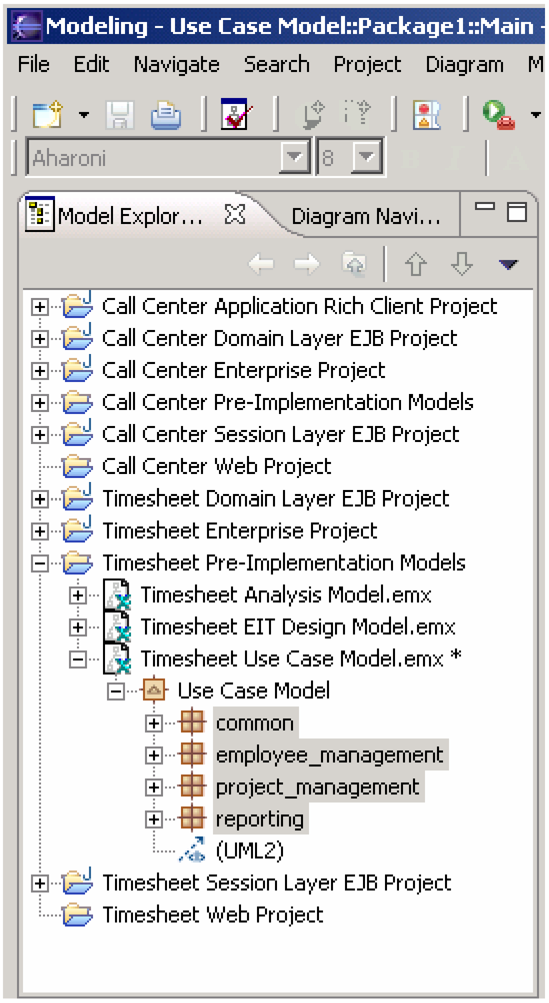
Figure 2-1

RSX provides an explorer view[2] that provides a combined physical and logical view of models. In the 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. So in **Figure 2-1** we see depicted in the Model Explorer a collection of projects that correspond to the two applications in our scenario. We see that UML projects have been used for the pre-implementation models, and we see that a collection of projects of solution-appropriate types 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.

In figure 1-2 we show how the Timesheet use case model might be internally organized into packages that represent some functionally cohesive subsets of the problem domain.

[2] In 6.x versions, this was the "Model Explorer". As of 7.0, models appear in the general-purpose explorer. Illustration in this paper are based on the 6.0 version and depict the "Model Explorer".


Modeling - Use Case Model::Package1::Main
File Edit Navigate Search Project Diagram M
Aharoni
8
Model Explor…
Diagram Navi…

Call Center Application Rich Client Project
Call Center Domain Layer EJB Project
Call Center Enterprise Project
Call Center Pre-Implementation Models
Call Center Session Layer EJB Project
Call Center Web Project
Timesheet Domain Layer EJB Project
Timesheet Enterprise Project
Timesheet Pre-Implementation Models
Timesheet Analysis Model.emx
Timesheet EIT Design Model.emx
Timesheet Use Case Model.emx *
Use Case Model
common
employee_management
project_management
reporting
(UML2)
Timesheet Session Layer EJB Project
Timesheet Web Project

Figure 2-2

In **Figure 2-1 and 1-2** each pre-implementation model resides in a single modeling file. In figure 1-3 we show how the Timesheet use-case model might be refactored into multiple modeling files that correspond to those same subsets of the problem domain. Notice how the root of each of the modeling files has been named to maintain a consistent namespace across all of the modeling files that make up the complete use case model.

3. RUP Model to RS*x* Model Mapping

The following table shows how the most commonly used RUP models can be mapped, by convention, to RS*x* modeling file "types". The mapping is generally straightforward, but it is the key to using this paper as a guide to practicing RUP with RSx. The RSx modeling file types mentioned in the table are discussed immediately following the table. Guidelines for internal organization of the various modeling file types, and what kinds of projects to keep them in, are provided in later sections. Those later discussions are presented in terms of the RSx modeling file types listed here.

RUP Model	***RSx* Modeling File Type**
Use-Case Model	Modeling file based on "Use-Case Model" template (alternate: start with blank modeling file, restrict content per RUP use-case model guidance)
Analysis Model	Analysis Model (alternate: start with blank modeling file, restrict content per RUP analysis model guidance) (alternate: use «analysis» packages in Design model)
Design Model	For n-tier business applications: modeling file based on "Enterprise IT Design Model" template (alternate: start with blank modeling file, restrict content per RUP design model guidance) For other types of applications: start with blank modeling file, restrict content per RUP design model guidance For design 'sketch': blank modeling file Optional supplement: additional blank modeling file used as Implementation Overview Model
Implementation Model	Eclipse projects containing implementation artifacts and diagram files
Deployment Model	Start with blank modeling file, restrict content per RUP deployment model guidance

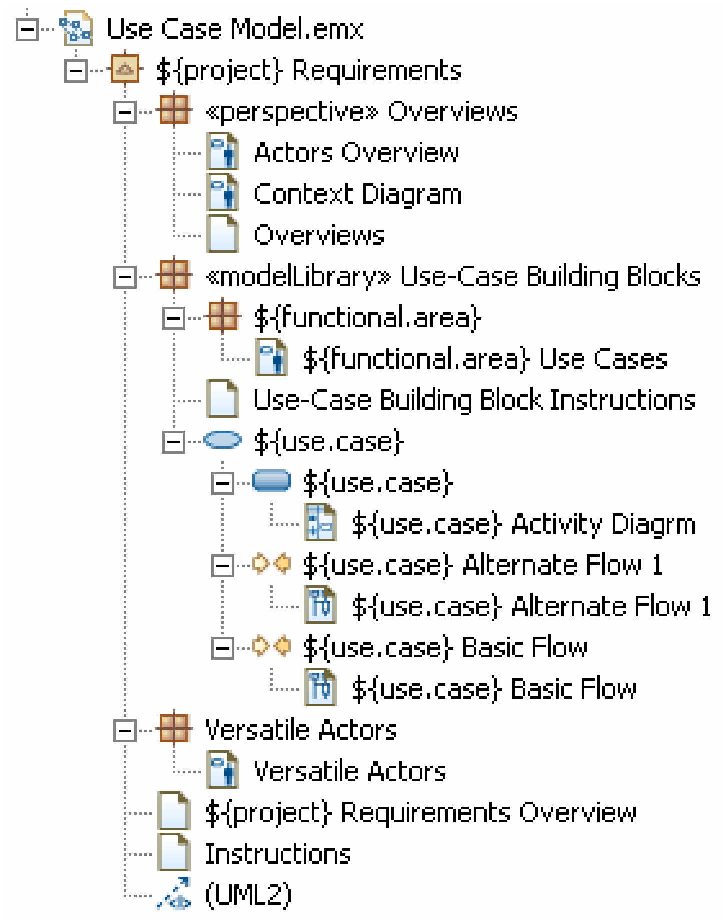
RS*x* Modeling File Types

Blank Modeling File

RSx 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 Modeling File

RSx 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’s 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.)

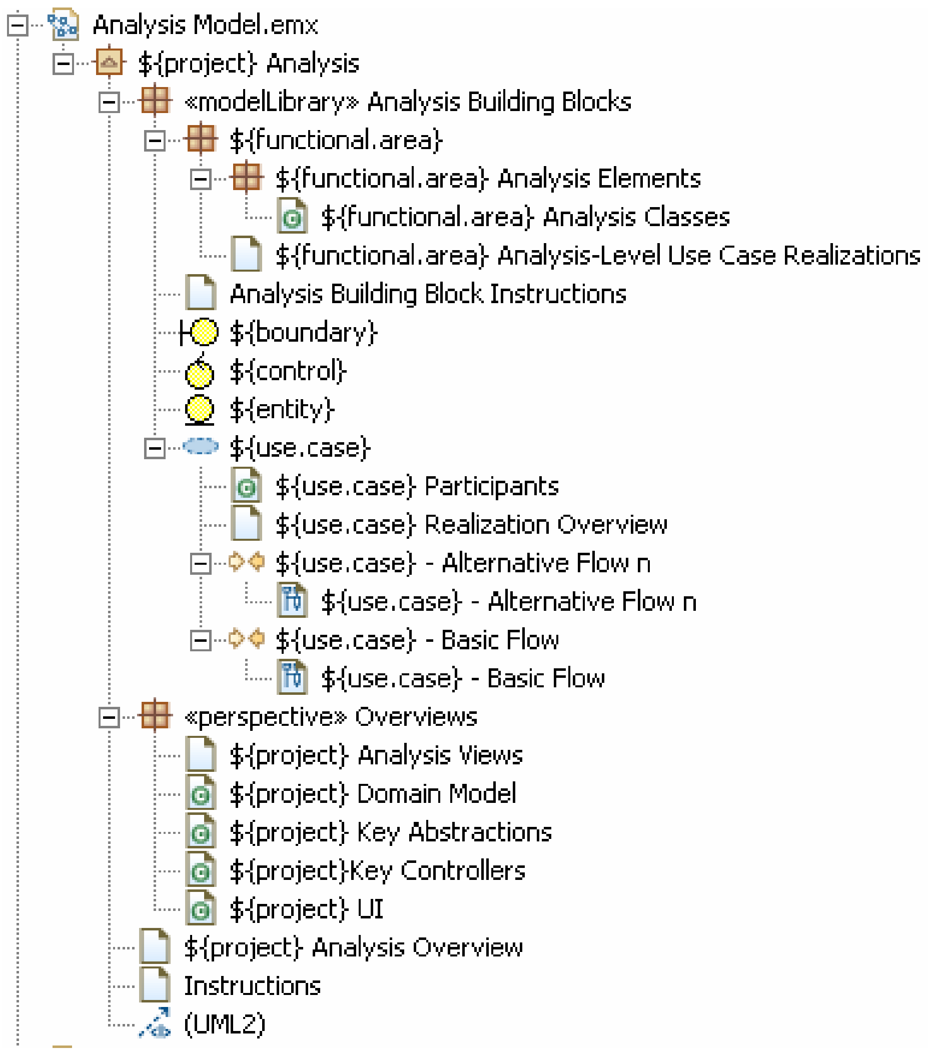


```
Use Case Model.emx
  ${project} Requirements
    «perspective» Overviews
      Actors Overview
      Context Diagram
      Overviews
    «modelLibrary» Use-Case Building Blocks
      ${functional.area}
        ${functional.area} Use Cases
      Use-Case Building Block Instructions
      ${use.case}
        ${use.case}
          ${use.case} Activity Diagrm
        ${use.case} Alternate Flow 1
          ${use.case} Alternate Flow 1
        ${use.case} Basic Flow
          ${use.case} Basic Flow
    Versatile Actors
      Versatile Actors
    ${project} Requirements Overview
    Instructions
    (UML2)
```

Figure 3-1

Analysis Modeling File

RSx 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:

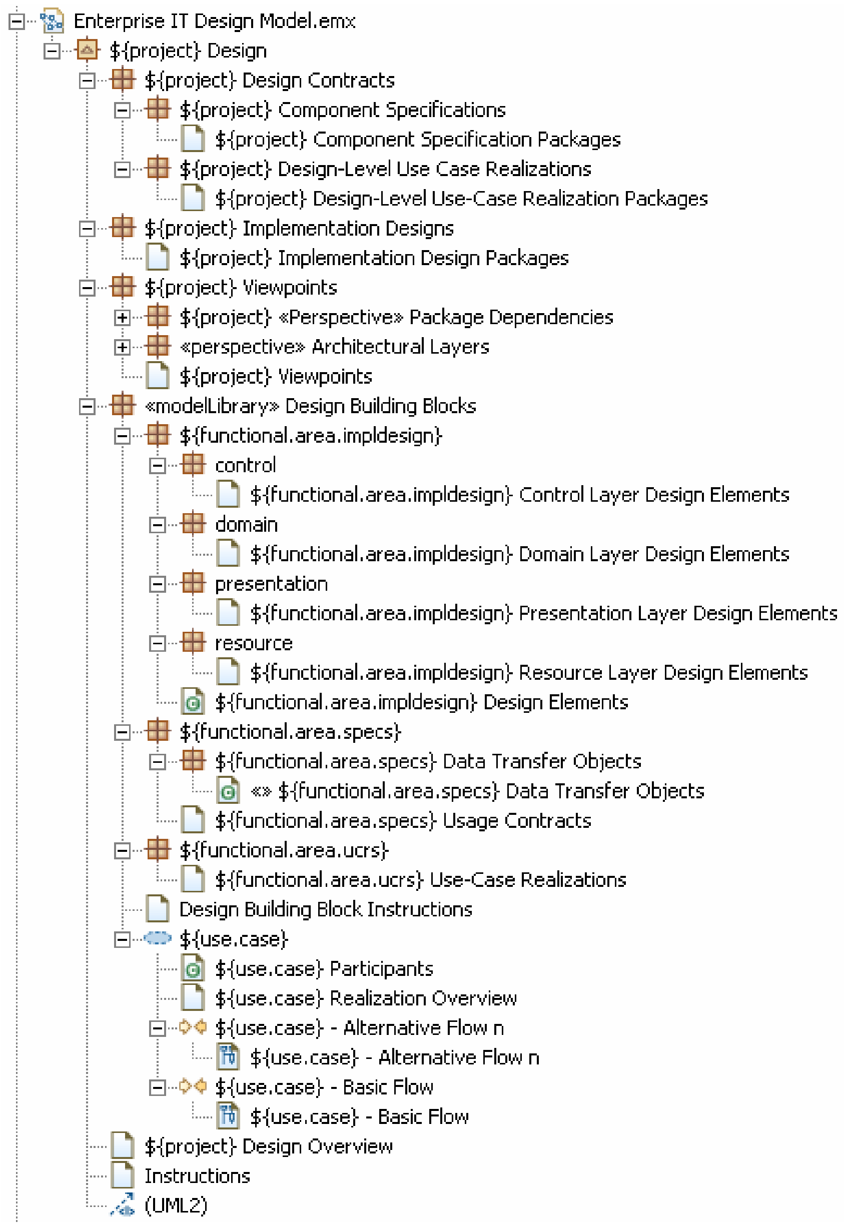


```
Analysis Model.emx
  ${project} Analysis
    «modelLibrary» Analysis Building Blocks
      ${functional.area}
        ${functional.area} Analysis Elements
          ${functional.area} Analysis Classes
        ${functional.area} Analysis-Level Use Case Realizations
      Analysis Building Block Instructions
      ${boundary}
      ${control}
      ${entity}
      ${use.case}
        ${use.case} Participants
        ${use.case} Realization Overview
        ${use.case} - Alternative Flow n
          ${use.case} - Alternative Flow n
        ${use.case} - Basic Flow
          ${use.case} - Basic Flow
    «perspective» Overviews
      ${project} Analysis Views
      ${project} Domain Model
      ${project} Key Abstractions
      ${project}Key Controllers
      ${project} UI
    ${project} Analysis Overview
    Instructions
    (UML2)
```

Figure 3-2

Enterprise IT Design Modeling File

RSx 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[3] 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 *RSX* code-generating transformations to support creation of such applications.



```
Enterprise IT Design Model.emx
    ${project} Design
        ${project} Design Contracts
            ${project} Component Specifications
                ${project} Component Specification Packages
            ${project} Design-Level Use Case Realizations
                ${project} Design-Level Use-Case Realization Packages
        ${project} Implementation Designs
            ${project} Implementation Design Packages
        ${project} Viewpoints
            ${project} «Perspective» Package Dependencies
            «perspective» Architectural Layers
            ${project} Viewpoints
        «modelLibrary» Design Building Blocks
            ${functional.area.impldesign}
                control
                    ${functional.area.impldesign} Control Layer Design Elements
                domain
                    ${functional.area.impldesign} Domain Layer Design Elements
                presentation
                    ${functional.area.impldesign} Presentation Layer Design Elements
                resource
                    ${functional.area.impldesign} Resource Layer Design Elements
                ${functional.area.impldesign} Design Elements
            ${functional.area.specs}
                ${functional.area.specs} Data Transfer Objects
                    «» ${functional.area.specs} Data Transfer Objects
                ${functional.area.specs} Usage Contracts
            ${functional.area.ucrs}
                ${functional.area.ucrs} Use-Case Realizations
            Design Building Block Instructions
            ${use.case}
                ${use.case} Participants
                ${use.case} Realization Overview
                ${use.case} - Alternative Flow n
                    ${use.case} - Alternative Flow n
                ${use.case} - Basic Flow
                    ${use.case} - Basic Flow
        ${project} Design Overview
        Instructions
        (UML2)
```

Figure 3-3

[3] 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 Modeling File

As part of your design model, In addition you may 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 RSx or Rational Application Developer projects and folders/packages in which you expect the code and related files (metadata, deployment descriptors, etc.) 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 RS*x* an implementation model consists of a project containing implementation artifacts and (optionally) diagrams that depict those artifacts[4].

"Sketch" Models

As noted in the "Basic Concepts and Terminology" section, you can 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 can treat them like sketches that serve to suggest a design and help clarify and communicate it, but are disposable once implementation has begun. RSx supports both approaches. Its features generally do not target one approach or the other, but your choices about how to use design models certainly help determine which RS*x* 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.

[4] 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 may 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, etc.
  - 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 may 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. RSx 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. RSx 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 1471- 2000 "Viewpoint".

Do not place semantic elements (classes, packages, associations, etc.) 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 RSx 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 may 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 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 RSx 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 on the package, which will open its default diagram.

These mechanisms support the following organizational **guideline**, 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 don’t 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

Use Case Model High-Level Organization

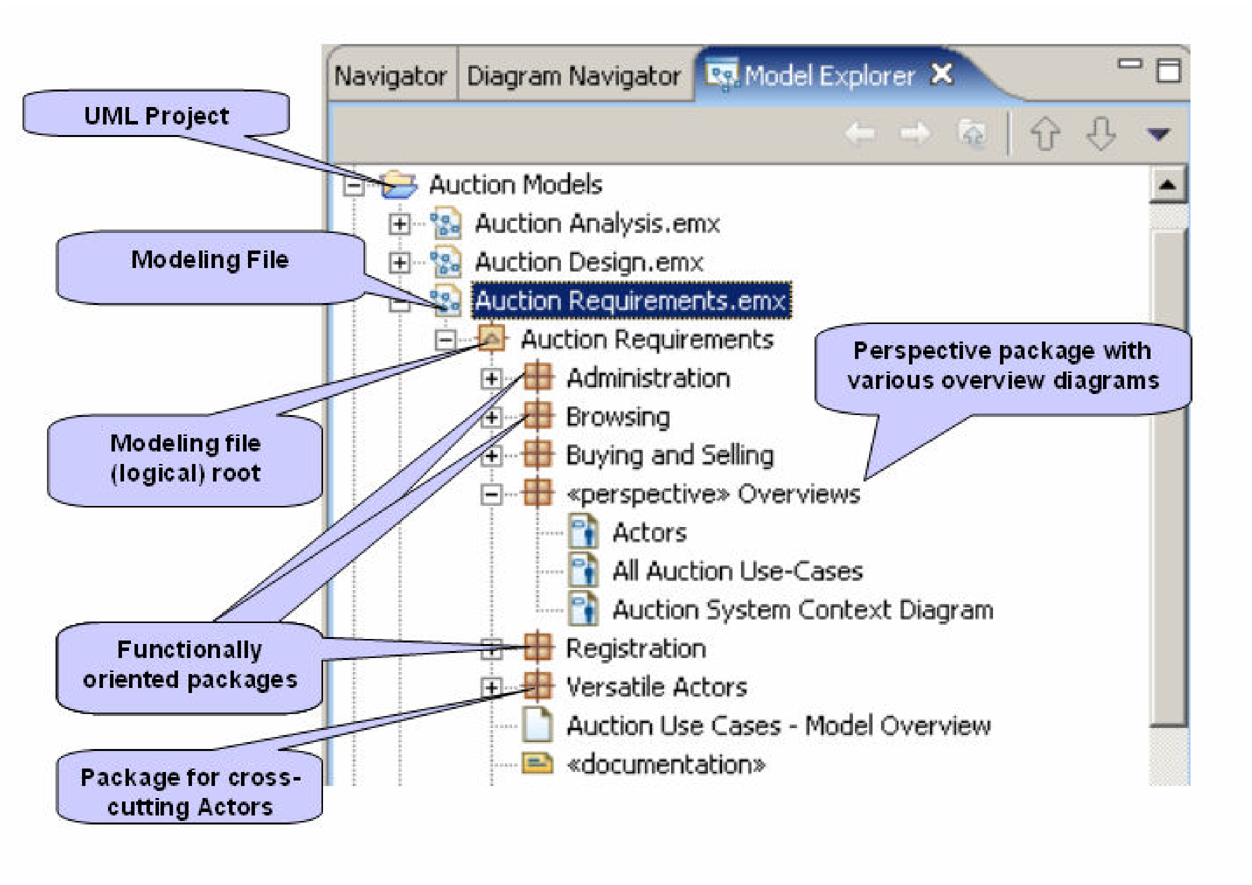


Figure 5-1

Figure 5-1 above 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).

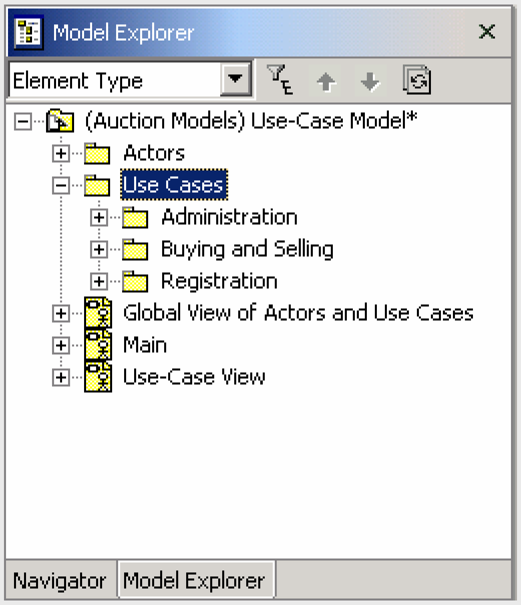
   - Compared to other organizational approaches, this will generally map better to the organization of the eventual implementation. This is important if you will be using transformations to seed each successive lower level of abstraction. Specifically, if you will be generating seed content into an analysis 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. And 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
RSX 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:



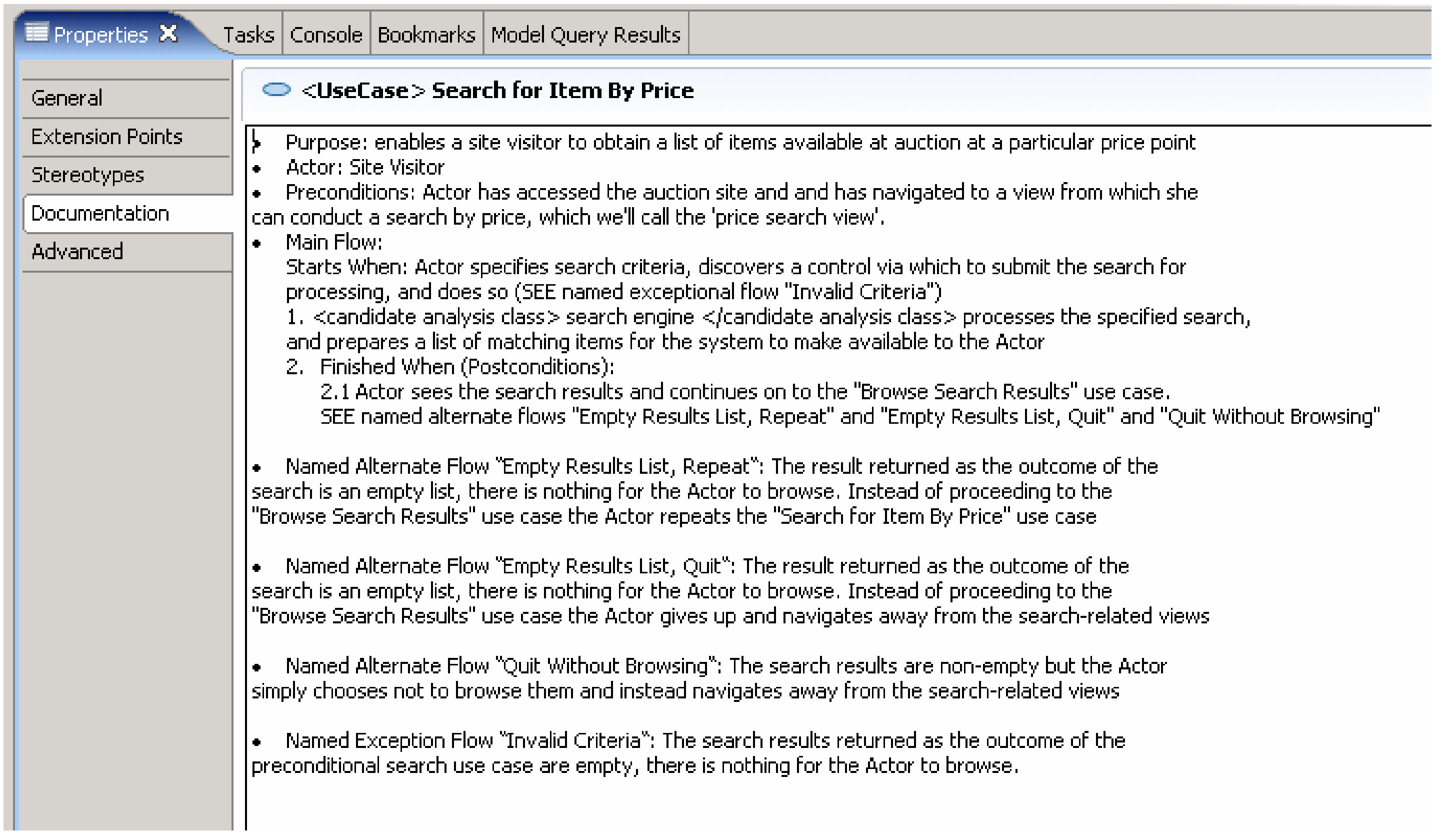
Use Case Model Content

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[5] (see **Figure 5-2**)



- Purpose: enables a site visitor to obtain a list of items available at auction at a particular price point
- Actor: Site Visitor
- Preconditions: Actor has accessed the auction site and and has navigated to a view from which she can conduct a search by price, which we'll call the 'price search view'.
- Main Flow:
  Starts When: Actor specifies search criteria, discovers a control via which to submit the search for processing, and does so (SEE named exceptional flow "Invalid Criteria")
  1. <candidate analysis class> search engine </candidate analysis class> processes the specified search, and prepares a list of matching items for the system to make available to the Actor
  2. Finished When (Postconditions):
     2.1 Actor sees the search results and continues on to the "Browse Search Results" use case.
     SEE named alternate flows "Empty Results List, Repeat" and "Empty Results List, Quit" and "Quit Without Browsing"

- Named Alternate Flow "Empty Results List, Repeat": The result returned as the outcome of the search is an empty list, there is nothing for the Actor to browse. Instead of proceeding to the "Browse Search Results" use case the Actor repeats the "Search for Item By Price" use case

- Named Alternate Flow "Empty Results List, Quit": The result returned as the outcome of the search is an empty list, there is nothing for the Actor to browse. Instead of proceeding to the "Browse Search Results" use case the Actor gives up and navigates away from the search-related views

- Named Alternate Flow "Quit Without Browsing": The search results are non-empty but the Actor simply chooses not to browse them and instead navigates away from the search-related views

- Named Exception Flow "Invalid Criteria": The search results returned as the outcome of the preconditional search use case are empty, there is nothing for the Actor to browse.

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 RS*x* 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 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 may also help you identify the various views (screens or pages) that will be required as part of the

[5] 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.

implementation. It also formally establishes the naming of the use case's various flows (scenarios) by assigning those names to semantic model elements (i.e. to the collaboration occurrences).

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

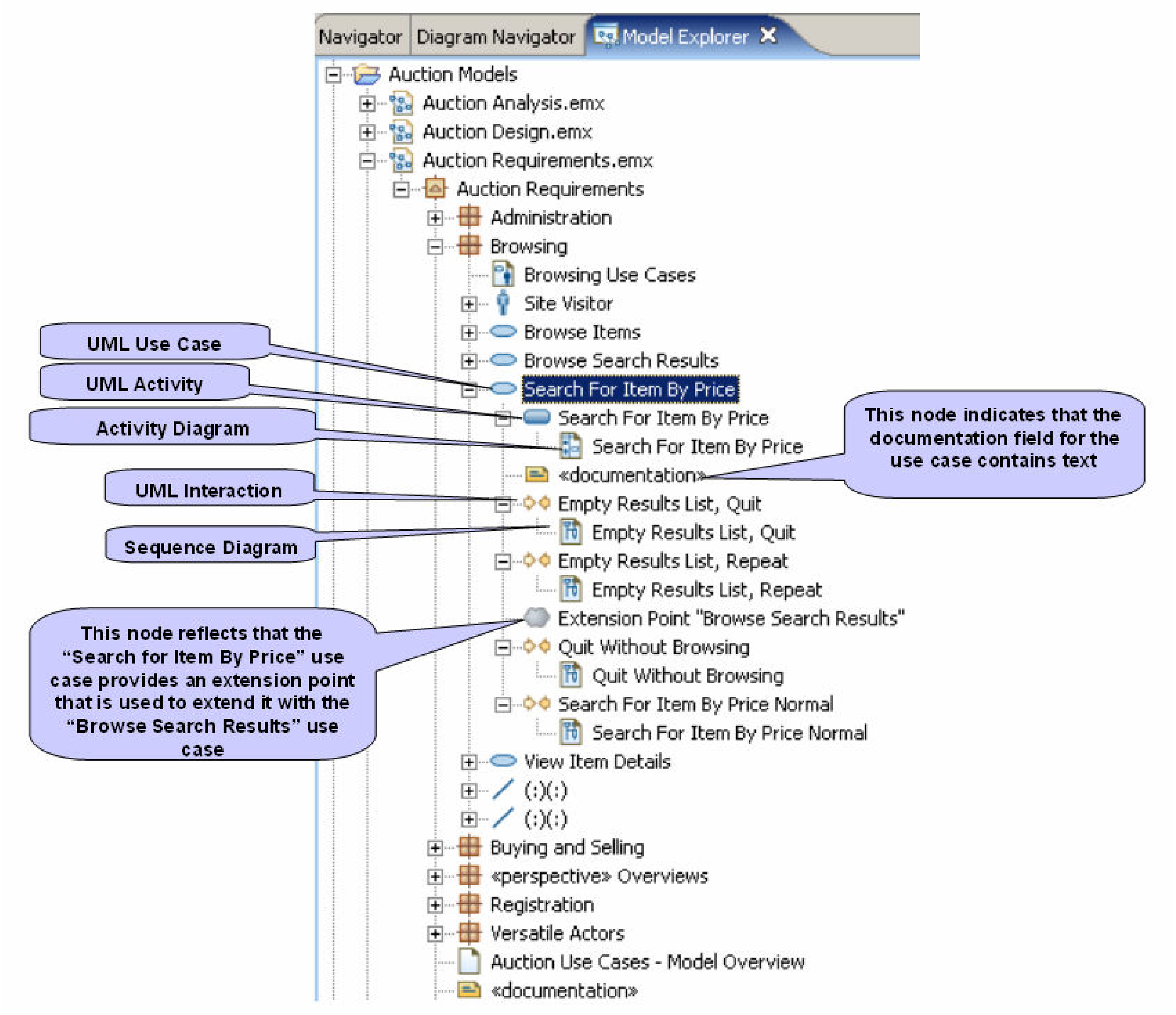


Figure 5-3

- **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 may 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.

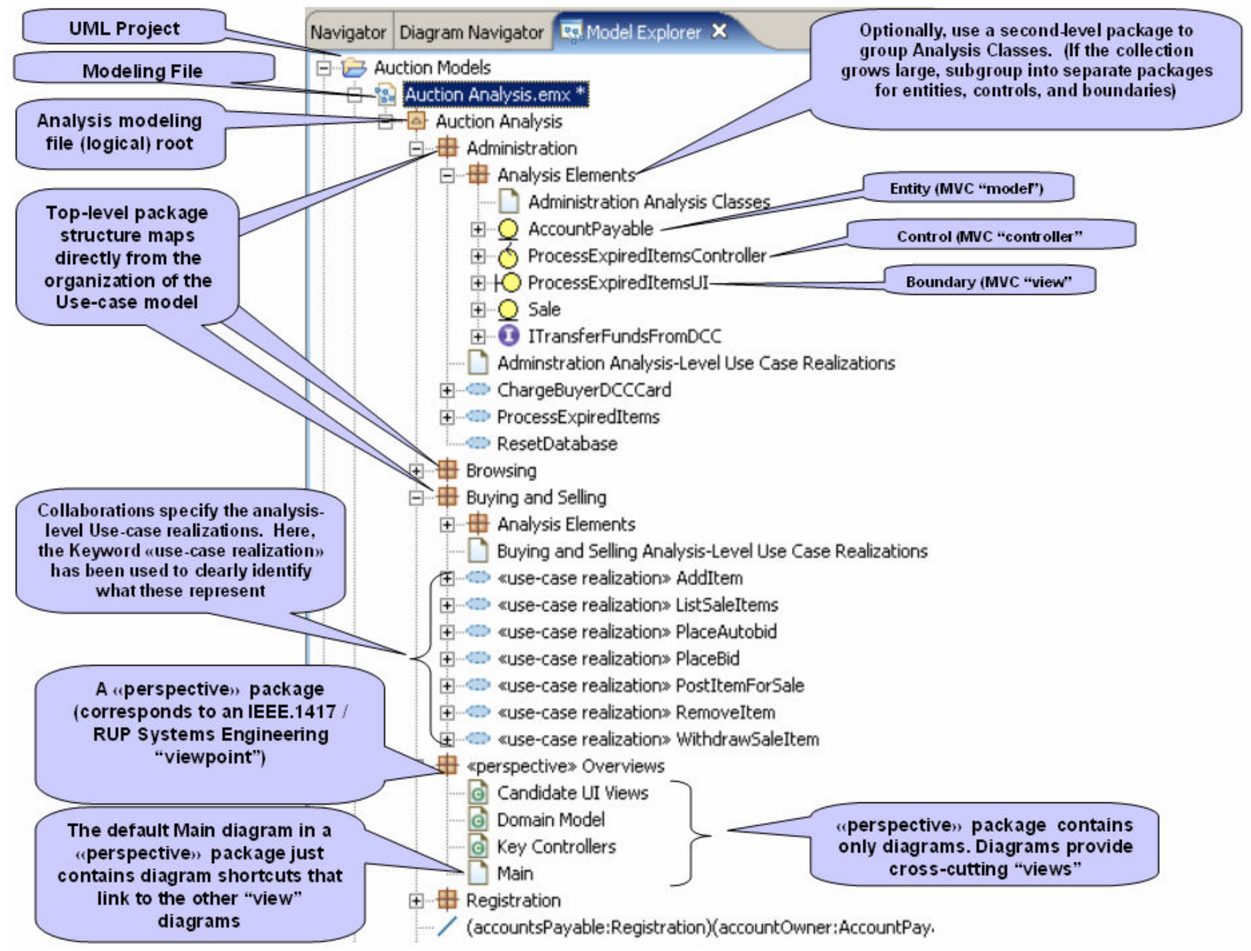
In RUP, whether or not an Analysis Model should be maintained independently of the Design Model is a project-specific decision, one you make on the basis of whether you believe the value of maintaining the separate Analysis Model will warrant the time invested. If a separate Analysis Model is created, 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[6]. 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 maintain 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, 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 RSx 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 RSx in this manner, options 2 or 3 above are preferred. The standard code-generating transformations packaged as part of RSx will bypass model packages that have the «analysis» keyword.

[6] 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


UML Project
Modeling File
Analysis modeling file (logical) root
Top-level package structure maps directly from the organization of the Use-case model
Collaborations specify the analysis-level Use-case realizations. Here, the Keyword «use-case realization» has been used to clearly identify what these represent
A «perspective» package (corresponds to an IEEE.1417 / RUP Systems Engineering "viewpoint")
The default Main diagram in a «perspective» package just contains diagram shortcuts that link to the other "view" diagrams
Optionally, use a second-level package to group Analysis Classes. (If the collection grows large, subgroup into separate packages for entities, controls, and boundaries)
Entity (MVC "model")
Control (MVC "controller")
Boundary (MVC "view")
«perspective» package contains only diagrams. Diagrams provide cross-cutting "views"

Navigator | Diagram Navigator | Model Explorer

```
Auction Models
  Auction Analysis.emx *
    Auction Analysis
      Administration
        Analysis Elements
          Administration Analysis Classes
          AccountPayable
          ProcessExpiredItemsController
          ProcessExpiredItemsUI
          Sale
          ITransferFundsFromDCC
        Adminstration Analysis-Level Use Case Realizations
        ChargeBuyerDCCCard
        ProcessExpiredItems
        ResetDatabase
      Browsing
      Buying and Selling
        Analysis Elements
        Buying and Selling Analysis-Level Use Case Realizations
        «use-case realization» AddItem
        «use-case realization» ListSaleItems
        «use-case realization» PlaceAutobid
        «use-case realization» PlaceBid
        «use-case realization» PostItemForSale
        «use-case realization» RemoveItem
        «use-case realization» WithdrawSaleItem
      «perspective» Overviews
        Candidate UI Views
        Domain Model
        Key Controllers
        Main
      Registration
      (accountsPayable:Registration)(accountOwner:AccountPay
```

Figure 6-1

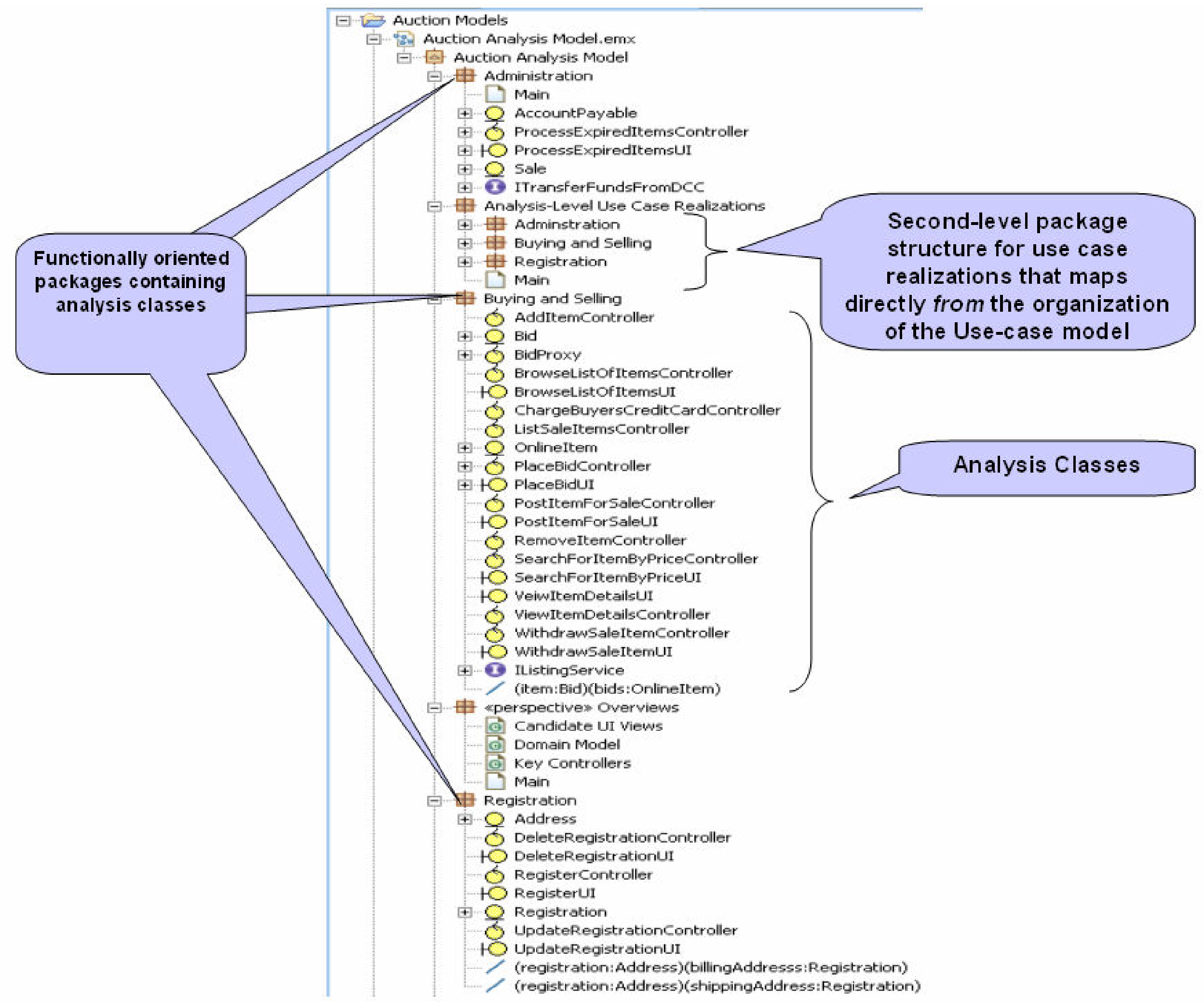
Figure 6-1 above illustrates the following guidelines for structuring analysis models:

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 below in **Figure 6-2** where we see the use of a top-level package to segregate the use-case realizations from the analysis classes. Within that top-level package we see 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.)


Functionally oriented packages containing analysis classes
Second-level package structure for use case realizations that maps directly *from* the organization of the Use-case model
Analysis Classes

```
Auction Models
  Auction Analysis Model.emx
    Auction Analysis Model
      Administration
        Main
        AccountPayable
        ProcessExpiredItemsController
        ProcessExpiredItemsUI
        Sale
        ITransferFundsFromDCC
      Analysis-Level Use Case Realizations
        Adminstration
        Buying and Selling
        Registration
        Main
      Buying and Selling
        AddItemController
        Bid
        BidProxy
        BrowseListOfItemsController
        BrowseListOfItemsUI
        ChargeBuyersCreditCardController
        ListSaleItemsController
        OnlineItem
        PlaceBidController
        PlaceBidUI
        PostItemForSaleController
        PostItemForSaleUI
        RemoveItemController
        SearchForItemByPriceController
        SearchForItemByPriceUI
        VeiwItemDetailsUI
        ViewItemDetailsController
        WithdrawSaleItemController
        WithdrawSaleItemUI
        IListingService
        (item:Bid)(bids:OnlineItem)
      «perspective» Overviews
        Candidate UI Views
        Domain Model
        Key Controllers
        Main
      Registration
        Address
        DeleteRegistrationController
        DeleteRegistrationUI
        RegisterController
        RegisterUI
        Registration
        UpdateRegistrationController
        UpdateRegistrationUI
        (registration:Address)(billingAddresss:Registration)
        (registration:Address)(shippingAddress:Registration)
```

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 below 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 may map well to the organization of code generated from the analysis/design, it may simplify subsequent configuration of the code-generating transformations.

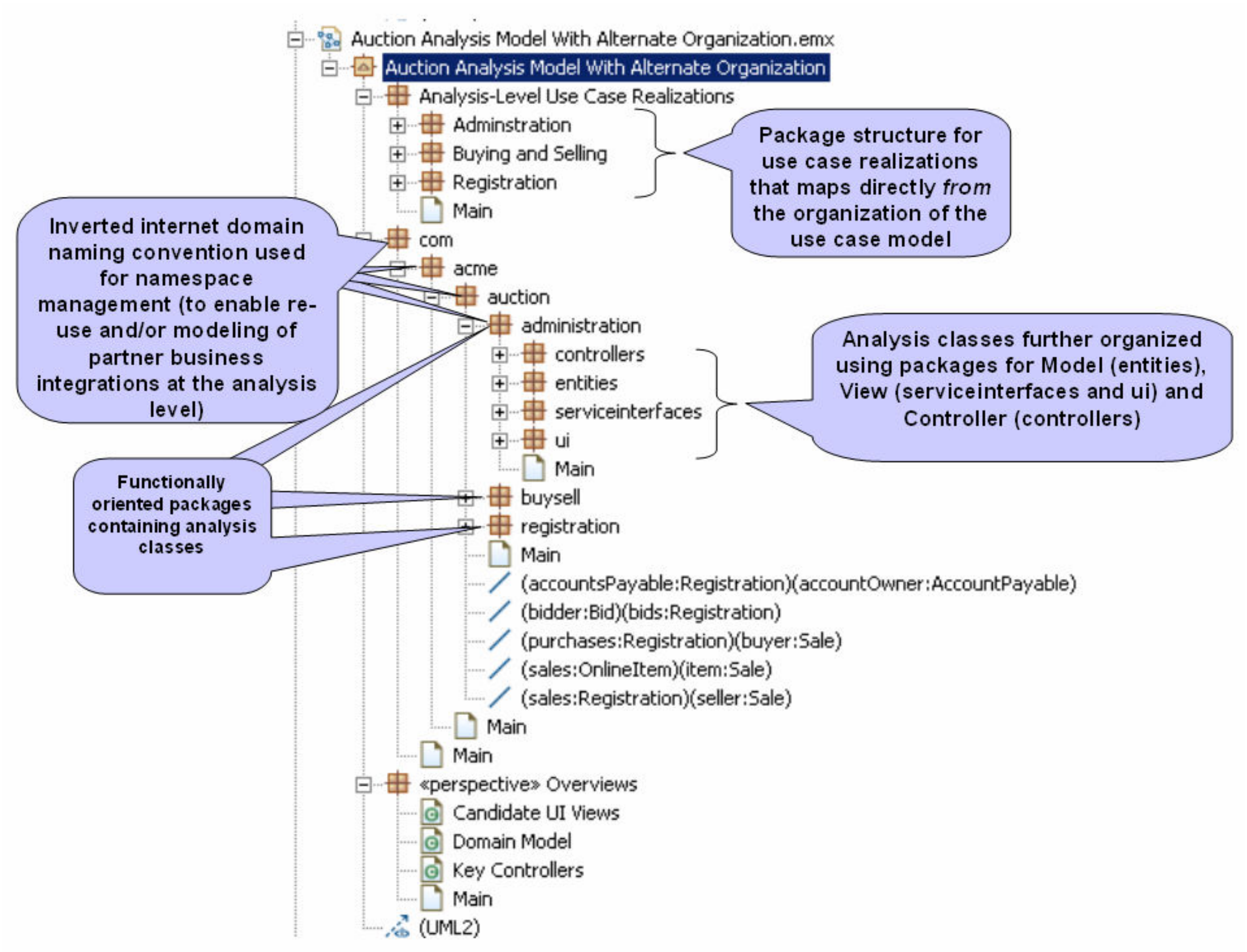

Auction Analysis Model With Alternate Organization.emx
Auction Analysis Model With Alternate Organization
Analysis-Level Use Case Realizations
Adminstration
Buying and Selling
Registration
Main
com
acme
auction
administration
controllers
entities
serviceinterfaces
ui
Main
buysell
registration
Main
(accountsPayable:Registration)(accountOwner:AccountPayable)
(bidder:Bid)(bids:Registration)
(purchases:Registration)(buyer:Sale)
(sales:OnlineItem)(item:Sale)
(sales:Registration)(seller:Sale)
Main
Main
«perspective» Overviews
Candidate UI Views
Domain Model
Key Controllers
Main
(UML2)
Package structure for use case realizations that maps directly *from* the organization of the use case model
Inverted internet domain naming convention used for namespace management (to enable re-use and/or modeling of partner business integrations at the analysis level)
Analysis classes further organized using packages for Model (entities), View (serviceinterfaces and ui) and Controller (controllers)
Functionally oriented packages containing analysis classes

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**)

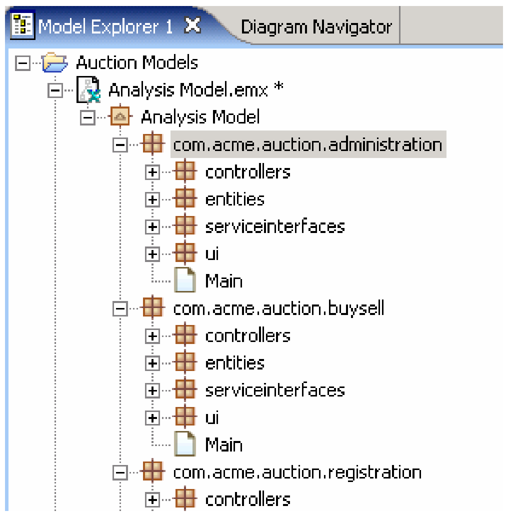
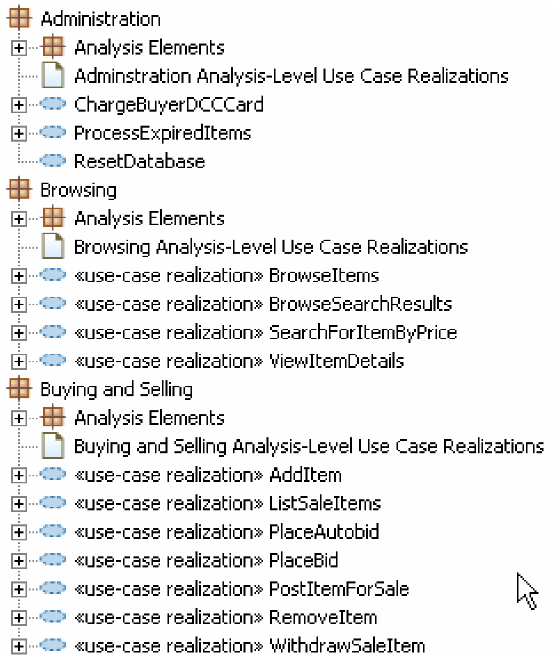


Figure 6-4

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[7] 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**)

[7] As previously established in the Use Case Model



Administration
- Analysis Elements
- Adminstration Analysis-Level Use Case Realizations
- ChargeBuyerDCCCard
- ProcessExpiredItems
- ResetDatabase

Browsing
- Analysis Elements
- Browsing Analysis-Level Use Case Realizations
- «use-case realization» BrowseItems
- «use-case realization» BrowseSearchResults
- «use-case realization» SearchForItemByPrice
- «use-case realization» ViewItemDetails

Buying and Selling
- Analysis Elements
- Buying and Selling Analysis-Level Use Case Realizations
- «use-case realization» AddItem
- «use-case realization» ListSaleItems
- «use-case realization» PlaceAutobid
- «use-case realization» PlaceBid
- «use-case realization» PostItemForSale
- «use-case realization» RemoveItem
- «use-case realization» WithdrawSaleItem

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 shows 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**

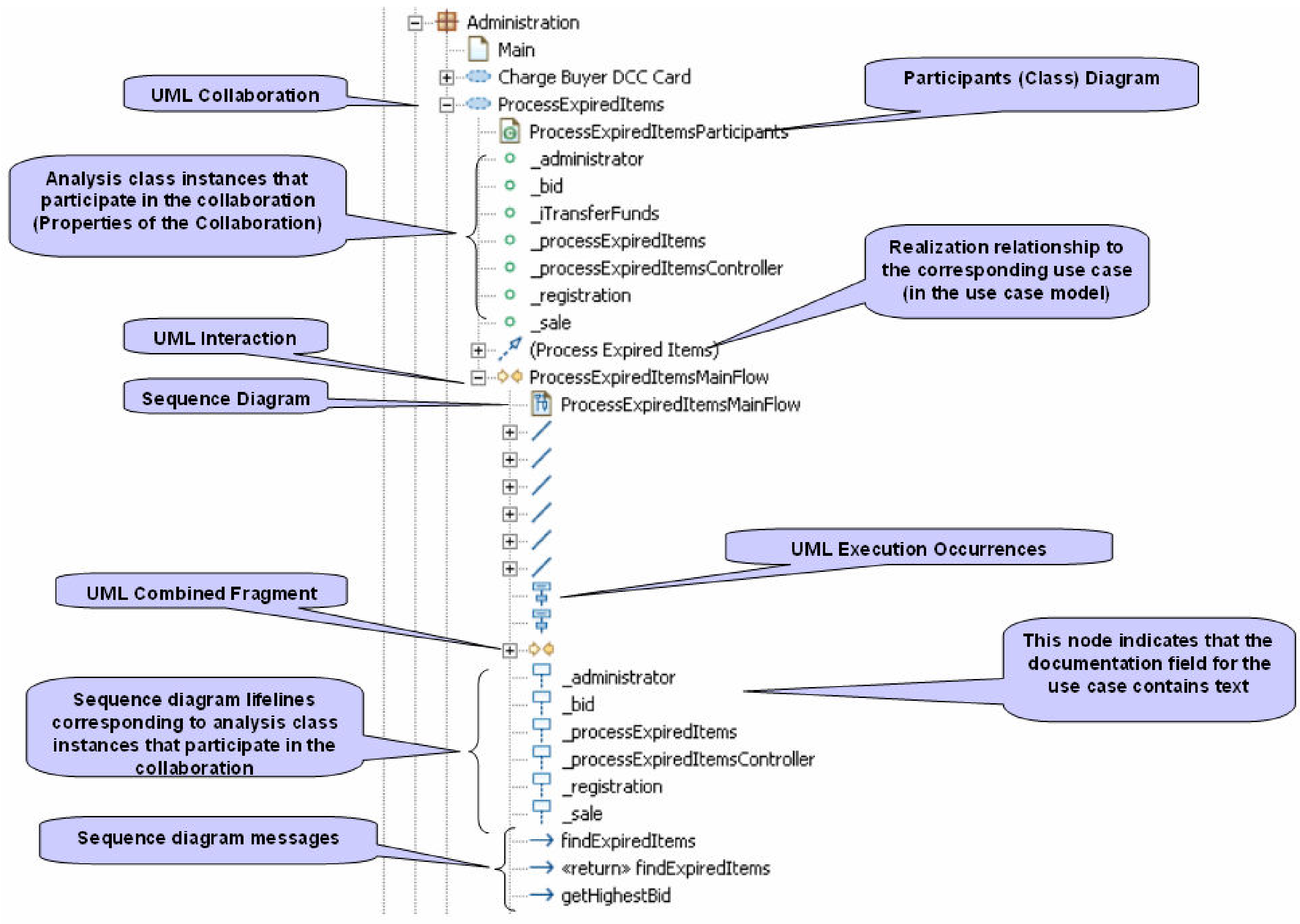
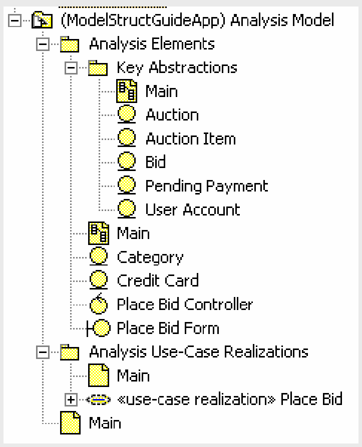

Administration
Main
Charge Buyer DCC Card
ProcessExpiredItems
ProcessExpiredItemsParticipants
_administrator
_bid
_iTransferFunds
_processExpiredItems
_processExpiredItemsController
_registration
_sale
(Process Expired Items)
ProcessExpiredItemsMainFlow
ProcessExpiredItemsMainFlow
_administrator
_bid
_processExpiredItems
_processExpiredItemsController
_registration
_sale
findExpiredItems
«return» findExpiredItems
getHighestBid
UML Collaboration
Participants (Class) Diagram
Analysis class instances that participate in the collaboration (Properties of the Collaboration)
Realization relationship to the corresponding use case (in the use case model)
UML Interaction
Sequence Diagram
UML Execution Occurrences
UML Combined Fragment
This node indicates that the documentation field for the use case contains text
Sequence diagram lifelines corresponding to analysis class instances that participate in the collaboration
Sequence diagram messages

Figure 6-6

XDE/Rose
The traditionally recommended structure for the **Analysis Model** as shown below is modified for RSx 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

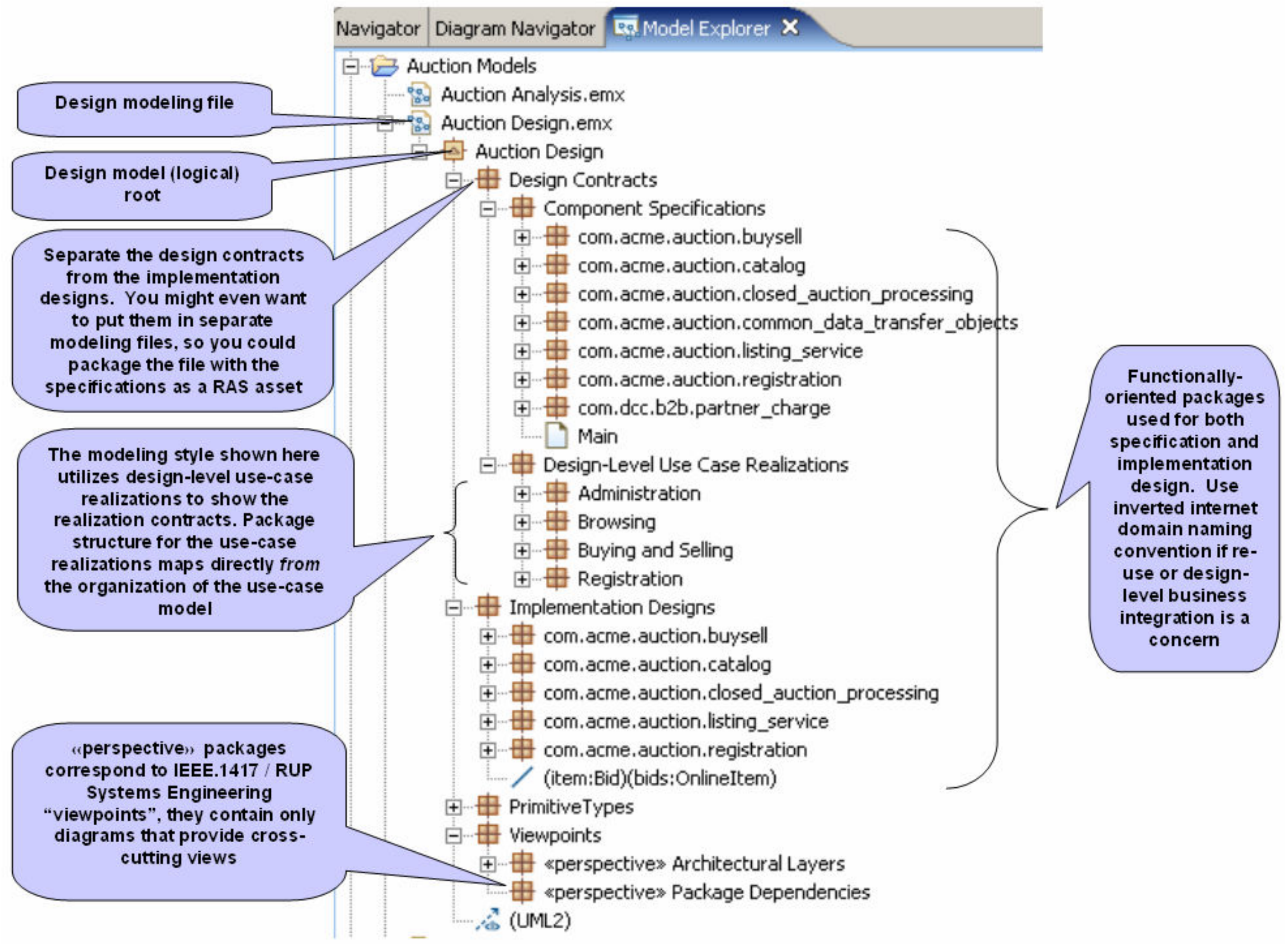

Navigator
Diagram Navigator
Model Explorer
Auction Models
Auction Analysis.emx
Auction Design.emx
Auction Design
Design Contracts
Component Specifications
com.acme.auction.buysell
com.acme.auction.catalog
com.acme.auction.closed_auction_processing
com.acme.auction.common_data_transfer_objects
com.acme.auction.listing_service
com.acme.auction.registration
com.dcc.b2b.partner_charge
Main
Design-Level Use Case Realizations
Administration
Browsing
Buying and Selling
Registration
Implementation Designs
com.acme.auction.buysell
com.acme.auction.catalog
com.acme.auction.closed_auction_processing
com.acme.auction.listing_service
com.acme.auction.registration
(item:Bid)(bids:OnlineItem)
PrimitiveTypes
Viewpoints
«perspective» Architectural Layers
«perspective» Package Dependencies
(UML2)
Design modeling file
Design model (logical) root
Separate the design contracts from the implementation designs. You might even want to put them in separate modeling files, so you could package the file with the specifications as a RAS asset
The modeling style shown here utilizes design-level use-case realizations to show the realization contracts. Package structure for the use-case realizations maps directly *from* the organization of the use-case model
«perspective» packages correspond to IEEE.1417 / RUP Systems Engineering "viewpoints", they contain only diagrams that provide cross-cutting views
Functionally-oriented packages used for both specification and implementation design. Use inverted internet domain naming convention if re-use or design-level business integration is a concern

Figure 7-1

Figure 7-1 above 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 may 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 may contain packages.   So in UML 2 «subsystem» Components are viable organizational/namespace alternatives to packages, yet UML2 is vague about appropriate use of subsystem vs. package. Suggestion: use Packages at levels of granularity such as the design subsystems of a particular application, and reserve Subsystems to represent entire applications (e.g. 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 may 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 "don't use spaces or punctuation other than underscores in the names".)

8. Use lower case 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»[8].
    - A) you have 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[9]. In fact (as described in the prior section "Guidelines for Internal Organization of the Analysis Model") you may 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 projects.[10] 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 only advisable 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

[8] Such packages will be bypassed by transformations.

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

[10] 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.

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.

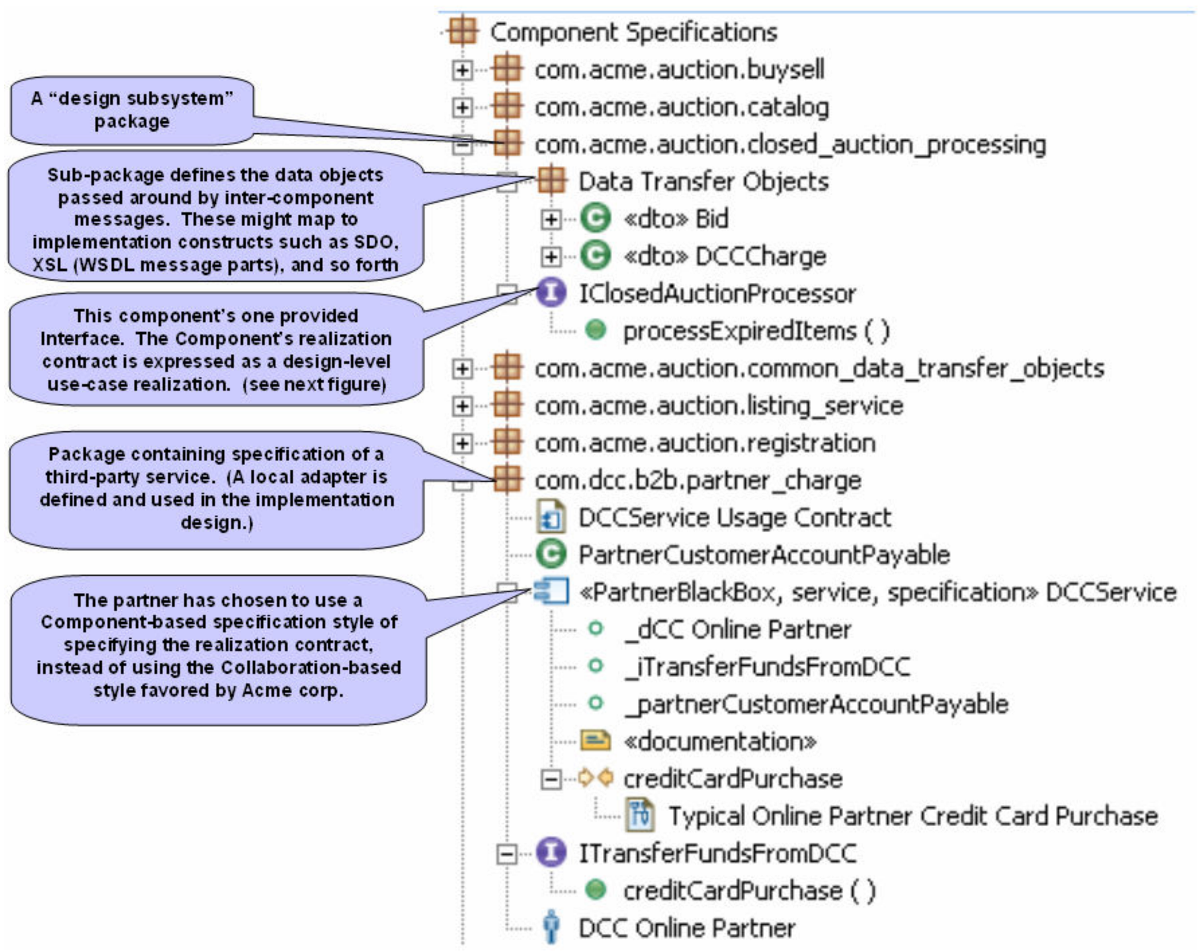

A "design subsystem" package
Sub-package defines the data objects passed around by inter-component messages. These might map to implementation constructs such as SDO, XSL (WSDL message parts), and so forth
This component's one provided Interface. The Component's realization contract is expressed as a 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

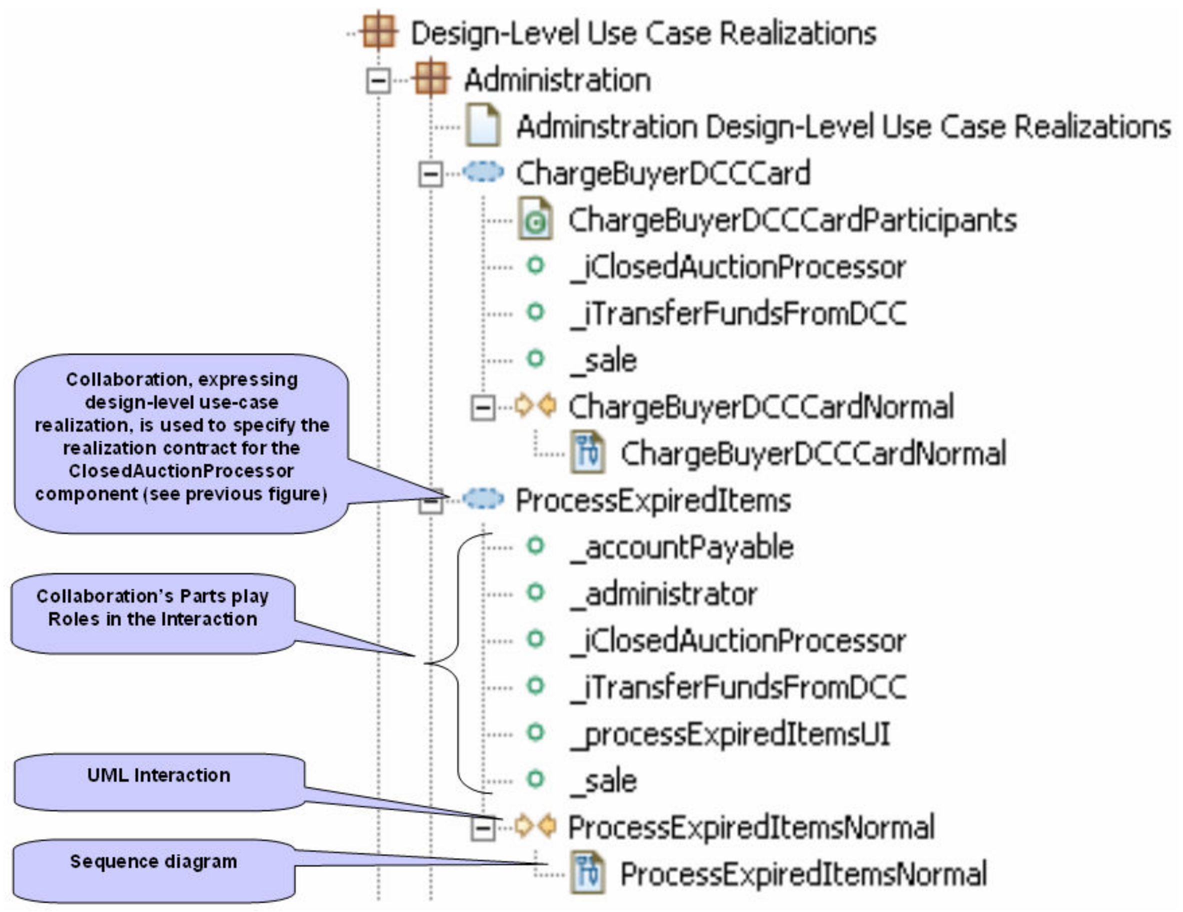

Design-Level Use Case Realizations
Administration
Adminstration Design-Level Use Case Realizations
ChargeBuyerDCCCard
ChargeBuyerDCCCardParticipants
_iClosedAuctionProcessor
_iTransferFundsFromDCC
_sale
ChargeBuyerDCCCardNormal
ChargeBuyerDCCCardNormal
ProcessExpiredItems
_accountPayable
_administrator
_iClosedAuctionProcessor
_iTransferFundsFromDCC
_processExpiredItemsUI
_sale
ProcessExpiredItemsNormal
ProcessExpiredItemsNormal
Collaboration, expressing design-level use-case realization, is used to specify the realization contract for the ClosedAuctionProcessor component (see previous figure)
Collaboration's Parts play Roles in the Interaction
UML Interaction
Sequence diagram

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[11] (**Figure 7-2**). The corresponding realization contract is specified by a single design level use-case realization expressed as a Collaboration[12] (**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[13]. 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.

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

[12] 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.

[13] 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[14]. 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.

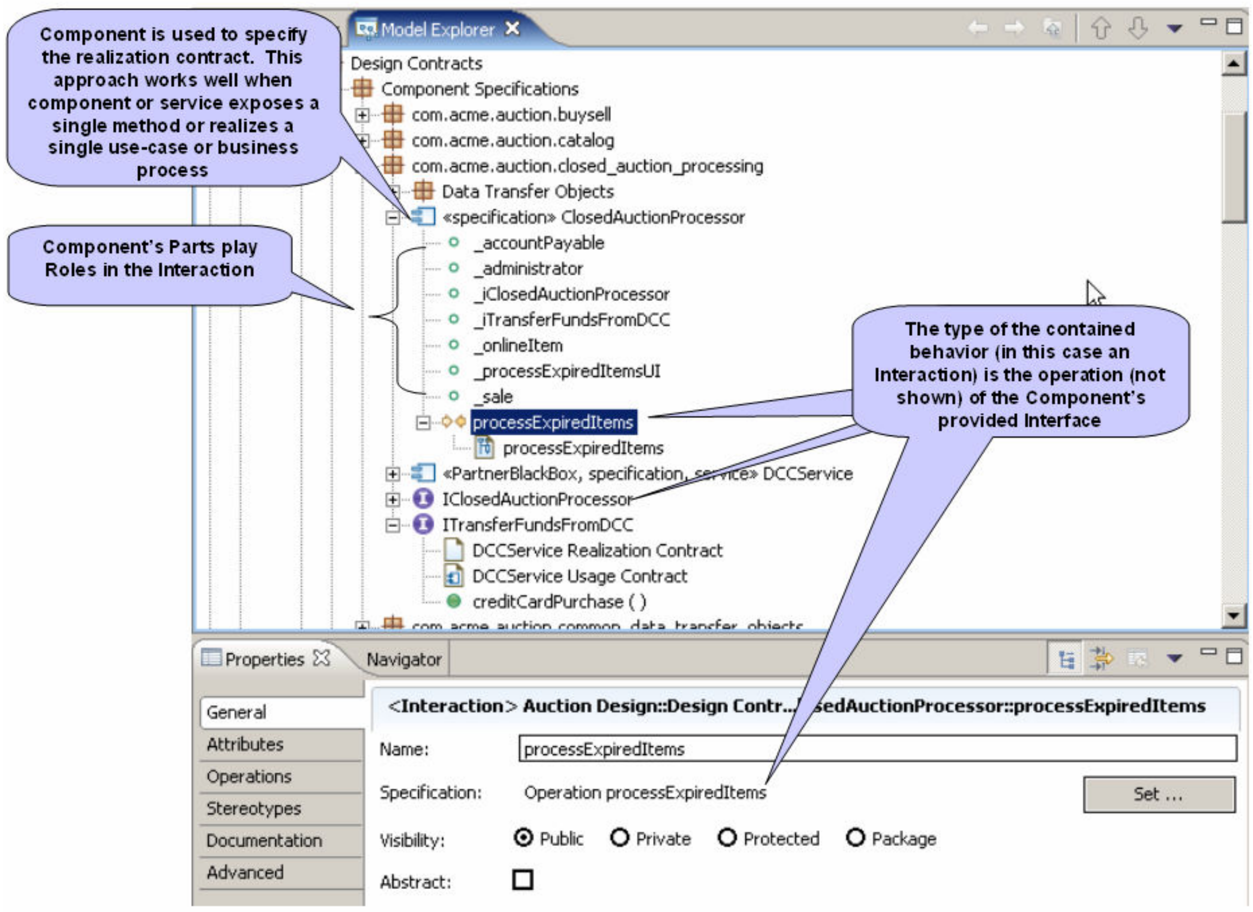
- The specification of data transfer objects (which would serve as the types of parameters of the provided operations, and might map to implementation constructs such as XML schema or SDOs) can also be included as part of the usage contract. For components that are not designed to be distributable, you may or may not choose to specify data transfer objects as the specifications of the types used as operation parameters. For distributable services (e.g. 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 RSx 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. So in RSx use Collaborations to represent use-case realizations.)

- [14] 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.


Component is used to specify the realization contract. This approach works well when component or service exposes a single method or realizes a single use-case or business process
Component's Parts play Roles in the Interaction
The type of the contained behavior (in this case an Interaction) is the operation (not shown) of the Component's provided Interface
Model Explorer
Properties
Navigator
Set ...

```
Design Contracts
  Component Specifications
    com.acme.auction.buysell
    com.acme.auction.catalog
    com.acme.auction.closed_auction_processing
      Data Transfer Objects
      «specification» ClosedAuctionProcessor
        _accountPayable
        _administrator
        _iClosedAuctionProcessor
        _iTransferFundsFromDCC
        _onlineItem
        _processExpiredItemsUI
        _sale
        processExpiredItems
          processExpiredItems
      «PartnerBlackBox, specification, service» DCCService
      IClosedAuctionProcessor
      ITransferFundsFromDCC
        DCCService Realization Contract
        DCCService Usage Contract
        creditCardPurchase ( )
    com.acme.auction.common.data_transfer_objects
```

General | Attributes | Operations | Stereotypes | Documentation | Advanced

<Interaction> Auction Design::Design Contr... sedAuctionProcessor::processExpiredItems

Name: processExpiredItems

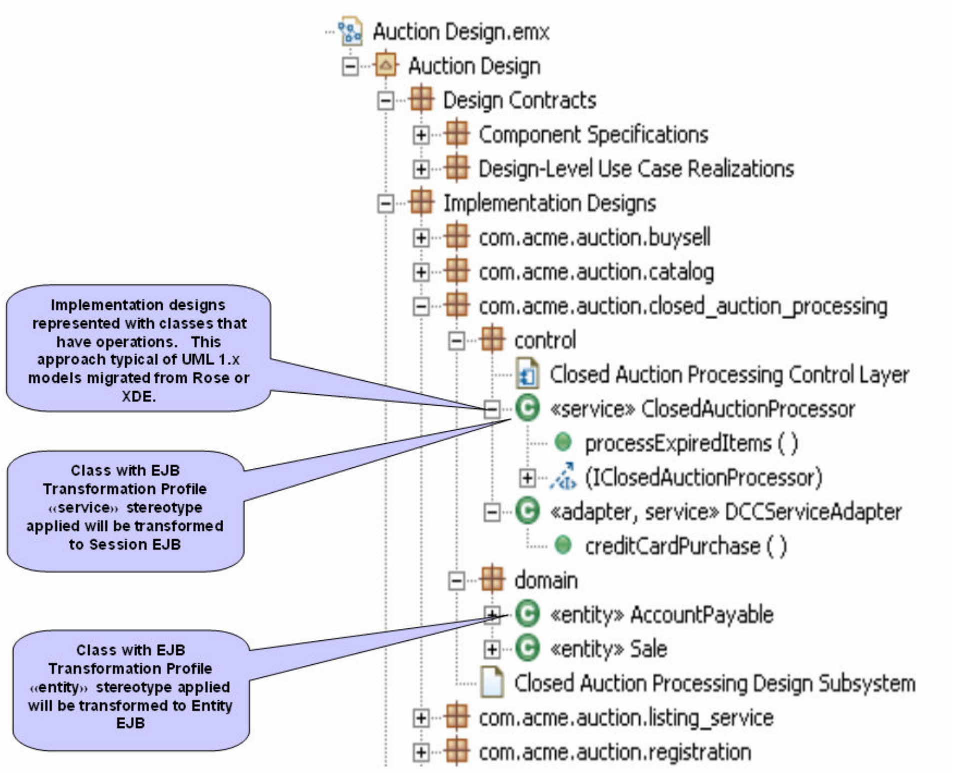
Specification: Operation processExpiredItems

Visibility: [✓ Public] Private Protected Package

Abstract: ☐

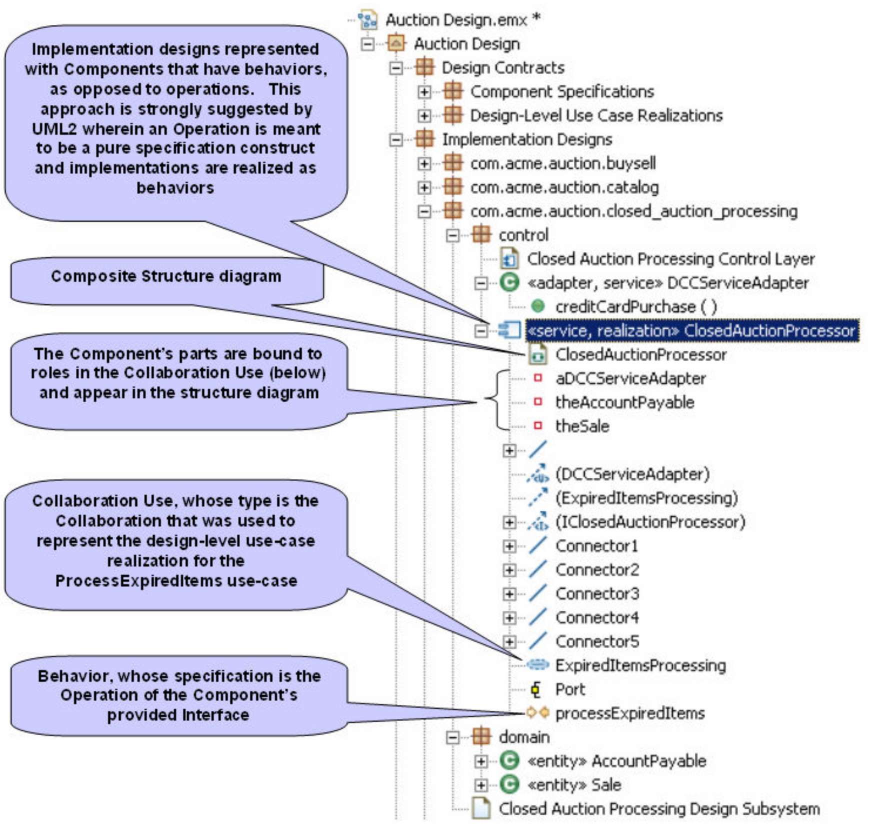
Figure 7-4

- 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 may 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).


Implementation designs represented with classes that have operations. This approach typical of UML 1.x models migrated from Rose or XDE.
Class with EJB Transformation Profile «service» stereotype applied will be transformed to Session EJB
Class with EJB Transformation Profile «entity» stereotype applied will be transformed to Entity EJB

```
Auction Design.emx
  Auction Design
    Design Contracts
      Component Specifications
      Design-Level Use Case Realizations
    Implementation Designs
      com.acme.auction.buysell
      com.acme.auction.catalog
      com.acme.auction.closed_auction_processing
        control
          Closed Auction Processing Control Layer
          «service» ClosedAuctionProcessor
            processExpiredItems ( )
            (IClosedAuctionProcessor)
          «adapter, service» DCCServiceAdapter
            creditCardPurchase ( )
        domain
          «entity» AccountPayable
          «entity» Sale
        Closed Auction Processing Design Subsystem
      com.acme.auction.listing_service
      com.acme.auction.registration
```

Figure 7-5


Implementation designs represented with Components that have behaviors, as opposed to operations. This approach is strongly suggested by UML2 wherein an Operation is meant to be a pure specification construct and implementations are realized as behaviors
Composite Structure diagram
The Component's parts are bound to roles in the Collaboration Use (below) and appear in the structure diagram
Collaboration Use, whose type is the Collaboration that was used to represent the design-level use-case realization for the ProcessExpiredItems use-case
Behavior, whose specification is the Operation of the Component's provided Interface

```
Auction Design.emx *
  Auction Design
    Design Contracts
      Component Specifications
      Design-Level Use Case Realizations
    Implementation Designs
      com.acme.auction.buysell
      com.acme.auction.catalog
      com.acme.auction.closed_auction_processing
        control
          Closed Auction Processing Control Layer
          «adapter, service» DCCServiceAdapter
            creditCardPurchase ( )
          «service, realization» ClosedAuctionProcessor
            ClosedAuctionProcessor
            aDCCServiceAdapter
            theAccountPayable
            theSale
            
            (DCCServiceAdapter)
            (ExpiredItemsProcessing)
            (IClosedAuctionProcessor)
            Connector1
            Connector2
            Connector3
            Connector4
            Connector5
            ExpiredItemsProcessing
            Port
            processExpiredItems
        domain
          «entity» AccountPayable
          «entity» Sale
        Closed Auction Processing Design Subsystem
```

Figure 7-6

8. Guidelines for Internal Organization of Implementation Overview Model

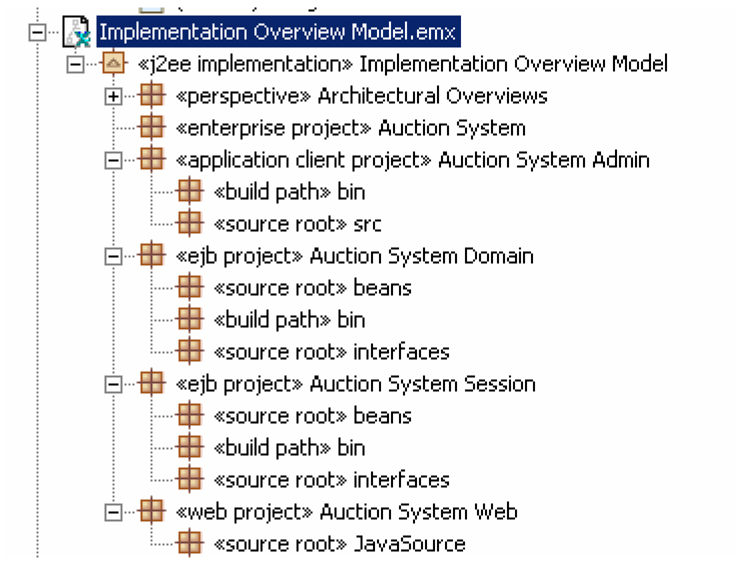
XDE/Rose
In the XDE Model Structure Guidelines, an Implementation Overview model was recommended as a device to provide a subsystems-level overview of the implementation. The details of each subsystem were then specified in the code model of the project that implemented the subsystem.

Strictly speaking, it should not be necessary to use an Implementation Overview model in RSx. 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.)

If you think about that in bottom-up terms, it says that you should think about how the implementation will be organized in terms of projects and folders and should factor that into the organization of the design model so that the transformation mappings end up being straightforward. Or if you look at it top-down it says that the organization of the design model, as it has evolved over the course of the design effort, should determine the set of implementation models (projects) that are needed. Either way, the end-game organization of the design model should naturally address the need for an overview of projects and subprojects, i.e. a diagram depicting the packages in the design model should be equivalent to an overview of projects and their sub-folders.

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. So there are some reasons you might want to use an Implementation Overview model. **Figure 8-1** Below 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** below shows an informal high-concept diagram of the auction system on which most of the samples in this paper are based.



```
Implementation Overview Model.emx
  «j2ee implementation» Implementation Overview Model
    «perspective» Architectural Overviews
    «enterprise project» Auction System
    «application client project» Auction System Admin
      «build path» bin
      «source root» src
    «ejb project» Auction System Domain
      «source root» beans
      «build path» bin
      «source root» interfaces
    «ejb project» Auction System Session
      «source root» beans
      «build path» bin
      «source root» interfaces
    «web project» Auction System Web
      «source root» JavaSource
```

Figure 8-1

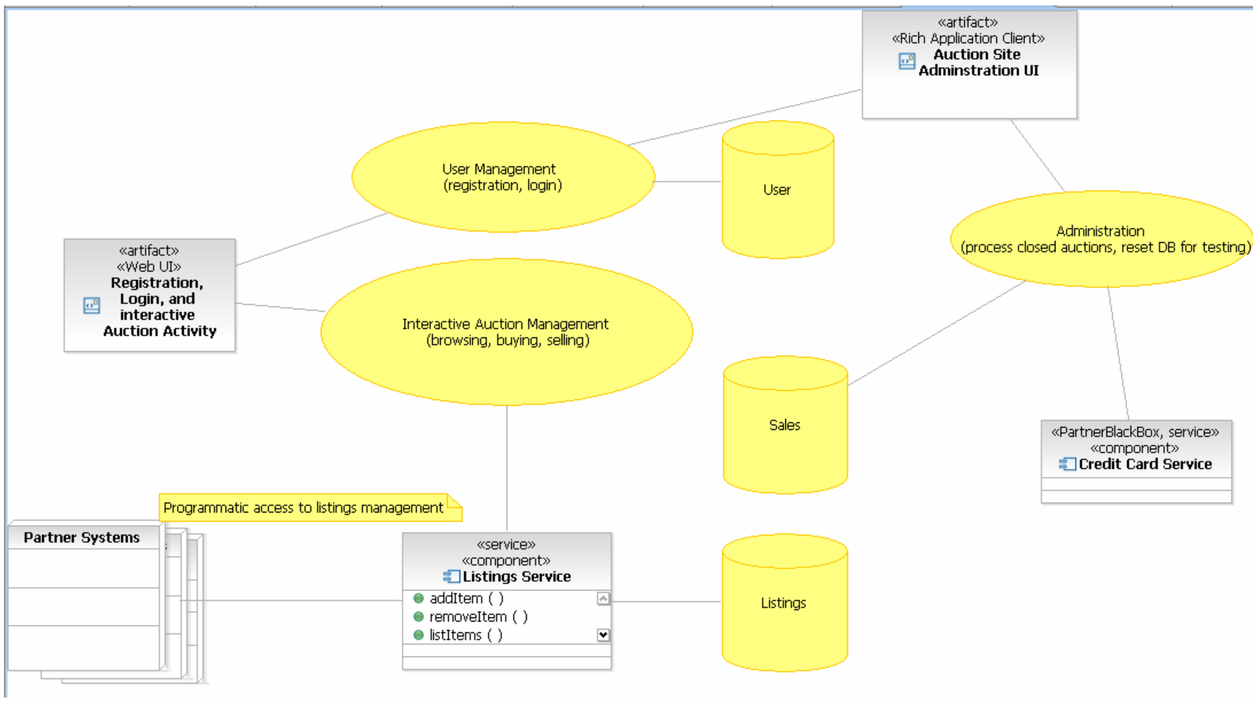
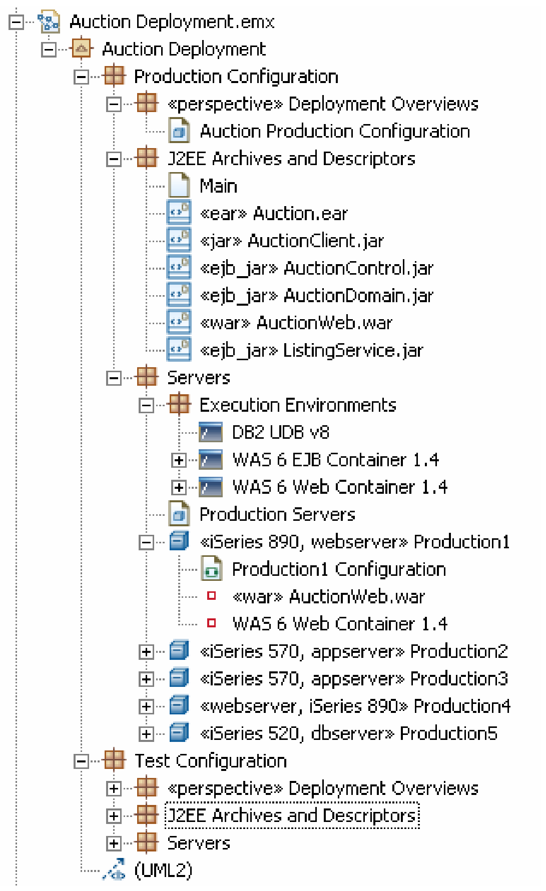

«artifact»
«Rich Application Client»
Auction Site Adminstration UI
User Management (registration, login)
User
Administration (process closed auctions, reset DB for testing)
«artifact»
«Web UI»
Registration, Login, and interactive Auction Activity
Interactive Auction Management (browsing, buying, selling)
Sales
«PartnerBlackBox, service»
«component»
Credit Card Service
Programmatic access to listings management
Partner Systems
«service»
«component»
Listings Service
addItem ( )
removeItem ( )
listItems ( )
Listings

Figure 8-2

9. Guidelines for Internal Organization of Deployment Model



```
Auction Deployment.emx
  Auction Deployment
    Production Configuration
      «perspective» Deployment Overviews
        Auction Production Configuration
      J2EE Archives and Descriptors
        Main
        «ear» Auction.ear
        «jar» AuctionClient.jar
        «ejb_jar» AuctionControl.jar
        «ejb_jar» AuctionDomain.jar
        «war» AuctionWeb.war
        «ejb_jar» ListingService.jar
      Servers
        Execution Environments
          DB2 UDB v8
          WAS 6 EJB Container 1.4
          WAS 6 Web Container 1.4
        Production Servers
        «iSeries 890, webserver» Production1
          Production1 Configuration
          «war» AuctionWeb.war
          WAS 6 Web Container 1.4
        «iSeries 570, appserver» Production2
        «iSeries 570, appserver» Production3
        «webserver, iSeries 890» Production4
        «iSeries 520, dbserver» Production5
    Test Configuration
      «perspective» Deployment Overviews
      J2EE Archives and Descriptors
      Servers
    (UML2)
```

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**. Just a couple of things of note in this example:

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

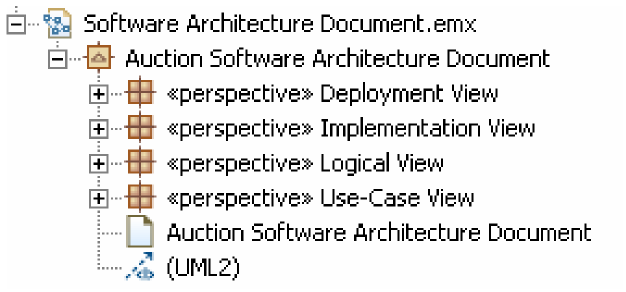
2. Overviews (e.g. of clusters, data centers, or enterprises) are maintained in «perspective» packages

3. A light-weight approach has been taken with regard to specializing and classifying nodes and artifacts: a combination of packaging and 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".

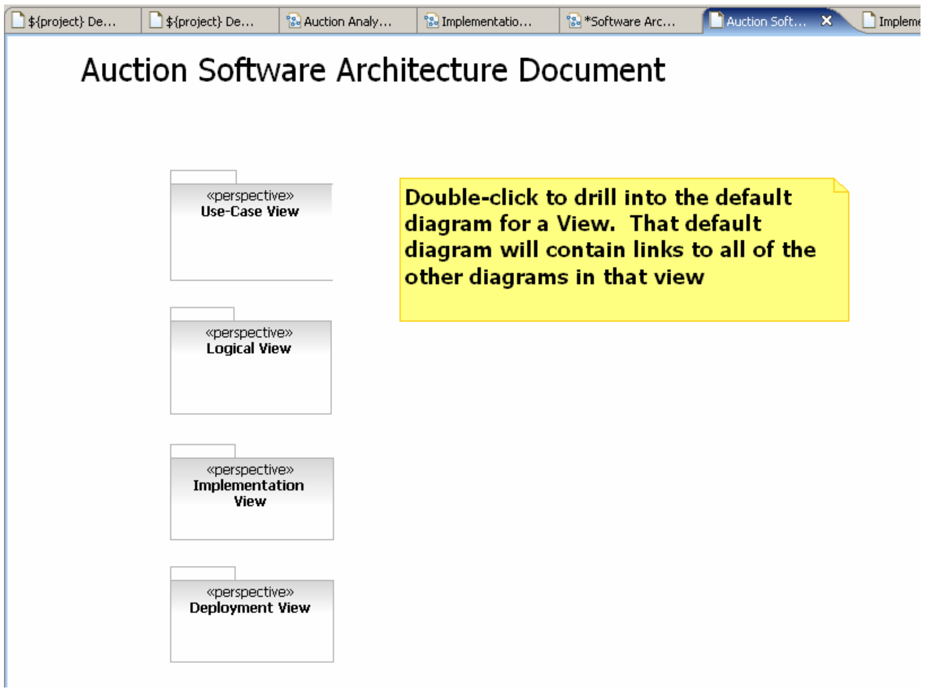
In simplest form, you might do something along the lines of **Figure 10-1**. Create a modeling file and populate it with a simple set of packages corresponding to the 4+1 Views. (The example is shown without a package for Process View, since the system in this example doesn't exhibit much in the way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



Software Architecture Document.emx
Auction Software Architecture Document
«perspective» Deployment View
«perspective» Implementation View
«perspective» Logical View
«perspective» Use-Case View
Auction Software Architecture Document
(UML2)

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



${project} De...
${project} De...
Auction Analy...
Implementatio...
*Software Arc...
Auction Soft...
Impleme

Auction Software Architecture Document

«perspective» Use-Case View
«perspective» Logical View
«perspective» Implementation View
«perspective» Deployment View

Double-click to drill into the default diagram for a View. That default diagram will contain links to all of the other diagrams in that view

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 and Model Management Considerations

This section will introduce some of the considerations for when and why you might choose to partition a model into multiple modeling files. A more comprehensive treatment of these issues is to be found in RSx online help. It is assumed here that the reader is somewhat familiar with the concepts of parallel development, and the notion of merging changes that have been made in parallel to multiple copies of an artifact.

First, a quick review: in the section "Basic Concepts and Terminology" we discussed the various model types such as Use Case, Analysis, and Design, as recognized by RUP. Examples were given to illustrate that in RSx…

- if you are building multiple applications you might have multiple models of each type (for instance multiple use case models, multiple analysis models, and so forth)
- a model (in the logical sense) can be persisted as one or more modeling files, for instance "the design model for application 'X'") can be persisted as a single modeling file or as a collection of multiple modeling files.

Partitioning Models

The techniques for partitioning models into multiple modeling files are covered in RSx online Help and are not addressed here. Here we are interested in when and why you would partition. There are two circumstances under which you might choose to maintain a particular model as multiple modeling files:

1. The model has grown to unmanageable size, or it's packaging structure has grown to unmanageable depth[15]
2. You begin to experience too many concurrent deliveries of changes to a modeling file, resulting in the need to perform merges with >2 change contributors[16]. (If you find this statement unclear, read on.)

Modeling in Teams

When your configuration management policy enables parallel development of models[17], uncoordinated changes will be made to the file (and in effect, to the logical model or model subset the file represents). At some point the changes must be merged. If it turns out that some of the changes conflict with one another, the merge is considered "**non-trivial**" because someone must make decisions about which of the conflicting changes should "win". (A 'trivial' merge is one in which the uncoordinated changes are not in conflict, and the model merge engine can perform the merges without human intervention.)
Non-trivial merges can be painful to undertake. How can they be minimized?

[15] Partitioning is required mainly when files grow too large for the machines that are in use in the user community. For example, a model that grows to 30MB on disk becomes very difficult to work with day-to-day on a 1GB RAM machine. On such a machine, one would want to partition the model with the goal of trying to keep 5 to 10MB worth of models in RAM at any time. An alternative is to upgrade the RAM (a relatively inexpensive solution, but one that works very well -- a 2GB RAM machine with no swap file performs much faster on almost every Eclipse operation, thus making very large models perform very well again.)

[16] The RSx model merge tooling supports merges with a maximum of 3 contributors: two 'change' contributors, and a 'base' contributor (aka the common ancestor). When there are more than 2 'change' contributors to deal with, the merges will begin to cascade and from that point one of the two 'change' contributors is the result set of the previously completed merges in the session.

[17] Examples of "parallel" development policies:

- a modeling file can be checked out for non-exclusive access
- a modeling file is worked on in parallel in the development streams of multiple practitioners)

You have two basic weapons for avoiding conflicts and the resultant non-trivial merges. One is “strong architecture”. The other is “strong ownership”. They go hand-in-hand: strong architecture *enables* strong ownership.

Weapon #1: Strong Architecture

“Strong architecture” in this context refers primarily to decomposition. The *principles* of architectural decomposition that apply here are the same ones that drive Object Oriented Development, Component Based Design, and Service Oriented Architectures:

- Strive for maximal decoupling of business functions
- Group together things that must remain tightly coupled. Isolate those groupings from one another
- If your resultant decomposition has a large number of 'grains', then depending on your staffing model (remember, strong architecture and strong ownership go hand-in-hand) you may want to start grouping those grains into high-affinity aggregates (which in modeling terms means UML Packages)
- There will *always* be some things that must be touched by many (and in some cases all) units of decomposition. Group those things together in a ‘common’ package and plan each development iteration so it includes a little “waterfall” at the start of the iteration that focuses on stabilizing the "common" stuff.
- There is also a time element. As you move from more abstract to more concrete understanding of a solution, your sense of the best organization for the architecture (and model) will evolve. You should plan for a model refactoring (reorganization) activity at the transition from each phase to the next (business analysis, requirements, application analysis, high-level design, ...).

If you look at your solution and everything appears to be highly interdependent and tightly coupled, either your architecture needs work or there is something about the nature of your problem domain that means you truly can't decompose the problem. In either case, you have a couple of choices:

- Resolve to assign the project to a very small team that shares a physical space and communicates very actively with one another regarding any changes they make that could affect other artifacts.
- Be prepared to perform lots of non-trivial merges

XDE/Rose

If you are a Rose or XDE user you have perhaps experienced doing non-trivial model merges. The good news is that non-trivial merges in RSx are far less painful. One primary difference is that RS*x* operates on a *mesh* of related modeling files as opposed to a *hierarchy* of petal files or sub-units. This critical difference means that we support high-level logical decomposition much better at the physical level in RS*x*.

We also support compare merge with better tooling – in fact, with a **vastly** better merge experience.

- The backend merge engine in RS*x* is 10,000 (yes, ten thousand) times faster than the XDE backend.
- It also implements a number of technologies such as “composite deltas” (a grouping mechanism that sorts changes by diagram and groups large diagramming gestures together to enable group and / or atomic operations), “model integrity protection”, “atomicity”, and “cascading delta processing”. These reduce the likelihood of corrupting a file by merge to about the same level as simply editing the file.
- There is now a true visual diagram merging GUI for resolving layout/cosmetic conflicts

Weapon #2: Strong Ownership

Once you have established strong architectural decomposition, it should prove fairly straightforward (specialized skill sets aside) to map “strong” ownership of architectural components to individual practitioners or small teams. When each logical package (or branch) in a model can be worked on exclusively by one practitioner, then merges performed with that model will be mostly trivial (regardless of whether the model is stored as a single modeling file or as multiple modeling files). The same may be said if each branch can be worked exclusively by a small team whose members are inclined to do a lot of informal communicating about what they are doing.

Can non-trivial merging be avoided by portioning models into multiple modeling files? In a word: “no”. **Architectural interdependencies are a logical phenomenon, not a physical one**. When you partition a model into multiple modeling files, the representations of the element interdependencies simply become cross-file references instead of in-file references. You don’t make it any easier to resolve conflicts (in fact you make it harder). And when you introduce cross-file references you introduce potential points of breakage (see sidebar).

Sidebar: Cross-Modeling File References

Whenever two modeling elements reside in different modeling files and you create a relationship between them, you create a “cross-modeling file reference”. Because modeling files (.emx files) are exposed in the host OS file system and can be moved, renamed, or otherwise modified outside of the Eclipse environment, these references represent potential points of breakage. However, so long as modeling files are always modified and change-managed through the Eclipse environment and you respect the following guidelines, you should not experience breakages.

Whenever you work with (edit) a ‘closure’ of modeling files (that is, a collection of modeling files that reference one another) you should have all of the modeling files of that closure present in the workspace. This does not strictly imply that all of the modeling files in a closure must reside in the same project, but using a single project generally does guarantee that all models will be present, since in typical CM workflows all model files travel together in the same project.

So let’s sum up:

- If you lack strong architecture, or if you have strong architecture but you lack strong ownership, you will experience frequent non-trivial merging that no amount of model partitioning can relieve.

- If you have strong architecture and strong ownership, you will greatly reduce (but will not eliminate) the frequency of non-trivial merging. You will not eliminate it because there will always be component interdependencies. The afore-mentioned ‘common’ elements are one example, although hardly the only example.

- Partitioning models into multiple files isn’t nearly as important as logically structuring models to enable multiple practitioners to work on a modeling file in parallel without introducing conflicting changes.

- The good news is RS*x* handles model merging far faster and more effectively than any other modeling tool available.

And when should you partition models? Try to avoid it, except in cases where the sheer size of a model has begun to test the limits of your hardware, and the modeling files you create can typically be worked on both *exclusively* (i.e. only one team member has a file checked out at any point in time) **and** *in isolation* (i.e. most changes can be made to the file without also requiring access to other files that contain related model elements).

Afterword: Changes Between Versions 6.x and 7.x of RSx

When RSx were first released (versions 6.x), they did not support the concept of a 'subunit' as supported by Rose and XDE (a subunit is a modeling file that contains a subset of a model and is 'transparent' in the sense that it does not appear in the logical view of the model that you see in the model explorer view). Instead, model partitioning was limited to defining multiple top-level models ("top-level" in the sense that each modeling file would appear as a separate, top-level entry in the model explorer view).

RSx also provide a capability that let you select a UML Package of an existing model, and "Make a Model" based on that Package. You can think of this as "shearing off" the Package to create a new modeling file. The result is that in the model explorer view, the Package appears to become a new top-level logical model. Visibility of the original hierarchical containment structure is lost, but whenever you open a modeling file from which Packages have been "sheared off" in this way, all of the "sheared off" models are also opened. This can result in unnecessary check-outs, as well as congestion in the model explorer view and appearance of large numbers of tabs in the editor pane that together, make navigation difficult.

The addition of subunit support in the 7.0 versions of RSx enables you to partition models into multiple files without loss of visibility to the hierarchical structure, and along with the elimination of the "model editor" tabs in the editor pane, overcomes the other navigation problems.

However, there was a lesson in model structuring to be learned from the earlier 6.x experience that still applies today. As noted above, opening a modeling file from which Packages had been sheared off would result in opening all of the sheared-off models. This could be avoided by planning a partitioning strategy in advance, instead of using the "Make a Model" feature.

Those following the "Make a Model" feature would typically start with a single modeling file, and create Packages to represent the organization of the solution architecture. For instance there might be a Package for each major component or subsystem of the solution (where 'component' and 'subsystem' reflect the informal usage of those terms that trace back to the early days of computing, not their formal UML semantic definitions). There might also be Packages for "common" or "utility" or "framework" components. Then, as such a model grew, the Packages that corresponded to application-specific components might be "sheared off" as separate modeling files, so that the teams building those components could (theoretically) work with those model files in isolation. But in practice, opening any one of those component-specific model files results in opening the original 'master' model (where the 'common' pieces reside), and that in turn opens all of the other application component-specific models. The results are the aforementioned unnecessary checkouts and difficult navigation.

A better approach is to plan in advance to define a separate model for each application-specific component, along with a model for the 'common' components (or perhaps, multiple models that define successive layers of common / reusable components, i.e. reflecting the abstraction layers of the implementation architecture). Then, any application-specific component model could be opened by the team that owns it, and only the 'common' models on which that application-specific model depends need also be opened.