Rational® XDE™ Model Structure Guidelines for J2EE™

Rational Software White Paper
TP 154, 05/03

Table of Contents

1. Introduction

This document provides recommendations on how to represent and structure the RUP® model artifacts in Rational XDE™, Java Platform Edition. Of course, whether you decide to model these RUP artifacts in XDE is a project-specific decision. Within this document, we note those models XDE provides automation support for and those that it does not, because this could influence your decision.

Since all XDE models exist within XDE projects, the XDE Project Structure section provides recommendations on what XDE projects should be created and what XDE models should be created in those projects.

Both RUP and XDE use the term "model" and the mapping between RUP models and XDE models is not always one to one. In the RUP Model to XDE Model Mapping section, the mapping from RUP models to XDE models is described.

The structure of each of the RUP model artifacts in their XDE model files is then described in its own section.

2. Scope

This document focuses on describing the recommended XDE model file structures, not on the process for developing the contents of the associated RUP artifacts. This document also does not describe detailed heuristics for defining the XDE projects that contain the described XDE models. For information on how to define, develop, and model the contents of the RUP artifacts, see RUP. For more information on projects, see the IDE documentation.

This document does not describe a complete example, but instead uses selected examples that emphasize the points being covered; however, all examples are consistent with each other and are taken from actual XDE models.

This version of the document does not address tag library development.

The project and model structures described in this document are just recommendations and could be replaced by any number of equally valid structures.

3. XDE Project Structure

The focus of this document is on how to structure XDE models. However, since all XDE models exist within XDE projects, it is important that we provide a brief introduction to the project structure in which our recommended model structures exist.

For a J2EE enterprise application that is being developed by multiple people, we recommend that you create the following XDE projects and models.
Note: If the XDE "create project" wizards are used, many of the models are created automatically when the project is created. In fact, if you are using WSS AD XDE, if you create an *Enterprise Application Modeling Project*, most of this multiple project structure is automatically created, including many of the models. XDE also provides model templates to jumpstart the content of the models.

XDE Project	Description	XDE Models "<recommended model name>" (<XDE file type: model template>]
Application Project (XDE Basic Modeling project)	The Application Project represents the entire application. Contains the XDE model files that describe the application as a whole	- "Use-Case Model" (Rational XDE: Use-Case Model) - "Analysis Model" (Rational XDE: Analysis Model) - "Overall Design Model" (Rational XDE: Design Model) - "Overall Implementation Model" (Rational XDE: Blank Model) - "EAR Deployment Model" (Java: EAR Deployment Model)
Data Modeling Project (XDE Data Modeling Project)	The Data Modeling Project contains the resources needed to model the application's data, as well as roundtrip engineer a Data Model to/from a database.	- "Logical Data Model" (Data: Logical Data Model) - "Physical Data Model" (Data: *vendor specific physical data model file*) [1] - "Domain Model" (Data: *vendor specific domain model file*)
EJB Projects (XDE EJB Modeling Project)	EJB projects contain the resources needed to implement EJB(s). The contained elements are packaged and deployed as an EJB module (.EJB JAR file). Separate EJB projects can be defined for individual EJBs or sets of EJB (each EJB project can contain at most one Java Code Model). The recommendation is to create an EJB project for each EJB-JAR that is to be produced. If separate projects are defined, then the name of the project should reflect its contents.[2]	- "EJB Code Model" (Java: EJB Code Model) - "EJB Deployment Model" (Java: EJB Deployment Model)
Web Projects (XDE Web Modeling Project)	Web projects represent the Web resources of the application. The contained elements are packaged and deployed in a Web archive file (WAR file). Separate Web projects can be defined for specific areas of the presentation logic. The recommendation is to create a Web project for each WAR that needs to be produced. If separate projects are defined, then the name of the project	- "Java Code Model" (Java: Java 1.3/1.4 Code Model) - "JSP Tag Library Model" (Web: JSP Tag Library Model)[4] - "Virtual Directory Model" (Web: Virtual Directory Model)[5] - "Web Deployment Model" (Web: Web Deployment Model)

1 Rational XDE provides physical database support for multiple database vendors. A vendor specific template exists for each database vendor supported by XDE.

2 If you are using XDE for WSAD, when you create additional EJB (Modeling) Projects, the wizard will ask for an Application Project to host the EAR. You should reuse the same Application (Modeling) Project above.

3 If you are using XDE for WSAD, when you create additional Web (Modeling) Projects, the wizard will ask for an Application Project to host the EAR. You should reuse the same Application (Modeling) Project above.

4 There can be multiple Tag Library models per project. In fact, you need a separate model for each .tld file. In any case, this version of the document does not address tag library development.

5 There can be multiple Virtual Directory models per XDE Web project.

	should reflect its contents.[3]	

An example of such a project and model organization is shown in Figure 1 (note the unique model names).

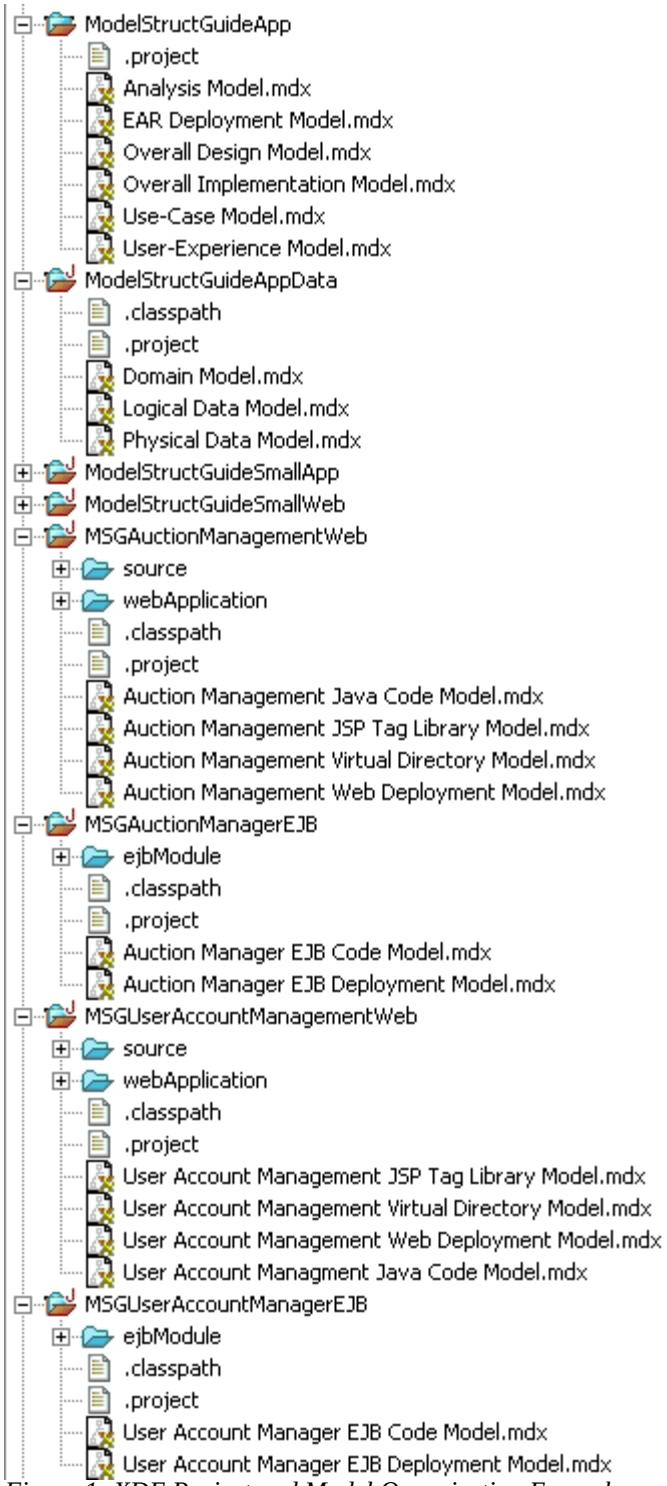

Figure 1: XDE Project and Model Organization Example

Alternatively, if the application is really small and is going to be developed by a single person, the above project

structure could be simplified to two projects, one that contains the application-wide and non-Web elements, and another that contains the Web elements. In addition to reducing the number of projects, the number of models can be reduced, as well. For example, for a small, single person project, the following simplifications are possible:

- A separate Analysis Model is not maintained. Analysis and design are both performed in the XDE roundtrip models.
- An "Overall Design Model" and an "Overall Implementation Model" are not maintained. The project is small enough that an overview can be obtained by looking at the XDE roundtrip models directly. Also, the Use-Case Realizations are maintained in the EJB code model and included references to elements in the Virtual Directory model.
- A separate Logical Data Model is not maintained. A physical data schema is developed directly in the "Physical Data Model".

Such a "small project structure" is summarized in the following table.

XDE Project	Description	XDE Models "<recommended model name>" (<XDE file type: model template>]
Application Project (XDE EJB Modeling Project)	The Application Project represents the non-Web aspects of the application. It contains the models that describe the application as a whole, the Data Model, and the EJB-specific models.	- "Use-Case Model" (Rational XDE: Use-Case Model) - "Physical Data Model" (Data: *vendor specific physical data model file*) - "EJB Code Model" (Java: EJB Code Model) - "EJB Deployment Model" (Java: EJB Deployment Model) - "EAR Deployment Model" (Java: EAR Deployment Model)
Web Project (XDE Web Modeling Project)	Web projects represent the Web resources of the application. The contained elements are packaged and deployed in a Web archive file (WAR file).	- "Java Code Model" (Java: Java 1.3/1.4 Code Model) - "JSP Tag Library Model" (Web: JSP Tag Library Model)[6] - "Virtual Directory Model" (Web: Virtual Directory Model)[7] - "Web Deployment Model" (Web: Web Deployment Model)

[6] There can be multiple Tag Library models per project. In fact, you need a separate model for each .tld file. In any case, this version of the document does not address tag library development.

[7] There can be multiple Virtual Directory models per XDE Web project.

An example of a small project and model organization is shown in Figure 2.

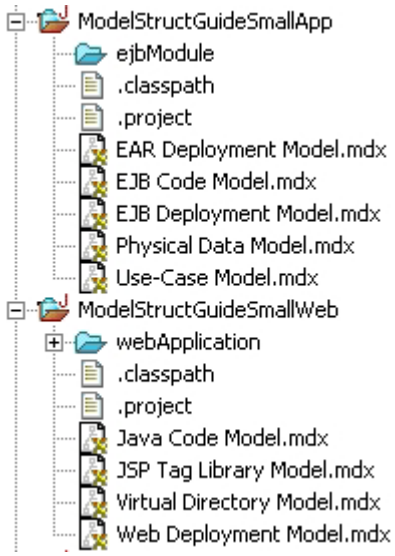

Figure 2: Small XDE Project and Model Organization Example

The actual selection of the number of projects and individual model files is an architectural choice and could vary for different projects. However, no matter how many projects are defined, there can only be one XDE Java Code model file per project. For more information on projects and the XDE model files they can contain, see the XDE documentation.

Also, it is strongly recommended that the *XDE model names be unique across all XDE projects*. This becomes extremely important when attempting to resolve references between XDE models. For more information on intermodel references and resolving them, see the XDE documentation.

For the examples in this document, the project and model structure shown in Figure 1 is used. Note that multiple EJB and Web projects have been defined. For the rationale for the multiple EJB and Web projects, see the Implementation Subsystems section.

4. RUP Model to XDE Model Mapping

Before describing how to represent the RUP model artifacts in XDE, it is important to address the confusion between a "RUP model" and an "XDE model" because they are different things and the mapping from the RUP models to the associated XDE models is not always one-to-one (close, but not one-to-one). Since "model" is used in both RUP and XDE, the initial assumption is that they should be the same. However, the models in RUP separate process concerns (analysis vs. design vs. implementation, etc.), where the models in XDE separate development concerns (separate code models for describing the programming language packaging structure versus a virtual directory structure, separate code models for different programming languages and development environments, etc.). In order to alleviate this confusion, in the context of this white paper, the term “model” is explicitly qualified with “RUP” or “XDE”.

The following table summarizes the mapping from RUP model to XDE model. The XDE models are those models introduced in the XDE Project Structure section. The structure of each of the XDE models is described in later sections of this white paper.

RUP Model	<XDE Project>: < XDE Model Name>
Use-Case Model	Application Project: Use-Case Model
Analysis Model	Application Project: Analysis Model
Design Model	Application Project: Overall Design Model [The Design Classes in each of the XDE roundtrip model files (see below)]
Data Model	XDE Data Models: - Data Modeling Project: Logical Data Model - Data Modeling Project: *Vendor-specific* Physical Data Model - Data Modeling Project: *Vendor-specific* Domain Model
Implementation Model	Application Project: Overall Implementation Model XDE roundtrip models[8] - EJB projects: EJB Code Models - Web projects: Java Code Models - Web projects: JSP Tag Library Models - Web projects: Virtual Directory Models
Deployment Model	XDE deployment models[9] - Application Project: EAR Deployment Model"[10] - EJB Projects: EJB Deployment Models - Web Projects: Web Deployment Models

[8] In the interest of brevity, I will use "XDE roundtrip models" throughout this white paper to represent these XDE models.

[9] In the interest of brevity, I will use "XDE deployment models" throughout this white paper to represent these XDE models.

[10] The EAR Deployment Model "crosses" the individual XDE deployment models. It contains diagrams that describe the deployment nodes and their connections. It also contains diagrams that map the individual archive files, defined in the individual deployment models, to the deployment nodes.

5. Use-Case Model

The recommended structure of the “Use-Case Model” is shown in Figure 3.

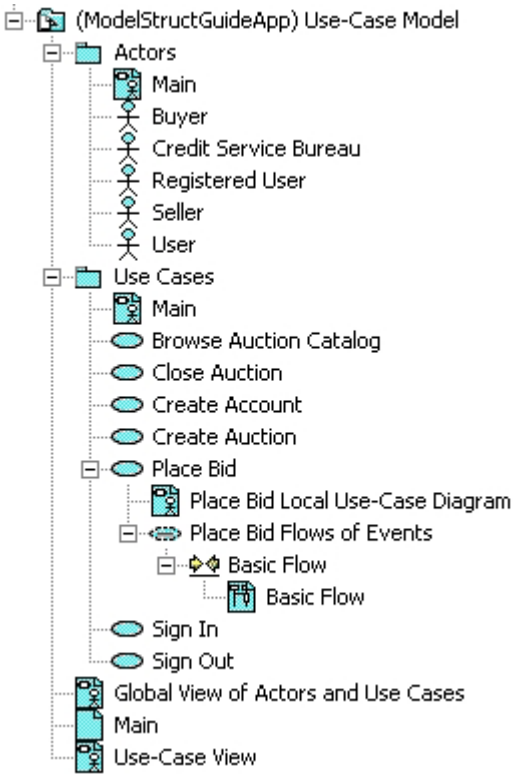

Figure 3: “Use-Case Model” Structure

The “Use-Case Model” is partitioned into two packages: “Actors” and “Use Cases”.

In addition to the **Use-Case Model** diagrams that contain the **Actors** and **Use Cases**, additional diagrams can be used to clarify different aspects of the **Use Cases**. The following supplemental model elements may be included “under” the **Use Case** model element in the **Use-Case Model**, as shown in Figure 3:

- The “Place Bid Local Use-Case Diagram” diagram contains the “Place Bid” **Use Case** and the **Actors** that participate in that **Use Case**.
- The “Place Bid Flows of Events” collaboration instance contains the interaction instances that describe graphically the flows of events described in the use-case description (i.e., the interactions between the **Actors** and the **Use Case**). The use-case collaboration instances should not be confused with **Use Case Realizations**, described in both the Analysis Model section and the Design Use-Case Realizations section, as the collaboration instances in the “Use-Case Model” are strictly “black box” and do not described interactions of elements within the application.
- The “Place Bid Flows of Events” activity graph contains the activity diagrams that describe graphically the flows of events described in the use-case description.

In the example shown in Figure 3, the “Global View of Actors and Use Cases” diagram in Figure 3 contains all of the **Use Cases** and **Actors** and their relationships, unlike the “Main” diagrams, which just contain the elements in the packages where the “Main” diagrams exist. If there are many **Actors** and **Use Cases**, the information on the “Global View of Actors and Use Cases” diagram can be expressed using multiple diagrams.

The “Use-Case View” diagram represents the Use-Case View of the software architecture. For more information on architectural views, see RUP.

If desired, additional packages can be created in the “Actors” and “Use Cases” packages to further organize the

contained model elements as shown in Figure 4.

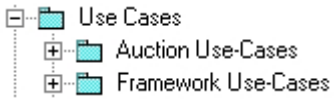

Figure 4: Additional Use-Cases Package Partitioning

6. Analysis Model

The Analysis Model is where the **Analysis Classes** and analysis **Use-Case Realizations** reside.

Note: Whether or not a separate **Analysis Model** and **Design Model** should be maintained is a project-specific decision. If a separate **Analysis Model** is created, but not maintained, then the **Analysis Classes** will be moved into the appropriate **Design Model** partition[11] and refined. Another option is to create the **Analysis Classes** and analysis **Use-Case Realizations** in the **Design Model** and then evolve them into their design form from there. See the Design Model section for more information on how the **Design Model** is represented in XDE.

The recommended structure for the **Analysis Model** is shown in Figure 5.

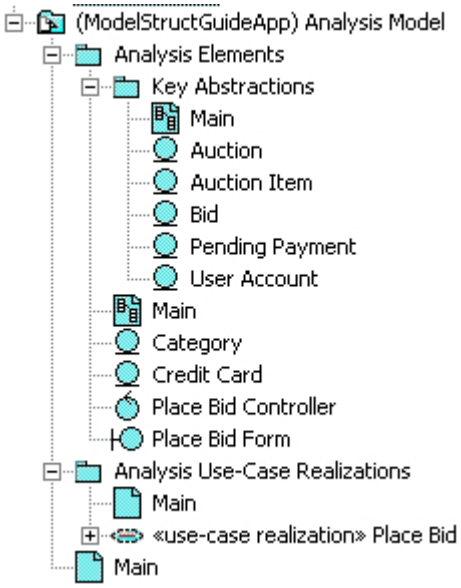

Figure 5: Analysis Model Structure

The "Analysis Elements" package contains the **Analysis Classes**. Instances of the **Analysis Classes** appear in the diagrams in the "Analysis Use-Case Realizations" package.

In addition to **Analysis Classes**, packages can be defined in the "Analysis Elements" package to further partition the contained **Analysis Classes** (see the "Key Abstractions package in Figure 5). Such additional partitioning is optional, especially if a separate **Analysis Model** is not to be maintained. In such cases, the **Analysis Classes** can be considered "transient" (i.e., they only exist until they evolve into design elements), so their organization is not considered critical. One possible exception is the key abstraction **Analysis Classes**.

As shown in Figure 5, the "Key Abstractions" package contains the **Analysis Classes** that are considered to represent the key abstractions of the system. As noted earlier, this package is optional. An alternative is to represent the key abstractions on a class diagram in the "Analysis Elements" package. However, creating a separate package provides a more explicit categorization of **Analysis Classes** as key abstractions. In fact, even if a separate **Analysis Model** is not maintained in its entirety, some projects may choose to maintain the key abstraction **Analysis Classes**.

11 As you will see later, the right "Design Model partition" just might be a package in one of the XDE roundtrip models since design of technology-specific elements is performed in the roundtrip models.

In such cases, defining a separate package to contain the **Analysis Classes** that are maintained is helpful. Note: The key abstractions also appear on the "Logical View: Key Abstractions" diagram in the "Overall Design Model". See the Design Model section for more information.

The "Analysis Use-Case Realizations" package contains the analysis-level **Use-Case Realizations**, which describe how the **Use Cases** are performed in terms of the **Analysis Classes** in the "Analysis Elements" package. Each of the analysis **Use-Case Realizations** realizes a **Use Case** in the **Use-Case Model**, has the same name as that **Use Case**, and should have the structure as that shown in Figure 6.

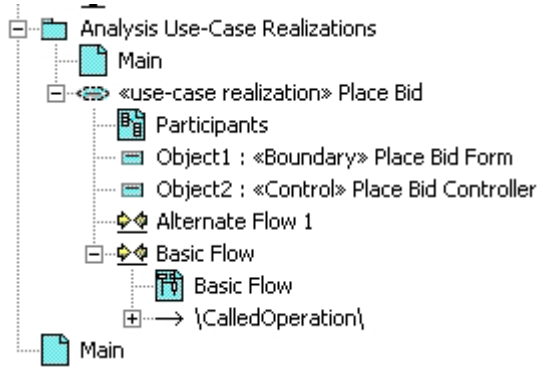

Figure 6: "Analysis Use-Case Realization" Package Structure

The "Participants" diagram shows the **Analysis Classes** (from the "Analysis Elements" package) participating in the **Use-Case Realization** (that is, those **Analysis Classes** whose instances appear on the interaction diagrams) and the relationships that support the collaboration described in the interaction diagrams.

The "flow" interaction instances ("Basic Flow" and "Alternate Flow 1") contain sequence diagrams that describe the **Use Case** flows of events. There should be one interaction instance for each significant use-case flow of events. The sequence diagrams in the interaction instances describe the flow between the participating **Analysis Classes** during the execution of the associated **Use Case**.

7. Design Model

The RUP **Design Model** is represented by multiple XDE models - the "Overall Design Model" and the roundtripped design elements that reside in separate XDE roundtrip models (roundtripped design elements are detailed design elements that participate in roundtrip engineering). That way, the automation available in the individual roundtrip models can be leveraged. For example, the XDE EJB patterns can be used to create the classes that specify the EJB.

The "Overall Design Model" describes the design of the application as a whole and contains elements that span multiple XDE roundtrip models. It contains the logical partitions that inspire the organization of the individual roundtrip models, as well as the **Use-Case Realizations** that tie everything together (the **Use-Case Realizations** describe the collaboration amongst the design elements from the different roundtrip models). The "Overall Design Model" contains diagrams that refer to the roundtripped design elements. For information on the individual XDE roundtrip models, see the Implementation Model section.

Another possibility is to represent the **Design Model** and the **Implementation Model** in the same XDE code model. This is only possible if you have only one target implementation language and your team is small. For an example of a small project structure, see the XDE Project Structure section.

Maintaining the "Overall Design Model" is optional, but may be a good idea for organizing diagrams, raising the level of abstraction, etc., as well as providing a place for design elements while still figuring out what implementation mechanism to apply.

The recommended structure of the "Overall Design Model" is shown in Figure 7.

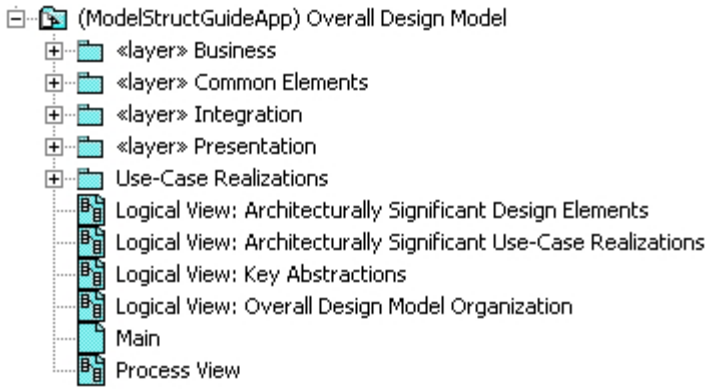

Figure 7: Overall Design Model Structure

The Overall Design Model contains the following packages:

- The «layer» packages contain (or contain diagrams that reference) the design elements of the system (**Design Classes**, **Interfaces**, and **Design Subsystems**). This structure represents a particular partitioning strategy that we describe in the Design Layers section.
- The "Use-Case Realizations" package contains the design-level Use-Case Realizations. The internal structure of the Use-Case Realizations is discussed in more detail in the Design Use-Case Realizations section.

The diagrams representing architectural views include "View" in the diagram name. For more information on architectural views, see RUP.

The "Logical View: Key Abstractions" diagram contains the key abstractions of the system. There are several options for maintaining these key abstractions:

- A complete **Analysis Model** is maintained. In that case, the "Logical View: Key Abstractions" diagram contains the **Analysis Classes** from the **Analysis Model** that represent the key abstractions of the system.
- A partial **Analysis Model** is maintained, namely, just the key abstractions. In that case, the "Logical View: Key Abstractions" diagram contains the **Analysis Classes** from the **Analysis Model** that represent the key abstractions of the system.
- No part of the **Analysis Model** is maintained. In that case, the **Analysis Classes** that represent the key abstractions can be maintained in a package in the **Design Model**, called "Key Abstractions"

For more information on the **Analysis Model**, see the Analysis Model section.

7.1 Design Layers

The «layer» packages contain the design elements of the system (e.g., **Design Classes**, **Interfaces**, and **Design Subsystems**) that evolve from the **Analysis Classes**. The «layer» packages could contain any number of subpackages that further partition the contained design elements. The design **Use-Case Realizations** (contained in the "Use-Case Realizations" package of the "Design Model" are discussed under the heading in the Design Use-Case Realizations section) are written in terms of the design elements contained in these packages.

The **Design Model** can follow any number of partitioning strategies. The partitioning strategy described in this section is shown in Figure 8.

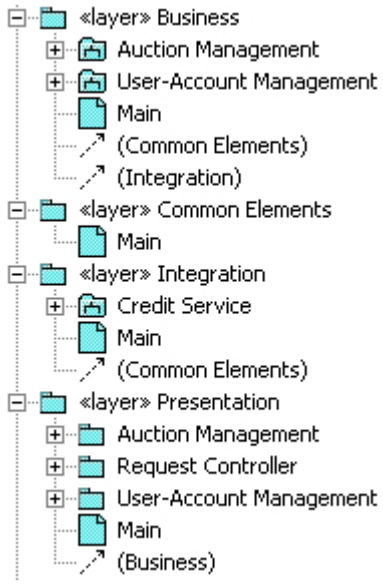

Figure 8: Design Package Partitioning Example

In this example, the first level packages are considered layers, where each layer has a specific responsibility. The second level packages further partition the layer package elements by business functionality.

The “Presentation” layer package is responsible for handling interactions with the end user. In a J2EE application, the design elements that might reside in the “Presentation” layer package include HTML pages, Java Server Pages (JSPs), and servlets. You can further divide the “Presentation” layer package into sub-packages to group elements that belong to a related set of **Use Cases**; for example, the “Auction Management” package in Figure 8.

The “Business” layer package is responsible for performing any business processing. In the “Design Model” structure presented in this document, the “Business” layer package is comprised of a set of design subsystem packages, one per major business function (for example, the “Auction Management” and “User Account Management”, subsystem packages in Figure 8). **Design Subsystem** packages are described in more detail under the heading in the Design Subsystems section.

The “Integration” layer package is responsible for providing access to back-end resources, including databases and external systems. In the **Design Model** structure presented in this document, the “Integration” layer package is also comprised of design subsystem packages, one per external system (for example, the “Credit Service” subsystem package in Figure 8). **Design Subsystem** packages are described in more detail under the heading in the Design Subsystems section.

The “Common Elements” layer package contains the elements that are shared across layers.

Again, the structure described in this section could be replaced with a different structure that reflects a different partitioning strategy.

7.2 Design Subsystems

Design Subsystems are represented by subsystem packages in the “Overall Design Model”. Each design subsystem package should have the same structure. The specifics of that structure vary depending on the level of detail being captured for the **Design Subsystem**.

An example of a more formal and rigorous **Design Subsystem** structure is shown in Figure 9.

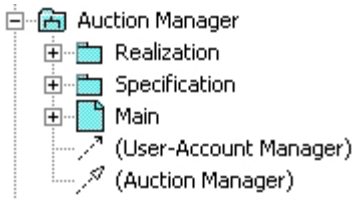

Figure 9: Design Subsystem Structure

This design subsystem package structure supports the definition of separate "Specification" and "Realization" packages within the design subsystem package. This structure was influenced by the book titled *UML Components: A Simple Process for Specifying Component-Based Software* written by J. Cheesman and J. Daniels. A simplified design subsystem package structure that does not contain these partitions could be used without impacting the other model file structures defined in this document. Each of the "Specification" and "Realization" packages is discussed in the following sections.

7.2.1 Subsystem Specification

The "Specification" package contains a description of the **Design Subsystem's** interfaces.[12] An example of a subsystem specification is shown in Figure 10.

Figure 10: Design Subsystem Specification Example

7.2.2 Subsystem Realization

The "Realization" package contains a description of how the **Design Subsystem** specification is realized. An example of the "Realization" package of a design subsystem package is shown in Figure 11.

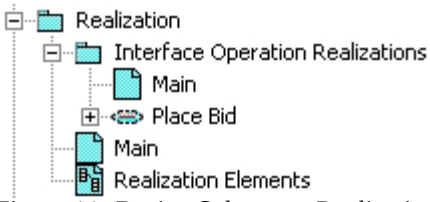

Figure 11: Design Subsystem Realization Example

The "Realization Elements" diagram contains references to the design elements that realize the subsystem. The design elements themselves may reside in the "Realization" package or they may reside in a separate XDE code model, where they participate in roundtrip engineering. For more information, see the XDE Roundtrip Models section.

[12] In this simple example, you might question the need for a separate package just for the interface. However, on a real project the package is worth maintaining because it can contain references to documents that describe the subsystem and, in particular, interface constraints such as preconditions and post conditions on the operations.

The “Interface Operation Realizations” package contains collaboration instances that describe how the subsystem elements realize the significant operations of **Design Subsystem** interfaces (in the “Specification” package). There is one collaboration instance per significant subsystem interface operation.[13] An example of an “Interface Operation Realizations” package is shown in Figure 12.

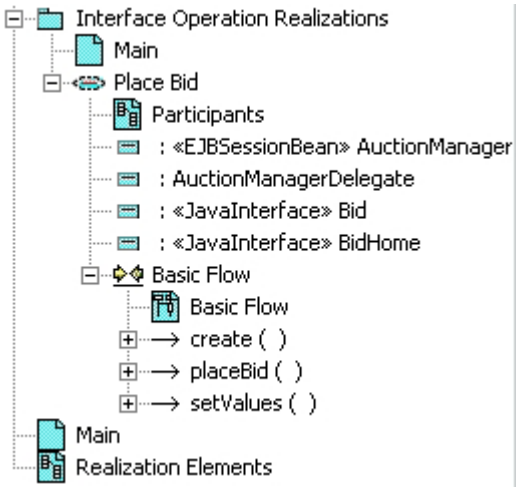

Figure 12: Interface Operation Realizations Package Example

As with the analysis-level **Use-Case Realizations** (discussed earlier in the Analysis Model section) and the design-level **Use-Case Realizations** (discussed later in the Design Use-Case Realizations section), each interface operation realization contains a class diagram containing the subsystem elements that participate in the realization (the “Participants” diagram in Figure 12), as well as interaction diagrams that describe how those participants collaborate to perform the subsystem interface operation (the “Basic Flow” diagram in Figure 12).

7.3 Design Use-Case Realizations

The “Use-Case Realizations” package contains the design-level **Use-Case Realizations**. Each of the **Use-Case Realizations** is associated with a **Use Case** in the **Use-Case Model**, has the same name as that **Use Case**, and should have the structure shown in Figure 16.

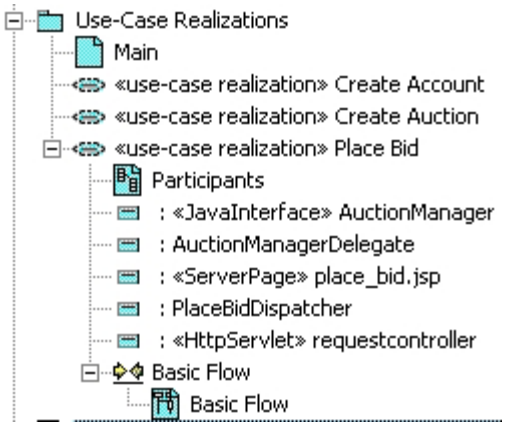

Figure 13: Design Use-Case Realization Structure

The **Use-Case Realization** “Participants” diagram shows the design elements that participate in the **Use-Case Realization** (that is, those design elements whose instances appear on the **Use-Case Realization** interaction diagrams) and the relationships that support the collaborations described in the interaction diagrams.

[13] Not all operations need to be defined at this level. Some simpler operations might not need a separate collaboration instance.

The “Basic Flow” diagram is an example of an interaction diagram that describes the flow between the participating design elements during the execution of the associated **Use Case**. There should be an interaction instance for each flow of events in the **Use Case**.

It is important to note that the **Use-Case Realization** diagrams may (and usually do) contain references to design elements that physically reside in separate XDE roundtrip models. The **Use-Case Realization** is where the collaboration amongst elements in separate roundtrip models is demonstrated.

8. Data Model

The RUP **Data Model** is represented by multiple XDE model files:

- **Logical Data Model** (optional). Represents the Logical Data Model, which is an application independent view of the logical design of the database.
- **Physical Data Model**. Represents a database vendor-specific Physical Data Model. It contains the detailed model elements for defining the specific characteristics of the tables of the database. The “Physical Data Model” XDE model file also includes the database specific implementation artifacts for implementing the tables in a vendor-specific database.
- **Domain Model** (optional). Represents the database vendor-specific data types that may be used to define consistent data types across the “Physical Data Model”.

The separation of the XDE model files provides the optimal flexibility for supported automation between the **Design Model**, the **Data Model**, and the physical database.

Each of these XDE model files is described in more detail below.

8.1 Logical Data Model (Optional)

The Logical Data Model may be used in situations where the project has a need to create a standalone logical data representation of the key entities and relationships important to the design of the database. Creating a XDE Logical Data Model is optional since the database design team may instead transform persistent **Design Classes** in the **Design Model** to tables in the **Data Model** to create the initial physical database design structure directly in the XDE Physical Data Model (see the Physical Data Model section below).

The XDE Logical Data Model may be partitioned into subject area packages, as needed. Subject area packages define logical groupings of entity classes. The XDE Logical Data Model may also contain a “Common Elements” package that contains model elements that cross subject areas.

The diagrams with “View” in the name are used to document the Data View of the architecture. The “Data View: Overall Logical Data Model Organization” diagram is used to document the high-level data organization of the Logical Data Model, as expressed in the major partitions (i.e., packages) of the XDE Logical Data Model. The “Data View: Key Logical Data Elements” is used to document the key logical elements of the **Data Model**. For more information on architecture views, see RUP.

An example of the recommended structure for the Logical Data Model is shown in Figure 14.

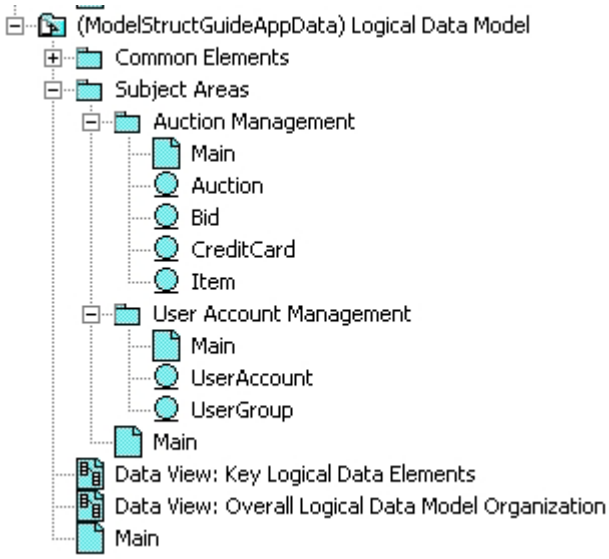

Figure 14: Logical Data Model Structure

In this example, there are two subject area packages, “Auction Management” and “User Account Management”. Each subject area package contains the entity classes that together comprise the Logical Data Model. There is not a direct mapping to package structures in the **Design Model** though there may be some similarity.

8.2 Physical Data Model

The Physical Data Model contains the detailed database table and stored procedure designs that are used to implement the database through the XDE Data Modeler forward engineering facilities. The Physical Data Model also consists of the model elements used to define the physical storage configuration of the database. In general, the model elements include the databases and tablespaces that comprise the physical layout of the database tables on the target storage media.

When creating the Physical Data Model, the Database Designer must select the appropriate target database. Supported databases include: DB2 MVS, DB2 UDB, Oracle, Sybase, and SQL Server. XDE will default the XDE model file name to the selected database. In the “Physical Data Model” example in this document, the XDE model file name has been updated to “Physical Data Model”. A Database Designer may choose to accept the default name when creating the “Physical Data Model”.

An example of the recommended structure for the Physical Data Model is shown in Figure 15.

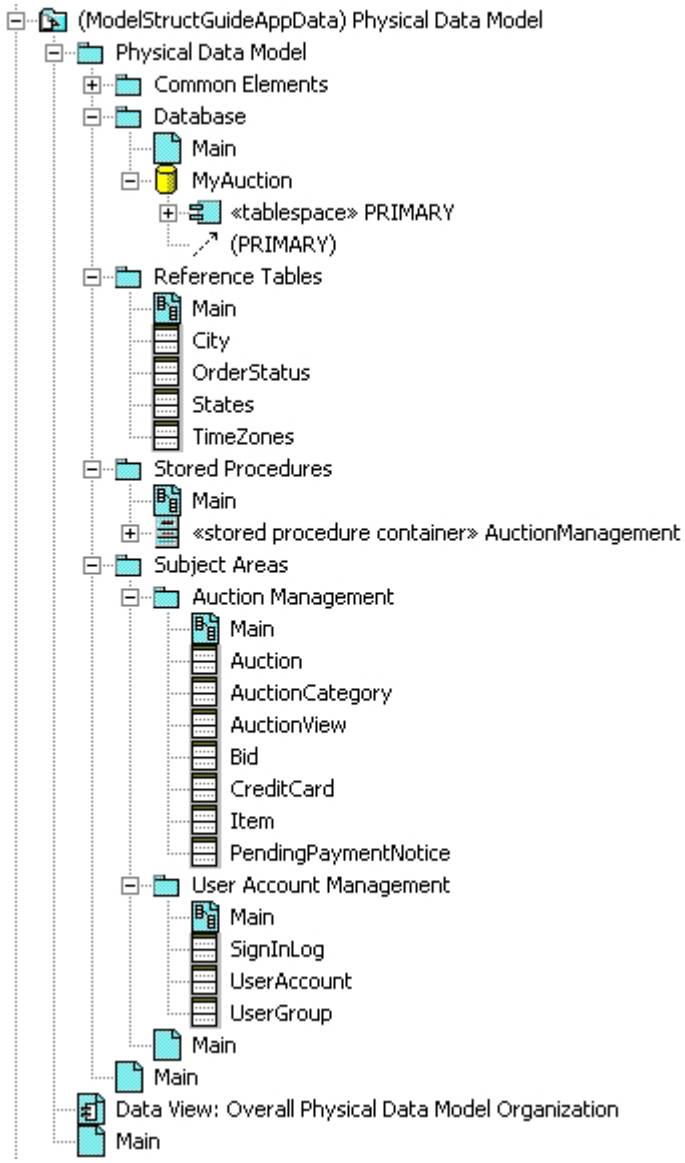

Figure 15: "Physical Data Model" Structure

The "Common Elements" package contains the database tables and views that cross the subject areas.

The "Database" package contains the model elements that define the physical storage configuration of the database. It contains the databases and tablespaces that comprise the physical layout of the database tables on the target storage media. Tablespaces are used to logically group tables within a database. For guidelines on defining tablespaces, see RUP. The "Database" package may be partitioned into lower level packages as needed, depending on the complexity of the application.

In the example shown in Figure 15, the "Database" package contains a single database, MyAuction, its associated tablespace, PRIMARY, and the table realization relationships. The tablespace can be named any appropriate name for a database project. For the MyAuction database, only one tablespace named PRIMARY is defined. When forward engineering is performed, the tables linked to the database via the realization relationship with the database's tablespace are created (either in a database or in a DDL).

The "Reference Tables" package contains the static data tables that hold "constant" data information needed by the application.

The “Stored Procedures” package contains all the classes that represent the database-stored procedures («stored procedure container » classes and the associated «stored procedure» operations). Stored procedures that relate to a single table can be packaged either in the “Stored Procedures” package or in the “Subject Area” package with the table the stored procedure references, depending on whether you want to represent a “stored procedure centric” or a “table centric” view[14].

The “Subject Areas” package contains packages that group logically related sets of tables and views[15]. It is recommended that views be created in the subject area package along with the tables. This recommendation is purely for organizational reasons. It can be helpful to keep views in the subject area where they are used, which places them in the same subject areas as the tables. In this example shown in Figure 15, there are two subject area packages, “Auction Management” and “User Account Management”. The number of subject area packages is dependent on the complexity of the application. However, in general, the subject area packages in the Logical Data Model “inspire” the subject area packages in the Physical Data Model. The subject areas in the Logical Data Model are abstractions of the subject areas in the Physical Data Model.

The tables in the subject area packages contain the columns and triggers defined for the table. The tables are created through one of the following

- XDE class to table transformation function.
- XDE reverse engineering an existing database function[16].
- Manual creation by the Database Designer.

When reverse engineering an existing database, a schema package(s) are created in the XDE Physical Data Model. The names of these packages are based on the database owner[17] of the database being reversed engineered. It is recommended that the reverse engineered tables be moved into subject area packages within the “Subject Areas” package and that the reverse engineered schema packages be deleted. Moving the tables into subject area packages functionally organizes the tables to allow the Database Designer to update the tables as necessary.

The diagrams with “View” in the name are used to document the Data View of the architecture. The “Data View: Overall Physical Data Model Organization” diagram is used to document the high-level data organization of the Physical Data Model, as expressed in the major partitions (i.e., packages) of the XDE Physical Data Model. For more information on architecture views, see RUP.

8.3 Domain Model (Optional)

The Domain Model is an optional XDE model that is used to store the user-defined datatypes for the database. Domains enable Database Designers to reuse element properties across the database design. A domain is used by the Database Designer to consistently document the properties of a column through out the database. The name of the column is defined in the table; the domain is used to define the *TypeExpression* of the column.

---

14 A table-centric view allows for better understanding of the database design / operation all in one view. A stored procedure centric view simplifies finding and changing/maintenance of the stored procedure.

15 Some may question the use of subject area packages in the physical data model, as it requires additional maintenance to maintain the logical and physical database subject area packages. The subject areas in the physical data model are here for consistency with the logical data model (if it is used) and more so for the case where the physical data model is “large” and there is no logical data model. In such a case the subject area packages can be used to manage the tables generated from the Class to Table transformation.

16 Typically the database is reverse engineered once, and then all future updates are synchronized using XDE’s Compare and Sync functions.

17 Within XDE, the database owner is captured as a property of the <<database>> component. Inside the Location property, as part of the connection string, there is a schema attribute. When reverse engineering a database, this is typically the database owner.

An example of the recommended structure for the Domain Model[18] is shown in Figure 14.

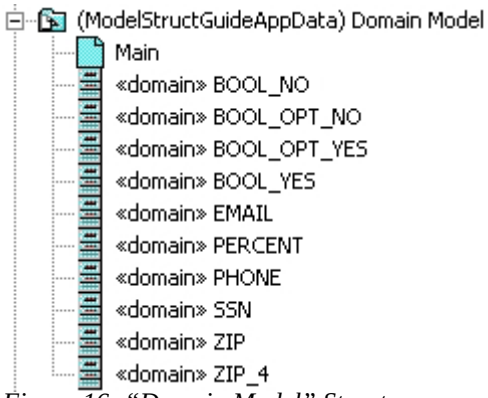

Figure 16: "Domain Model" Structure

In this example demonstrates the SQL Server domain values organized within the "SQL Server Domain" package. In cases where the Database Designer defines a large number of domains, it may be necessary for the Database Designer to organize the domains using packages under the "SQL Server Domain" package.

9. Implementation Model

The RUP **Implementation Model** is represented by multiple XDE models - the "Overall Implementation Model" and multiple XDE roundtrip model(s)[19]. Using multiple roundtrip models allows different implementation concerns to be represented most efficiently, such as describing a Java packaging structure versus a virtual directory structure. Also, the automation available in the individual roundtrip models can be leveraged (i.e., XDE provides automation for the creation of implementation-technology-specific model elements, and the synchronization of these model elements with source code using roundtrip engineering).

The "Overall Implementation Model" describes the implementation of the application as a whole and contains elements that span multiple roundtrip models. It contains diagrams that refer to elements in the roundtripped design elements, and usually does not "own" any model elements itself. The "Overall Implementation Model" does not participate in any XDE roundtrip engineering.

Maintaining the "Overall Implementation Model" is optional; however, it can be useful for describing the overall structure of the implementation, including the units of integration (defined in RUP as **Implementation Subsystems**), as well as for documenting the Implementation View of the architecture. **Implementation Subsystems** are discussed in more detail in the Implementation Subsystems section

The number of XDE roundtrip models depends on the defined XDE project and model organization (for more information, see the XDE Project Structure section). However, one option is to define separate projects (and roundtrip models) for each **Implementation Subsystem**. For information on how to represent **Implementation Subsystem** in XDE, see the Implementation Subsystems section. Alternatively, as mentioned earlier, if you have a single target implementation language and your team is small, you may choose to use a single XDE roundtrip model to represent both the RUP **Design Model** and **Implementation Model**.

---

18 Within XDE, several vendor databases are supported, including DB2, Oracle, Sybase, and SQL Server. When creating a Domain XDE Data Model, the Database Designer will create the Domain XDE Data Model by selecting the appropriate vendor database. XDE will create a default list of domains for the selected database vendor.

19 XDE roundtrip models are hybrid models. They are used to represent both RUP **Design Models** and RUP **Implementation Models**. The model elements in the XDE roundtrip models represent RUP **Design Classes** (classes that map directly to physical classes are considered **Design Classes**) and physical implementation files. The structures of the XDE roundtrip models represent the physical directory structures.

An example of an “Overall Implementation Model” is shown in Figure 17.

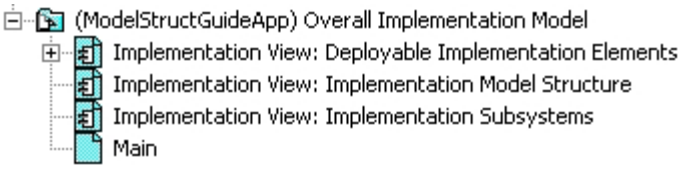

Figure 17: Overall Implementation Model Structure

As shown in Figure 17, the “Overall Implementation Model” just contains diagrams. The diagrams representing architectural views include “View” in the diagram name. For detailed information on architectural views, see RUP.

The “Implementation View: Deployable Implementation Elements” diagram references the archive files that will be deployed to nodes in the XDE deployment models. The archive files themselves are physically contained in the deployment models. For more information, see the Deployment Model section.

The “Implementation View: Implementation Subsystems” diagram references the application’s Implementation Subsystems. Dependency relationships may be drawn between the **Implementation Subsystems** in the diagram. These dependencies represent the **Implementation Subsystem** imports, which determine the order in which the **Implementation Subsystems** should be integrated. For information on how to represent Implementation Subsystems in XDE, see the Implementation Subsystems section.

The “Implementation View: Implementation Model Structure” diagram contains references to all XDE models used to represent the **Implementation Model**, and their relationships.
Note: When individual projects are defined for each of the **Implementation Subsystems**, then the content of the “Implementation View: Implementation Model Structure” diagram is redundant with the “Implementation View: Implementation Subsystems” diagram, and can be omitted. For more information on Implementation Subsystems, see the Implementation Subsystems section.

9.1 Implementation Subsystems

RUP **Implementation Subsystems** are units of integration[20]. As such, they align very nicely with J2EE modules (**Implementation Subsystems** can be packaged and deployed in J2EE modules). Since one possible way to set-up the XDE project structure is to define an XDE project for each J2EE archive, it follows that identified **Implementation Subsystems** can be used to drive what XDE projects should be defined to support detailed design and implementation. Specifically, a RUP **Implementation Subsystem** can be represented in XDE as an XDE project. This is the approach used in these guidelines where **Implementation Subsystems** are represented using XDE projects.

The **Implementation Subsystems** for the examples in these guidelines are as follows:

- An Implementation Subsystem for each Design Subsystem in the “Overall Design Model”. For example: User Account Manager and Auction Manager.
- An Implementation Subsystem for the Auction Management presentation elements
- An Implementation Subsystem for the User Account Management presentation elements

[20] Guidelines for identifying **Implementation Subsystems** are not within the scope of this document. For more information, see RUP.

The associated XDE projects and models are shown in Figure 18.

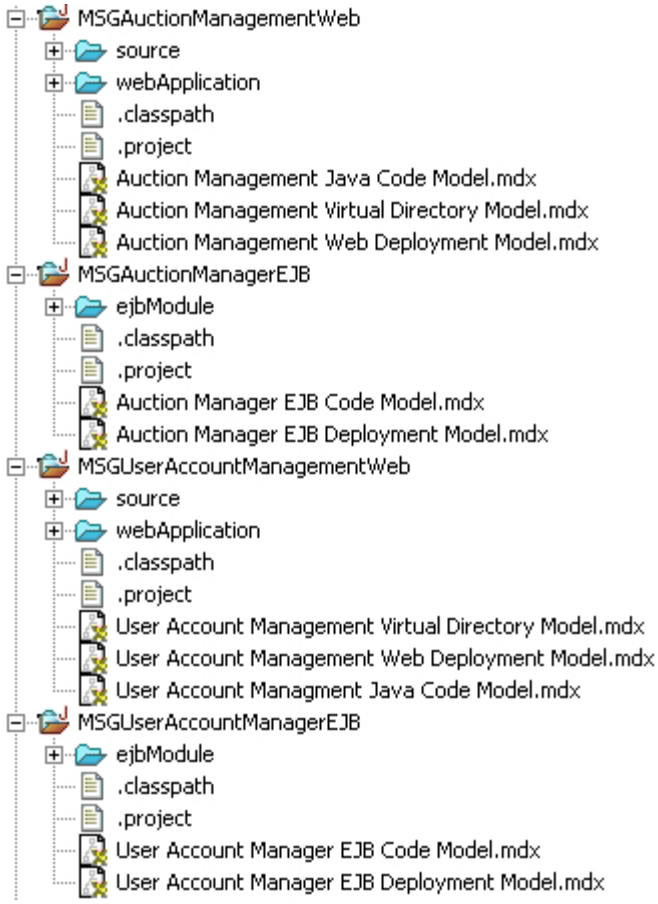

Figure 18: Implementation Subsystems as XDE Projects

When multiple Web and EJB projects are defined, it is strongly recommended that the names of the projects and contained model files be unique. This makes it much easier to interpret the diagrams in the models, as well as resolve inter-model references. In the example shown in Figure 18, the name of the **Implementation Subsystem** was used in the name of the project, as well as in the name of each of the contained models.

Alternatively, if the project is small, a RUP **Implementation Subsystem** can be represented in XDE as package in an XDE model. Figure 19 provides an example of the case where each Implementation Subsystem is represented using an XDE package (the “auctionmanager” and “useraccountmanager” packages).

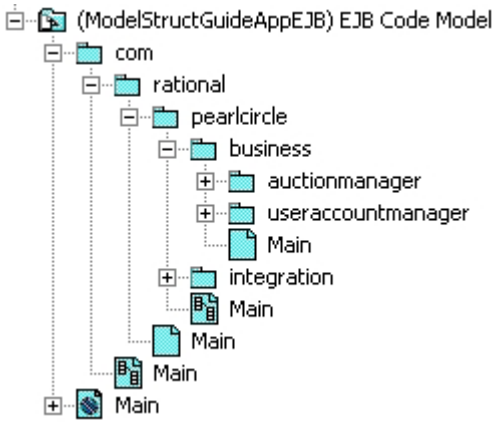

Figure 19: Implementation Subsystems as Packages

9.2 XDE Roundtrip Models

This section provides examples for the following XDE roundtrip models.

- (EJB Project) EJB Code Model (see the EJB Project: EJB Code Model section)
- (Web Project) Java Code Model (see the Web Project: Java Code Model section)
- (Web Project) Virtual Directory Model (see the Web Project: Virtual Directory Model section)

In general, when structuring your XDE roundtrip models, the recommendation is to align the structure as close as possible to the logical structure as described in the "Overall Design Model" (described in the Design Model section), taking implementation constraints into consideration, of course. By aligning the structure of the **Design Model** and the **Implementation Model** as close as possible, the traceability between them becomes implicit and more straightforward to maintain. This is important since the mapping between the **Design Model** and the **Implementation Model** is something that needs to be maintained and managed (as part of maintaining and managing the architecture).

The **Design Classes** that are part of the XDE roundtrip models can be created through one of the following

- XDE automation for creating implementation technology dependent elements like EJBs, servlets, etc., either by creating the elements from scratch or by transforming the elements from technology independent elements like **Analysis Classes**.
- XDE reverse engineering of an existing implementation
- Manual creation.

9.2.1 EJB Project: EJB Code Model

An EJB Code Model contains the Java resources needed to implement EJBs (e.g., implementation bean classes, home interface classes, remote interface classes, etc.).

The Source Root property of the EJB Code Model should be set to the directory where the source code will be placed. For example, the Source Root property of the EJB Code Model could be set to the "ejbModule" subdirectory of the EJB project[21].

[21] If you create the project using the XDE create project wizard, the project subdirectory is created automatically and the Source Root property of the code model is set automatically.

An example of an EJB Code Model is shown in Figure 20.

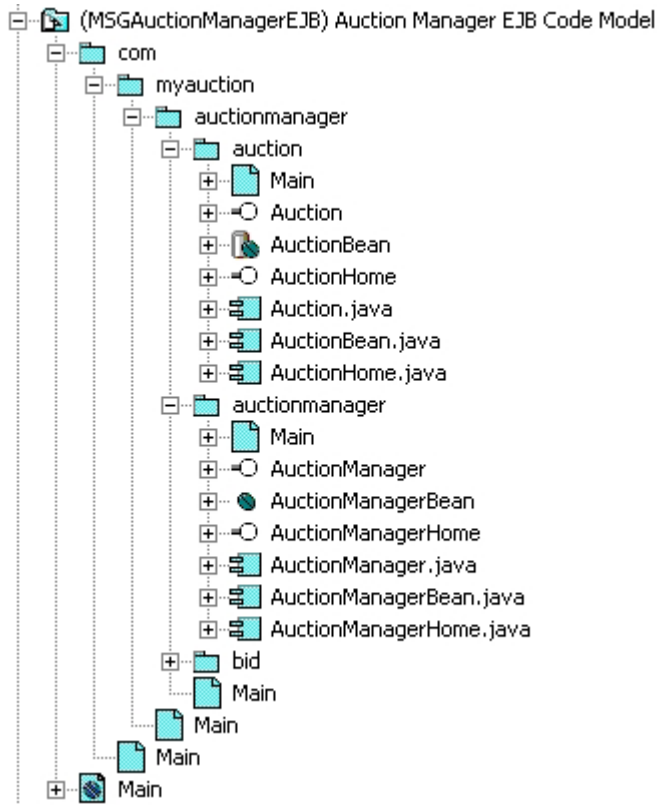

Figure 20: EJB Code Model Example

The EJB Code Model shown in Figure 20 contains the Java resources that implement the Auction Manager **Design Subsystem.** It is the implementation counterpart to the "Auction Manager" **Design Subsystem** package of the "Business" layer package in the "Overall Design Model" discussed in the Design Layers section.

As illustrated in Figure 20, the EJB Code Model structure follows the convention of using the domain name as the initial Java package name. The domain name for the sample application is "www.myauction.com". Therefore, the packages containing the implementation elements are placed within the "myauction" package within the "com" package. As a result, all Java elements within the "myauction" package will have a fully qualified name that is prefixed with "com.myauction". For example, the fully qualified name of the "auction" package is "com.myauction.business.auctionmanager.auction". The convention of using the domain name as the initial Java package name guarantees that Java class names will be unique, even if a third-party Java class library is incorporated.

Within the "myauction" package, the structure reflects the structure of the "Overall Design Model" (discussed in the Design Model section). There is a package for the design layer that contains the Auction Manager **Design Subsystem** ("business" package) and then there is a package that represents the Auction Manager **Design Subsystem** ("auctionmanager" package). Additional packages have also been defined in the EJB Code Model to collect related model elements (e.g., the "auctionmanager", "auction", and "bid" packages).
Note: Since the Java programming language does not allow spaces in package names, a Java package name might not be identical to the name of the associated "Overall Design Model" package.

As shown in Figure 20, an EJB Code Model not only contains the visual representation of the source code files (the .java elements), but it also contains the classes that specify those implementation elements. These classes represent RUP **Design Classes** that have evolved and matured to the point where they can be implemented and, in the case of XDE, roundtrip engineered. Note: XDE provides patterns that automatically create all the classes used to specify an EJB.

9.2.2 Web Project: Java Code Model

A Web Project Java Code Model contains the Java Web resources (e.g., JavaBeans, servlets, helper classes, etc.).

The Source Root property of the Web Project's Java Code Model should be set to the directory where the source code will be placed. For example, the Source Root property of the Web Project's Java Code Model could be set to the "Java Source" subdirectory of the Web project[22].

An example of a Web Project Java Code Model" is shown in Figure 21.

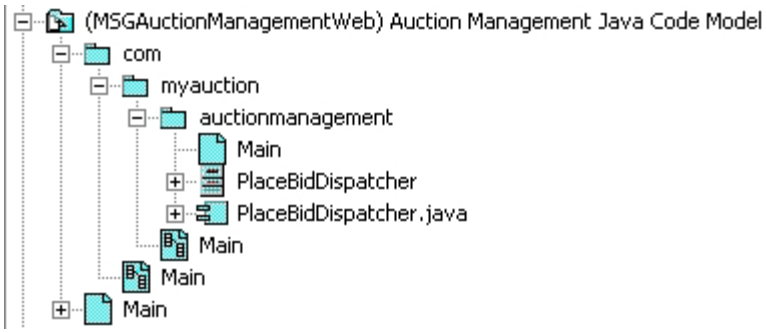

Figure 21: Web Project Java Code Model Example

The Java Code Model shown in Figure 21 contains the Java resources that implement the Auction Management presentation design elements. It is the Java implementation counterpart to the "Auction Management" package of the "Presentation" layer package in the "Overall Design Model" discussed in the Design Layers section).

As illustrated in Figure 21, the Web Project Java Code Model structure follows the convention of using the domain name as the initial Java package name. The domain name for the sample application is "www.myauction.com". Therefore, the packages containing the implementation elements are placed within the "myauction" package within the "com" package. As a result, all Java elements within the "myauction" package will have a fully qualified name that is prefixed with "com.myauction". For example, the fully qualified name of the "auctionmanagement" package is "com.myauction.presentation.auctionmanagement". The convention of using the domain name as the initial Java package name guarantees that Java class names will be unique, even if a third-party Java class library is incorporated.

Within the "myauction" package, the structure reflects the structure of the "Overall Design Model" (discussed in the Design Model section). There is a package for the design layer that contains the Auction Management presentation elements (the "presentation" package). Then there is a package that represents the Auction Management presentation elements (the "auctionmanagement" package).
Note: Since the Java programming language does not allow spaces in package names, a Java package name might not be identical to the name of the associated "Overall Design Model" package.

As shown in Figure 21, the Web Project Java Code Model not only contains the visual representation of the source code files (the .java elements), but it also contains the classes that specify those implementation elements. These classes represent RUP **Design Classes** that have evolved and matured to the point where they can be implemented and, in the case of XDE, roundtrip engineered.

9.2.3 Web Project: Virtual Directory Model

A Virtual Directory Model contains non-Java source code Web resources (for example, JSPs and HTML pages). There can be multiple Virtual Directory models per XDE Web project. Multiple Virtual Directory Models support the development of J2EE applications that have multiple virtual directories. In such applications, the resulting Web site is physically divided into directories that have no common root. For example, in an online retail store the www.mystore.com would be used for catalog shopping, whereas the order.mystore.com would be used for

22 If you create the project using the XDE create project wizard, the project subdirectory is created automatically and the Source Root property of the code model is set automatically.

monitoring order status.

The Source Root property of the Virtual Directory Model should be set to the directory where the Web resources that will be deployed to the Web container will be placed. For example, the Source Root property of the Virtual Directory Model could be set to the "Web Content" subdirectory of the Web project[23].

An example of a Virtual Directory Model is shown in Figure 22.

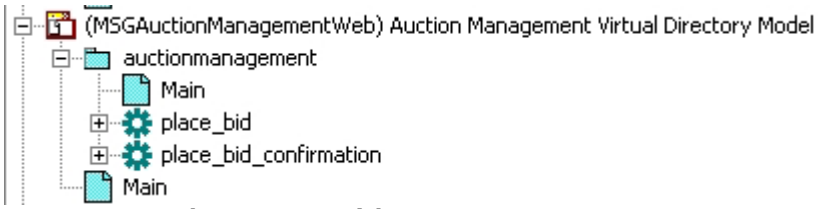

Figure 22: Virtual Directory Model Structure

The Virtual Directory Model shown in Figure 22 contains the non-Java source code resources that implement the Auction Management presentation design elements. It is the non-Java source code implementation counterpart to the "Auction Management" package of the "Presentation" layer package in the "Overall Design Model" discussed in the Design Layers section.

In this example, the structure of the Virtual Directory Model reflects the structure of the "Overall Design Model" (discussed in the Design Layers section). There is a package for each package in the "Overall Design Model" whose content will be implemented by non-Java elements to be deployed to the Web container, such as JSPs and HTML pages. Due to the constraints imposed by the naming of directories accessed by the Web server, the names of the directories under the virtual directory are not always identical to those in the "Overall Design Model".

As shown in Figure 22, the Virtual Directory Model contains the visual representation of the classes that specify the implementation elements. These classes represent RUP **Design Classes** that have evolved and matured to the point where they can be implemented and, in the case of XDE, roundtrip engineered. Unlike, the EJB Code Model and Web Project Java Code Model, XDE does not produce a visual representation of the associated source code files in the Virtual Directory Model. The name of the associated source code file is maintained as a property of the class.

10. Deployment Model

The RUP **Deployment Model** is represented by multiple XDE models - the "EAR Deployment Model" and a set of individual XDE deployment model(s), one for every XDE project that contains elements to be deployed. Using multiple deployment models leverages XDE's deployment automation (XDE provides automation for the creation and refinement of J2EE archive files and deployment descriptors, and the synchronization of these model elements with the model elements whose source is packaged in the archive).

This section provides examples of the following deployment models:

- (Application Project) EAR Deployment Model (see the EAR Deployment Model section)
- (EJB Project) EJB Deployment Model (see the EJB Deployment Model section)
- (Web Project) Web Deployment Model (see the Web Deployment Model section)

The following are some general items worth noting regarding deployment in XDE:

- Deployment descriptors are not explicitly represented in the XDE deployment models as files (UML artifacts). Instead, they are represented by the contents of the XDE deployment models. XDE uses the elements contained in the deployment models, and the property values assigned to those elements, to determine the content information written to the model's deployment descriptor files.

[23] If you create the project using the XDE create project wizard, the project subdirectory is created automatically and the Source Root property of the Virtual Directory Model is set automatically.

- XDE uses an EAR Deployment Model for all deployments, even when you deploy only a single EJB JAR or WAR. This reflects a limitation of most application servers, which require an EAR for all deployments. Hence, even if you are modeling only EJB-JAR's or WAR's, you should still use an EAR Deployment Model for deployment to the application servers supported by XDE. However, XDE can *export* any archive to the file system. This ability can be used to deploy to application servers that XDE does not support. After you export the archive, you can invoke the server-specific tools to complete deployment.
- Different J2EE archive files can be defined for different deployment environments (testing, production, etc.). These archives can be defined in the same XDE deployment model or in separate XDE deployment models. XDE automation supports both, but it does define default deployment models for projects and default archives per deployment model, but you can adjust these definitions, as they are modeled in the XDE deployment models. Whichever approach you choose, EJB deployment models should be defined in EJB projects, and Web deployment models should be in Web projects ().[24]

10.1 EAR Deployment Model

The "EAR Deployment Model" contains the visual representation of the J2EE Application archive file (.EAR file), EAR deployment descriptor information, and the node on which the EAR is to be deployed. The "EAR Deployment Model" may also contain diagrams that show what J2EE modules (from other deployment models) are contained within the EAR.

The "EAR Deployment Model" can also be used to describe the deployment configuration of the application as a whole, as well as the Deployment View of the architecture. The "EAR Deployment Model" can contain diagrams that show all of the deployment nodes and their connections, as well as what archives are to be deployed to what nodes. Such diagrams would reference elements (nodes and archives) from all of the individual deployment models.

An example of an "EAR Deployment Model" is shown in Figure 23.

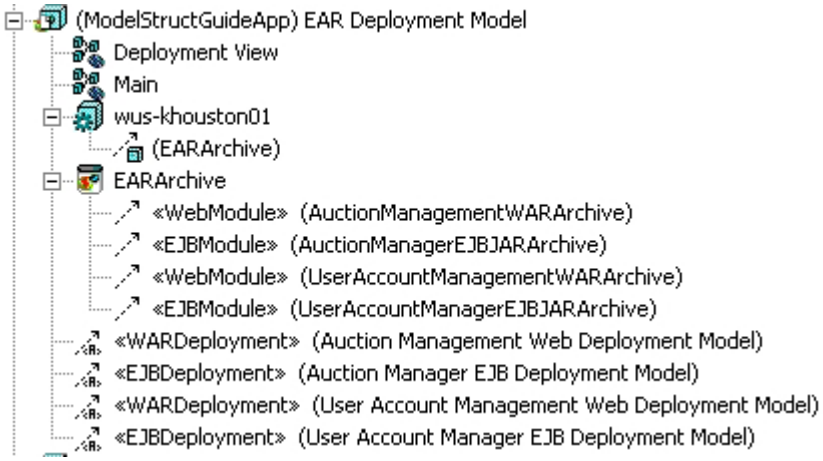

Figure 23: EAR Deployment Model

The "Deployment View" diagram represents the Deployment View of the architecture. It includes the deployment nodes, their interconnections, and what J2EE archives are to be deployed to these nodes. In some cases, this diagram may just include the J2EE Application archive and the application server node it is to be deployed to. However, in the case where standalone J2EE Module archives are deployed to specific nodes, this diagram should show what archives are deployed to what nodes. For information on architectural views, see RUP.

---

[24] An EJB Deployment Model must be in an EJB Project, but it doesn't have to be the *same* project as the corresponding EJB Code Model. In fact, you can "mix and match" EJBs from different code models into one Deployment Model. Same comment applies for the Web models.

10.2 EJB Deployment Model

The EJB Deployment Model contains EJB component(s), the EJB-JAR artifact, and the EJB deployment descriptor information. The EJB Deployment Model may also contain diagrams that show what implementation elements (from the EJB Code Model) are contained in the EJB-JAR. To be precise, in the EJB Deployment Model, EJB components are used to represent the implementation elements, and it is the EJB components that are mapped to the EJB-JAR.

An example of an EJB Deployment Model is shown in Figure 24.

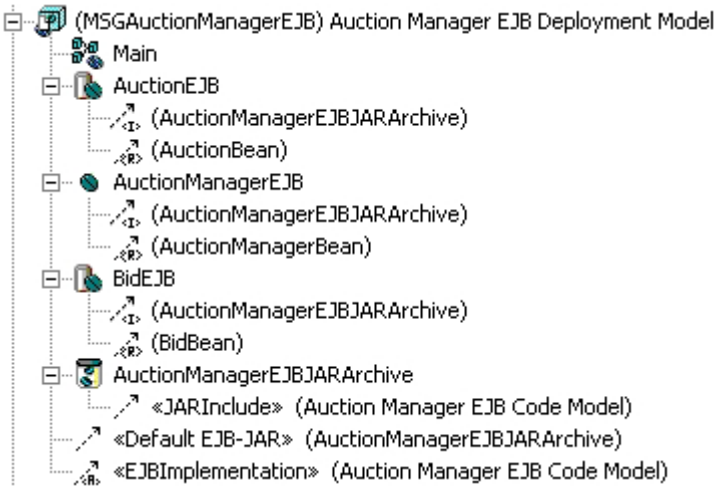

Figure 24: EJB Deployment Model Example

10.3 Web Deployment Model

The Web Deployment Model contains Web component(s), the WAR archive file, and Web deployment descriptor information. The Web Deployment Model may also contain diagrams that show what implementation elements (from the Web project roundtrip models) are contained in the WAR. To be precise, in the Web Deployment Model, Web components are used to represent the implementation elements, and it is the Web components that are mapped to the WAR.

An example of a Web Deployment Model is shown in Figure 25.

Figure 25: Web Deployment Model

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

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

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