Rational XDE™ Model Structure Guidelines for Microsoft® .NET

Table of Contents

1. Introduction

This document provides recommendations on how to represent and structure the RUP model artifacts in Rational XDE™, Microsoft® .NET 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 it does not, which may 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 model files 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.

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.

A VS.NET solution is a collection of projects and within each project there can be one or more XDE model files[1]. Thus, the project structure affects the number of model files that are created, as well as their content.

A .NET Enterprise application may consist of multiple projects depending on the how the application is architected. For example, if the application implements a XML Web service, Windows and Web interface, it is recommended that the solution have a Web Service, Windows Application and Web Application projects respectively. For more information on different VS.NET project templates, see the VS.NET help.

For a .Net enterprise application that is being developed by multiple people, we recommend that you create the following XDE projects and models

---

[1]

XDE defines two types of model files - code and non-code model files. Code model files are used to model the C# language specific elements of a project while the non-code model files have no mapping to an implementation language and act as analysis and design models. There can be only one code model file associated with a project while there can be more than one non-code model files associated with a project.

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) - "Deployment Model" (.Net: 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$_2$" (Data: *vendor specific physical data model file*) $_2$ - "Domain Model" (Data: *vendor specific domain model file*)
.Net Class Library Projects (XDE Class Library Modeling Project)	A Class Library project is a "C#" VS.NET project that contains the C# elements (i.e. classes, interfaces, etc.) needed to implement a class library, (Auction Manager in this example). The contained elements are packaged and deployed as a .NET assembly. Note that a .NET application may contain more than one such class library projects. As noted earlier, the selection of the number of projects is an architectural choice and may vary for different applications.	- ".Net Code Model" (.Net: .Net Code Model) - ".Net Deployment Model" (.Net: .Net 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 .Net assembly. Separate Web projects can be defined for specific areas of the presentation logic. The recommendation is to create a Web project for each assembly that needs to be produced. If separate projects are defined, then the name of the project should reflect its contents.	- ".Net Code Model" (.Net: .Net Code Model) - ".Net Deployment Model" (.Net: .Net Deployment Model)

An example of this project and model organization using both the Solutions Explorer and the XDE Model Explorer views is shown in Figure 1. The Solution Explorer view is shown on the left side of the figure and the Model Explorer view is shown on the right.

2

Rational XDE provides physical database support for multiple database vendors. A vendor specific template exists for each database vendor supported by XDE.
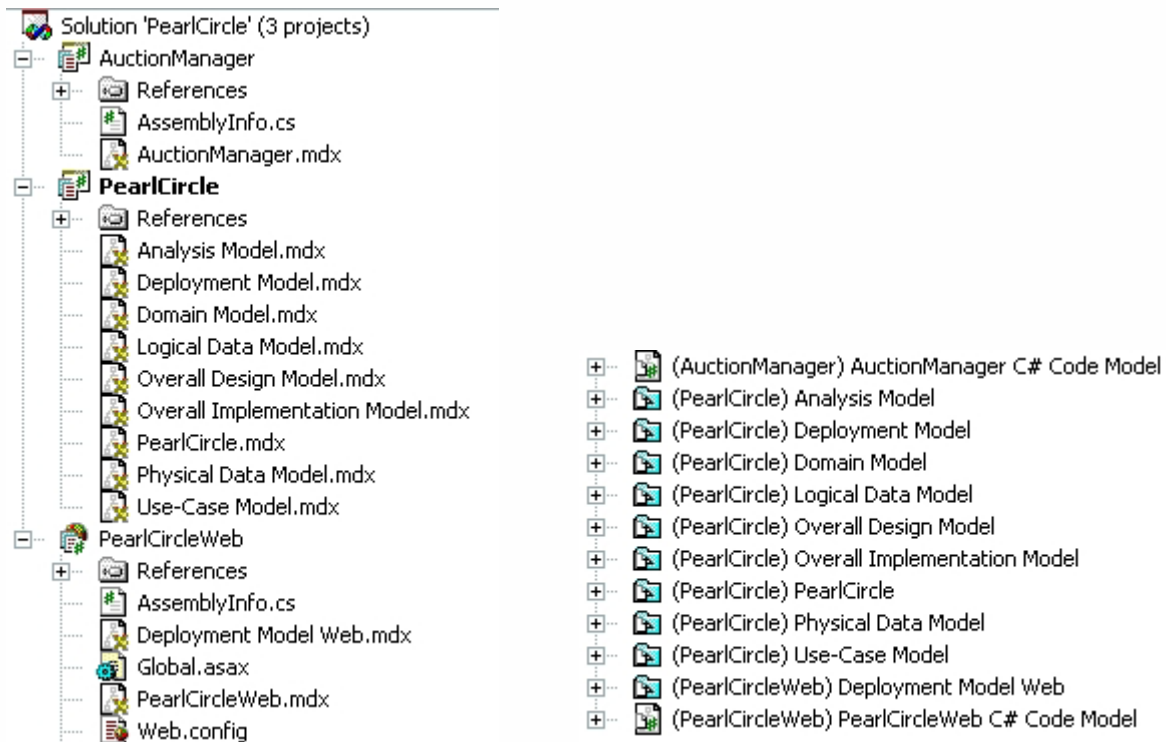
Figure 1: Project Model and Organization

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 .Net code 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 the non-Web aspects of the Class Library application. It contains the models Modeling that describe the application as a Project) whole, the Data Model, and the .Net-specific models.	The Application Project represents	- "Use-Case Model" (Rational XDE: Use-Case Model) - "Physical Data Model" (Data: *vendor specific physical data model file*) - ".Net Code Model" (.Net: .Net Code Model) - ".Net Deployment Model" (.Net: .Net Deployment Model)
Web Project (XDE Web	Web projects represent the Web resources of the application. The	- ".Net Code Model" (.Net: .Net Code Model) ".Net Deployment Model" (.Net: .Net Deployment

| Modeling
Project) | contained elements are packaged
and deployed in a .Net assembly. | Model) |

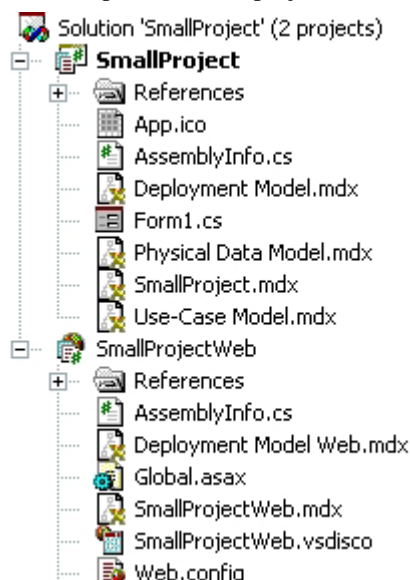An example of a small project and model organization is shown in Figure 2.

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 .Net 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.

The structure of the XDE models shown in Figure 1 is the focus of the remainder of this document.

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: Design Model

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: Implementation Model
Deployment Model	Application Project: Deployment Model

5. Use-Case Model

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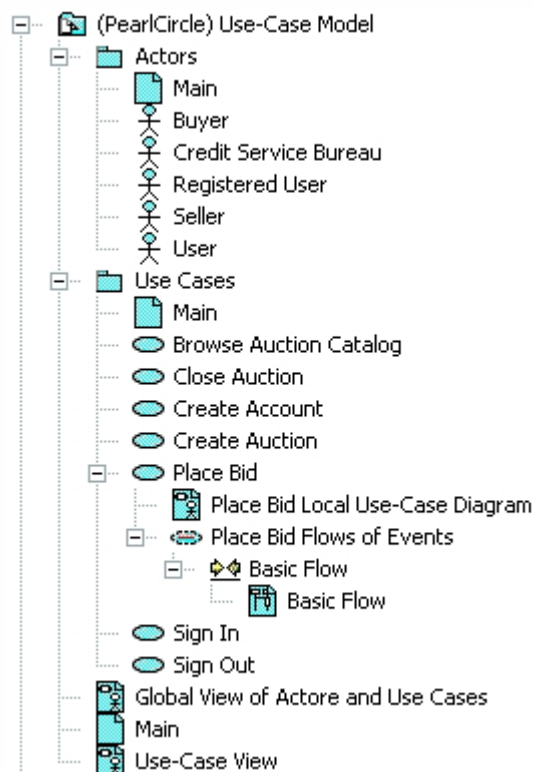

Figure 3: "Use Case Model" Structure

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 diagrams that describe

graphically the flows of events described in the use-case description (i.e., the interactions between the **Actors** and the **Use Case**). This collaboration instance 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 describe interactions of elements within the application.

☐ The "Place Bid Flow 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 in which 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.
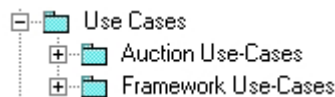

Figure 4: Additional Use-Cases Package Partitioning

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 not maintained, then the **Analysis Classes** will be moved into the appropriate **Design Model** partition[3] and refined. Another option is to simply 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.

---

[3]
As you will see later, the right "Overall 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.
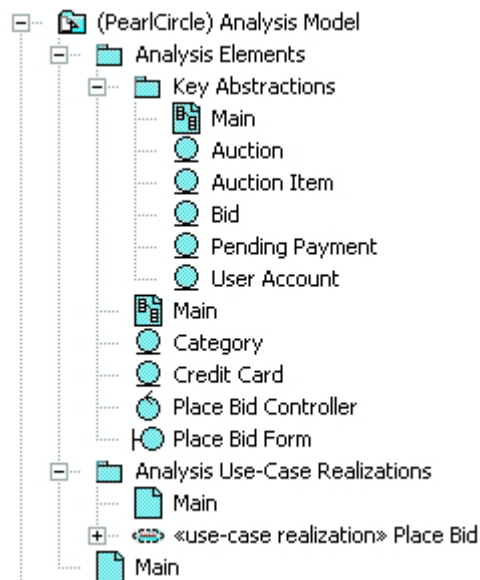
Figure 5: Analysis Model Structure

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**. 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.
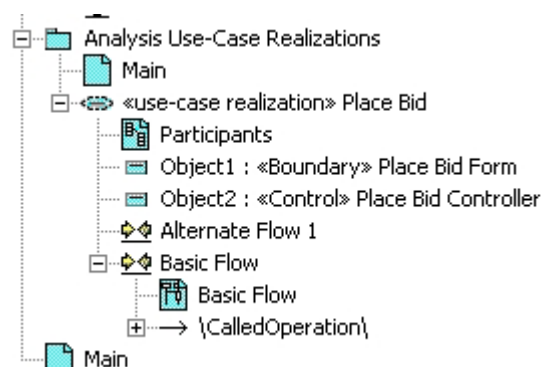

Figure 6: "Analysis Use-Case Realization" Package Structure

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.

The "Overall Design Model" describes the design of the application as a whole and contains elements that span multiple 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 only have one target implementation language and your team is small.

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.

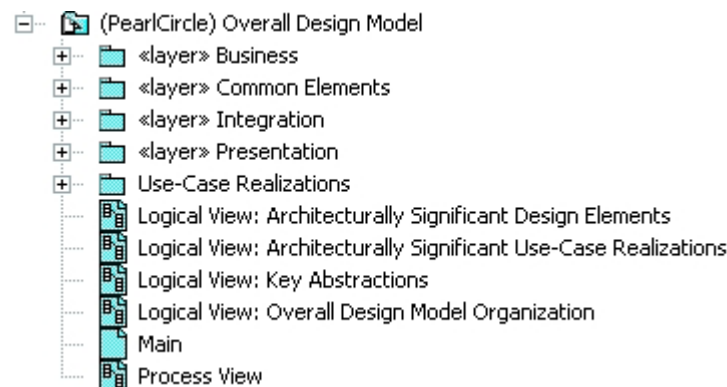The recommended structure of the "Overall Design Model" is shown in Figure 7.

Figure 7: Overall Design Model Structure

This 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 "Overall Design Model" are discussed under the heading in the Design UseCase 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.
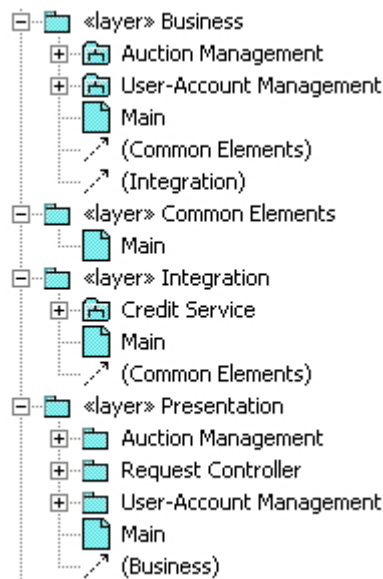

Figure 8: Design Package Partitioning Example

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 .Net application, the design elements that might reside in the "Presentation" layer package include Active Server Pages (ASP.NET). 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 "Overall 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.
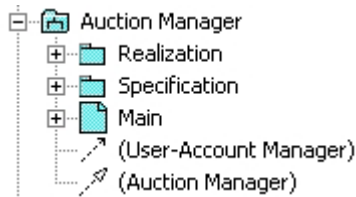

Figure 9: Design Subsystem Structure

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.[4] An example of a subsystem specification is shown in Figure 10.

Figure 10: Design Subsystem Specification Example

---

[4]
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.

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.
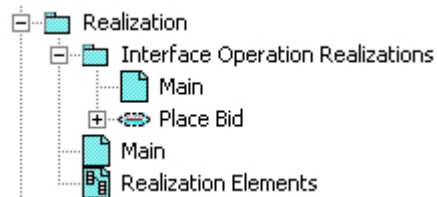

Figure 11: Design Subsystem Realization Example

The "Realization Elements" diagram contains references to the design elements that realize the subsystem. The design elements themselves reside in a .Net Code Model, where they participate in roundtrip engineering. For more information, see the Implementation Model section.

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.[5] An example of an "Interface Operation Realizations" package is shown in Figure 12.
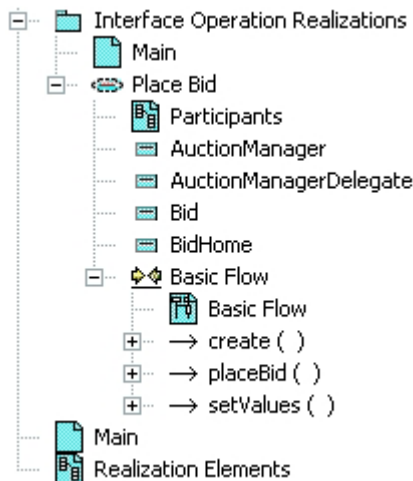

Figure 12: Interface Operation Realizations Package Example

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).

---

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

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 13.
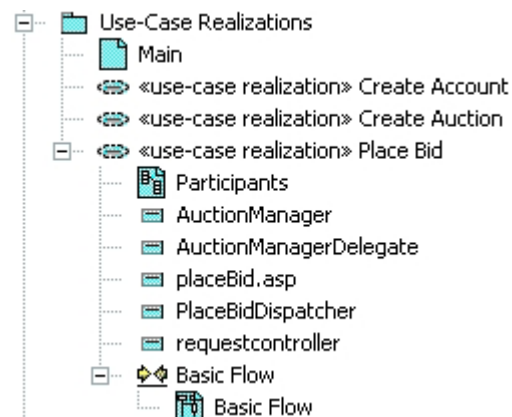

Figure 13: Design Use-Case Realization Structure

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.

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 "Overall 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**. If a Logical Data Model were being maintained (i.e., there is a separate "Logical Data Model"), then this diagram would contain elements from the XDE Logical Data Model. For more information on architecture views, see RUP.

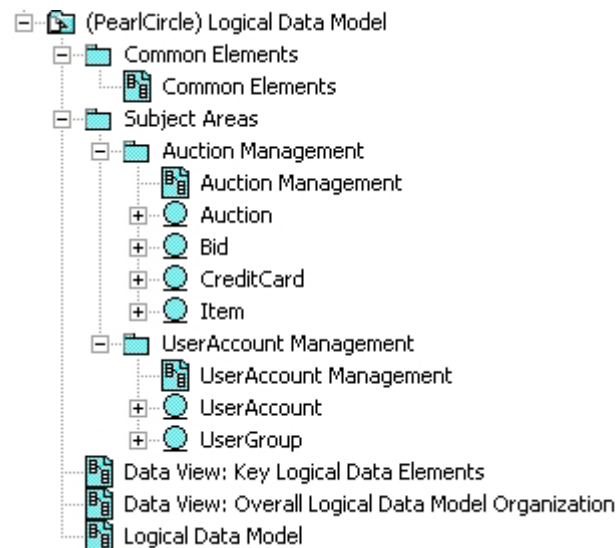An example of the recommended structure for the XDE Logical Data Model is shown in Figure 14.

Figure 14: XDE 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 XDE 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 XDE Physical Data Model is shown in Figure 15.
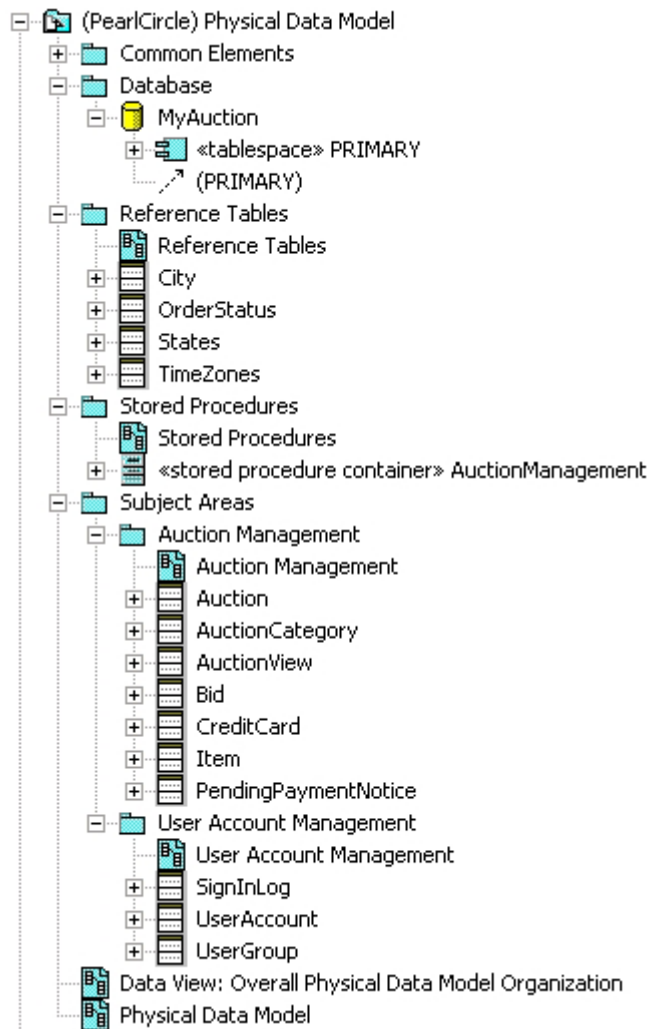

Figure 15: XDE Physical Data Model Structure

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[6].

The "Subject Areas" package contains packages that group logically related sets of tables and views[7]. 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[8].

  ☐ 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[9] 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.

6

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.

7

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.

[8] Typically the database is reverse engineered once, and then all future updates are synchronized using XDE's Compare and Sync functions.

9

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 XDE Domain Model[10] is shown in Figure 16.
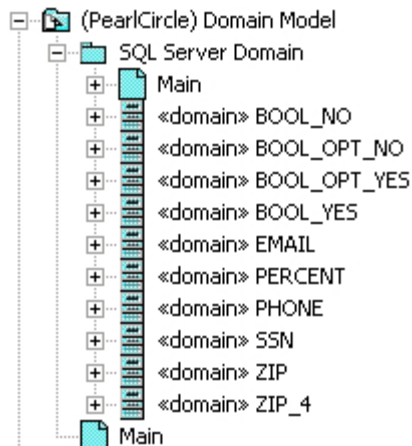

Figure 16: XDE Domain Model Structure

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 **Implementation Model**, as defined in RUP, contains the implementation elements, both their visual and physical representation (e.g., the UML elements representing the implementation elements, as well as the physical files in the file system). The value of XDE with regards to the **Implementation Model** is its ability to automatically synchronize these separate representations via roundtrip engineering.

Within XDE, the **Implementation Model** is represented in multiple XDE models, an example of which is shown in Figure 17:
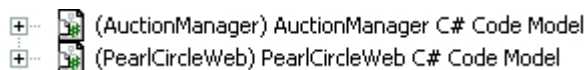

Figure 17: Implementation XDE Models

In this example, we defined the following XDE model files to represent the **Implementation Model**:

- The "AuctionManager C# Code Model" is an XDE code model file that contains the Microsoft® Visual C# code elements that constitute the Auction Manager **Implementation Subsystem**. The elements in this model participate in XDE roundtrip engineering.
-
- The "PearlCircleWeb C# Code Model" XDE model file contains the ASP.NET C# code elements (Web Forms, Web Controls, and HTTP Handlers) that constitute PearlCircle Web **Implementation Subsystem**. This model corresponds to a VS.NET Web Application project. The elements in this model participate in XDE roundtrip engineering.

Keep in mind that there can only be one XDE code model file per VS.NET project. The selection of the number of projects and individual models is an architectural choice and may vary for different projects. For more information, see XDE online help.

Each of these models is described in more detail in a later section.

10

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.

9.1 C# Code Model for Class Library Project

The "C# Code Implementation Model" contains the elements that are implemented using C#.
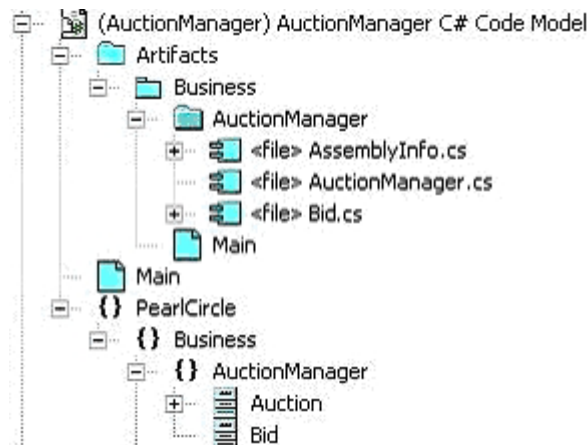

Figure 18: "Auction Manager C# Code Model" Structure

In this example, the structure of the "Auction Manager C# Code Model" reflects the structure of the "Overall Design Model" (discussed in section 7). There is a package (representing a .NET namespace) for each "Overall Design Model" package whose content will be implemented in C# (this includes Serviced Components and other supporting C# classes). .NET namespaces in XDE are modeled as package with a *namespace* stereotype (closed curly braces '{ }' icon). Since the C# programming language does not allow spaces in namespace names, a C# namespace name may not be identical to the name of the equivalent "Overall Design Model" model element.

As shown in Figure 18, the "C# Code Model" contains the visual representation of the source code files (the .cs elements). XDE creates an *Artifacts* package for each code model file that contains the source code files. These source code files map to the classes defined in the "Overall Design Model" (see section 7) that have evolved/matured to the point where they can be implemented (and in the case of XDE, roundtrip engineered).

As we can see from Figure 18, the "C# Code Model" structure follows the convention of using the company name as the initial C# namespace name. The company name for the sample application is "Pearl Circle". Thus, the packages containing the implementation elements are placed within the *PearlCircle* namespace. As a result, all C# elements within the *PearlCircle* namespace will have a fully qualified name that is prefixed with "PearlCircle". For example, the fully qualified name of the *AuctionManager* namespace is "*PearlCircleBusiness.AuctionManager*". The convention of using the company name as the initial C# namespace name guarantees that C# class names will be unique, even if a third-party C# class library is incorporated.

9.2 C# Code Model for Web Application Project

XDE Web Model contains elements that correspond to an Active Server Page .NET (ASP.NET) Web Application project. Figure 19 provides an example of the "PearlCircleWeb" XDE code model file that presents the elements of the ASP.NET project.
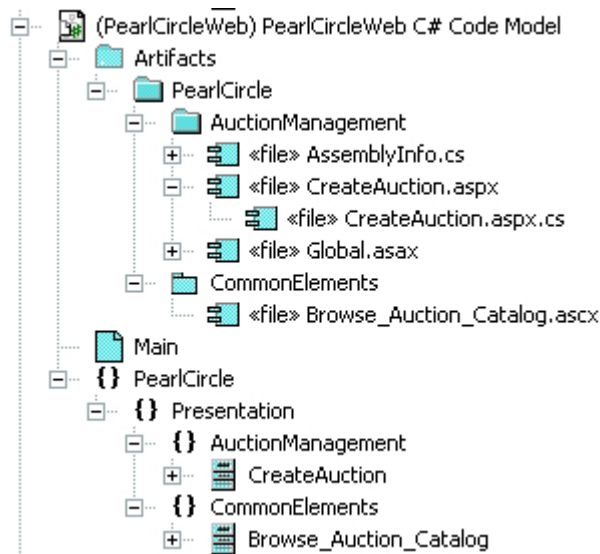
Figure 19: "PearlCircle Web C# Model" Structure

In this example, the structure of the "PearlCircle Web C# Model" reflects the structure of the "Overall Design Model" (discussed in section 7). There is a namespace for each package in the "Overall Design Model" whose content will be implemented in C# (this includes code-behind classes and other supporting C# classes). .NET namespaces in XDE are modeled as package with a *namespace* stereotype (closed curly braces '{ }' icon). Since the C# programming language does not allow spaces in namespace names, a C# namespace name may not be identical to the name of the equivalent "Overall Design Model" model element.[11]

The C# code-behind classes (.aspx.cs or .ascx.cs) associated with the Web Forms and Web Controls, and other supporting C# classes such as HttpHandlers, can be roundtrip engineered using XDE. Note that currently XDE does not support RTE for .aspx or .ascx files.[12]The class *CreateAuction* shown in Figure 19 models the code-behind class residing inside file *CreateAuction.aspx.cs* . In the Artifacts package, the *CreateAuctiont.aspx.cs* file is shown under the *CreateAuction.aspx* file.

If all of the architecturally significant functionality is contained within the code-behind classes, then the automatically generated XDE code model is all that is needed. However, in cases where an architecturally significant function is implemented inside a Web Control file (.ascx), a class corresponding to this file may be manually added to the model. An example of this is the *Browse_Auction_Catalog* class shown in Figure 19. Note that this class was manually added to this diagram.

10. Deployment Model

The **Deployment Model** is represented in an XDE "Blank Model" file named "Deployment Model".

The "Deployment Model" contains the nodes and their connections that represent the network configuration of the deployment environment. It also identifies the implementation elements that will be deployed on these nodes.

An example "Deployment Model" is shown Figure 20.

---

[11]

In Visual Basic.Net, the project property "Root Namespace" must be the namespace that is also defined for the class you are forward engineering. The recommended best practice that also addresses this issue is to remove the text from the "Root Namespace" project property before reverse engineering the Visual Basic project. All namespaces will thereafter be handled via Namespace declarations in code.

[12]

The ASCX and ASPX files are created with the appropriate form and other required tags. For example if you add a label or textbox to the code-behind class in XDE, it will forward engineer the private member variable into the code-behind class. The only reason the controls do not appear in the Visual Studio Designer is that the tags in the ASPX and code in the code-behind to create and position those elements was not generated. For more information please see the VS.NET documentation.
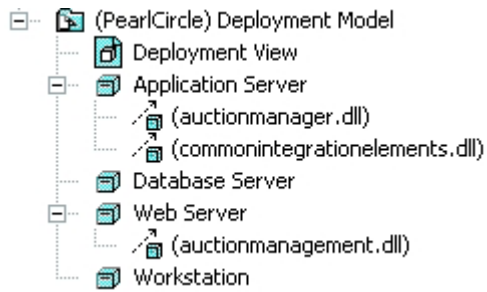
Figure 20: Deployment Model Structure

In this example, the identified nodes are the "Database Server", "Application Server", and "Web Server". The auctionmanagement.dll is deployed on the Web Server. The auctionmanager.dll and commonintegrationelements.dll are deployed on the Application Server.

The "Deployment Model" model file may also contain architectural view diagrams. In Figure 20, the "Deployment View" diagram represents the Deployment View of the architecture. For more information on architectural views, see RUP.