Benefits of Modeling ..........2

What Is UML? ..........2
What are UML views? ..........2
What are UML diagrams? ..........3

How to Create Use Cases ..........3
Identifying the system ..........3
Identifying actors ..........3
Defining the interactions between the actor and the system ..........3
Determining the system boundary ..........3
Use case diagram artifacts ..........3

What Are Usage Scenarios? ..........3
Why create current state usage scenarios? ..........3
Benefits ..........3
Creating a usage scenario ..........3

How to Identify Candidate Business Objects ..........3
Identifying objects ..........3
Example of business objects ..........3

How to Identify Services ..........3
Identifying services ..........3
Examples of services ..........3

How to Identify Attributes ..........3
Identifying attributes ..........3

How to Identify Relationships ..........3
Identifying relationships from use cases ..........3
Examples of relationships ..........3

How to Refine UML Models ..........3
Objects and services inventory ..........3
Class diagrams ..........3
Class diagram artifacts ..........3
Sequence diagrams ..........3
Sequence diagram artifacts ..........3
Activity diagrams ..........3
Activity diagrams artifacts ..........3
Component diagrams ..........3
Component diagram artifacts ..........3

## *Benefits of Modeling*

You use models to describe both the current and proposed solution to a business challenge. Some of the benefits of using models are:

- Models provide you with a common terminology that can describe both the current and proposed solutions.
- Models help to describe complex problems in a simpler structure and enable easy communication.
- Models enable consensus by helping the project team understand the business challenge, the business and user requirements, and the information that must be gathered.

As the business processes are modeled and adapted to reflect the requirements, you can build a model of the architecture that describes the final business solution. Two commonly used modeling notations are:

- Unified Modeling Language (UML)
- Object Role Modeling (ORM)

## *What Is UML?*

UML is a standard modeling language that you use to model software systems of varying complexities. These systems can range from large corporate information systems to distributed Web-based systems.

UML was developed to provide users with a standard visual modeling language so that they can develop and exchange meaningful models. UML is independent of particular programming languages and development processes. You use UML to:

- Visualize a software system with well-defined symbols. A developer or application can unambiguously interpret a model written in UML by another developer.
- Specify the software system and help build precise, unambiguous, and complete models.
- Construct models of the software system that can correspond directly with a variety of programming languages.
- Document the models of the software system by expressing the requirements of the system during its development and deployment stages.

**More Info**
For additional information about UML, you might find the following references useful: ***The Unified Modeling Language User Guide*** by Grady Booch, Ivar Jacobson, and James Rumbaugh (Addison-Wesley, 1999) and ***Use Case Driven Object Modeling with UML***: A Practical Approach by Doug Rosenberg with Kendall Scott (Addison-Wesley, 1999).

- It is a simple, extensible, and expressive visual modeling language.
- It consists of a set of notations and rules for modeling software systems of varying complexities.
- It provides the ability to create simple, well-documented, and easy to understand software models.
- UML is both language independent and platform independent.

### What are UML views?

UML enables system engineers to create a standard blueprint of any system. UML provides a number of graphical tools that you can use to visualize and understand the system from different

viewpoints. You can use diagrams to present multiple views of a system. Together, the multiple views of the system represent the model of the system.

You use models or views to depict the complexity of a software system. The various UML views depict several aspects of the software system. The views that are typically used are:

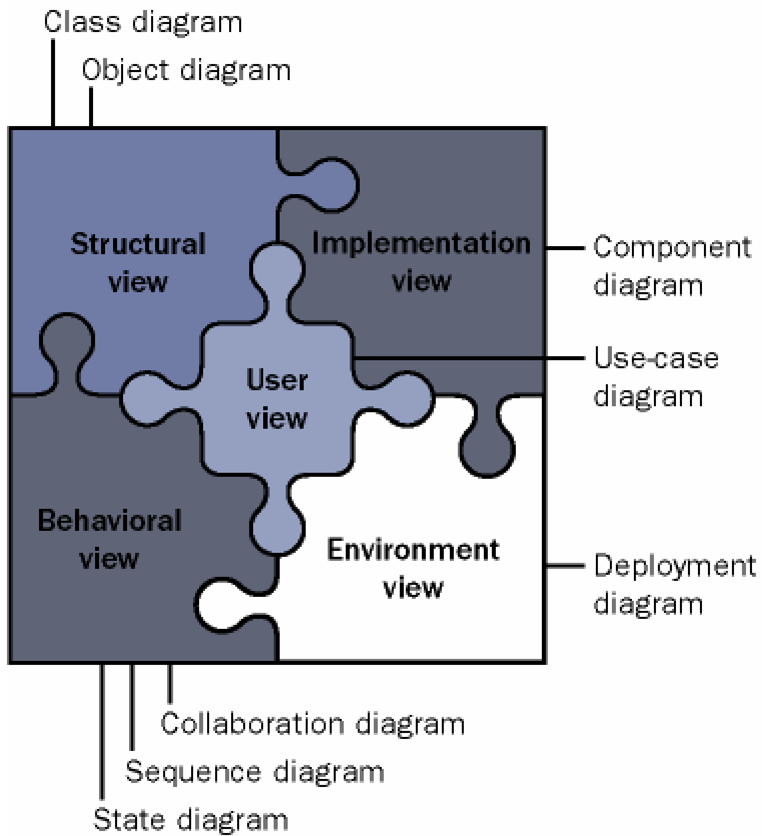
- **The user view**. The user view represents the goals and objectives of the system from the viewpoint of the users and their requirements for the system. This view represents the part of the system with which the user interacts. The user view is also referred to as the use-case view.
- **The structural view**. The structural view represents the static or idle state of the system. The structural view is also referred to as the design view.
- **The behavioral view**. The behavioral view represents the dynamic or changing state of the system. The behavioral view is also referred to as the process view.
- **The implementation view**. The implementation view represents the structure of the logical elements of the system.
- **The environment view**. The environment view represents the distribution of the physical elements of the system. The environment of a system specifies the functionality of the system from the user's point of view. The environment view is also referred to as the deployment view.

## What are UML diagrams?

The various UML views include diagrams that provide multiple perspectives of the solution being developed. You might not develop diagrams for every system you create, but you must understand the system views and the corresponding UML diagrams. Similarly, you might not use every diagram to model your system. You need to identify which models will best suit the needs of modeling the system successfully.

Use the following UML diagrams to depict various views of a system:

- **Class diagrams**. A class diagram depicts various classes and their associations. Associations are depicted as bidirectional connections between classes.
- **Object diagrams**. An object diagram depicts various objects in a system and their relationships with each other.
- **Use case diagrams**. A use case diagram represents the functionality that is provided to external entities by the system.
- **Component diagrams**. A component diagram represents the implementation view of a system. It represents various components of the system and their relationships, such as source code, object code, and execution code.
- **Deployment diagram**. A deployment diagram represents the mapping of software components to the nodes of the physical implementation of a system.
- **Collaboration diagrams**. A collaboration diagram represents a set of classes and the messages sent and received by those classes.
- **Sequence diagrams**. A sequence diagram describes the interaction between classes. The interaction represents the order of messages that are exchanged between classes.
- **State diagrams**. A state diagram describes the behavior of a class when the external processes or entities access the class. It depicts the states and responses of a class while performing an action.




## *How to Create Use Cases*

Use cases are functional descriptions of the transactions that are performed by the system when a user initiates an event or action. The use cases that you develop should represent the system processes, including all events that can occur in all possible situations.

Use cases consist of elements that are inside the system and are responsible for the functionality and behavior of the system. They are the actions that the system performs to generate the results that the users request. This model allows the project stakeholders to agree on the capabilities of the system and system boundary.

A use case diagram documents the following design activities:

- Identifying the system
- Identifying actors
- Defining the interactions between the actor and the system
- Determining the system boundary

### Identifying the system

A system is a collection of subsystems that have a real-world purpose. For example, in a sales and marketing scenario, the order system might have a subsystem that determines applicable discounts for a customer invoice. When you develop use cases, you identify a single system or subsystem. A collection of use cases indicates the relationships among the subsystems that make up the system, in addition to the relationships between systems that interact with each other.

## Identifying actors

The actor is an integral part of the use case. The use case represents the interactions between an actor and the system. An actor is an entity that interacts with the system to be built for the purpose of completing an event. An actor can be:

- A user of the system.
- An entity, such as another system or a database, that resides outside the system.

The roles played by actors explain the need and outcome of a use case. By focusing on the actors, the design team can concentrate on how the system will be used instead of how it will be developed or implemented. Focusing on the actors helps the team to refine and further define the boundaries of the system. Defining the actors also helps to identify potential business users that need to be involved in the use case modeling effort.

When looking for actors, ask the following questions:

- Who uses the system?
- Who starts the system?
- Who maintains the system?
- What other systems use this system?
- Who gets information from this system?
- Who provides information to the system?
- Does anything happen automatically at a preset time?
- Who or what initiates events with the system?
- Who or what interacts with the system to help the system respond to an event?
- Are there any reporting interfaces?
- Are there any system administrative interfaces?
- Will the system need to interact with any existing systems?
- Are any actors already defined for the system?
- Are there any other hardware or software devices that interact with the system and that should be modeled during analysis?
- If an event occurs in the system, does an external entity need to be informed of this event? Does the system need to query an external entity to help it perform a task?

## Defining the interactions between the actor and the system

After you have identified the system and the actor, you need to describe the interaction between them. You need to create one use case for each interaction. Describe only those interactions that are important to the business challenge and the vision statement.

## Determining the system boundary

One of the more difficult aspects of use case modeling is determining the exact boundary of the system to be built. People who are new to use case modeling might find it difficult to determine whether certain actors should be part of the system.

To define the boundary of the system, the team should try to answer the following questions:

- What happens to the use cases associated with that actor?
- Who or what interacts with those use cases now?
- What if you find new requirements? Will these requirements be a part of the system?
- Are these requirements necessary for this system?
- Are these requirements something this system would logically do?

- Can these requirements be handled by one of the current actors?
- Are these requirements something the customer/user would expect the system to do?

Typically, an actor is shown as a stick figure, a single use case is shown as an ellipse, and a set of cases are enclosed in a box, which represents a system.

## Use case diagram artifacts

- An ***actor*** is a person, system, piece of hardware, or other thing that interacts with your system.

- A ***use case*** is a piece of functionality the system will provide. It is usually named in the format <verb><noun>, such as "Deposit Check" or "Withdraw Cash." Use cases are high-level and implementation-independent.

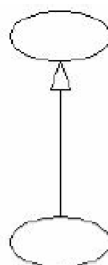
- A ***communicates relationship*** between an actor and a use case indicates that the actor initiates the use case. An actor may initiate one or more use cases.

- An ***includes relationship*** suggests that one use case must include another. In other words, running one use case means that the other must be run as well. One use case may be included by one or more other use cases.

- An ***extends relationship*** is used when one use case optionally extends the functionality provided by another. In other words, if one use case runs, an extending use case may or may not run.

- A ***package*** is a UML mechanism used to group items together. Grouping can help to organize the model, and can also help in managing any changes in the model elements. You may nest one package inside another to further organize the model.

- A ***generalization relationship*** between two use cases indicates that one use case (the child) inherits all of the functionality provided by the other use case (the parent). A generalization relationship between actors indicates that one actor (the child) inherits the characteristics of another actor (the parent). The child actor may initiate all of the use cases that the parent can initiate.



## *What Are Usage Scenarios?*

Use cases describe the high-level interactions between an actor and a system. The use cases grouped together describe a workflow process in detail. Usage scenarios provide additional information about the activities and task sequences that constitute a process. Usage scenarios document the sequence of tasks.

Usage scenarios describe in detail a particular instance of a use case. It takes many usage scenarios to document a use case completely. Whereas use cases are diagrams, usage scenarios are narratives.

Usage scenarios depict objects in a workflow process. Objects are something that are affected by the system, something that affects the system, or something that a system needs to be aware of to function properly. For example, objects in a training center include the customer, the training course, and the sales representative. Objects provide a view of the characteristics and behavior of elements in the problem domain that is addressed by the business challenge. In the conceptual design phase, you create usage scenarios that depict the objects in the problem domain.

In addition to objects, usage scenarios depict exceptions. Exceptions are atypical events or alternate task sequences to meet the use case. An example of an exception condition for entering information for a new customer into the training center contact system is when the system is down and the sales representative must use other means to take customer information.

Tips for handling exceptions in a system include:

- Ask “what if” to capture exceptions to the work task or step.
- Determine the relative probability of each exception.
- Discuss how the exception is currently handled and any alternative methods.
- Incorporate handling of high-probability exceptions into the design of the solution.

When you develop usage scenarios, you might identify a task that must be treated as a use case. For example, in the “Register customer for course” use case, you might determine that the task sequence “Process customer payment” is a high-level use case that is part of the workflow process and has several usage scenarios.

Consequently, each usage scenario for “Register customer for course” ends with “Entering course number.” Then you create all relevant usage scenarios for the new “Process customer payment” use case.

You identify the use cases that correspond to the workflow process and then develop usage scenarios that describe the task sequences for each use case. From the usage scenarios, you can determine the current state requirements.

### Why create current state usage scenarios?

After you identify a use case, you can determine the different usage scenarios that can occur for the use case. You then use the information that you gathered from users to describe the different usage scenarios possible for the use case. Creating a usage scenario helps you determine if you gathered the appropriate level of information. Gathering the required amount of information for describing the current state is part of the iterative aspect of gathering and analyzing information.

You can create scenarios for both the current and the future states of the business environment. The current state scenario depicts how business activities are currently conducted; the future state

scenario presents the activities as the business wants them to be. For both states, the scenario emphasizes business processes, information, users, and tasks.

In some situations, full scenarios need be developed only for use cases that are known to have many exceptions or dependencies. This allows the project team to balance costs against the potential benefits. Use cases and scenarios should be developed iteratively, and can be discovered or continued during development work on the initial set of use cases.

### Benefits

Although there are various ways to analyze current work processes, use cases and usage scenarios are especially effective in modeling the process. Creating use cases and scenarios provides the following benefits:

- By measuring productivity levels of the current system, the team can determine whether the new system has achieved usability goals.
- The team can identify the problems in the system and what works in the system.
- The team might discover that the problems perceived by users are different from the actual problems and their causes. The team can then concentrate on the real problems.

### Creating a usage scenario

To create a usage scenario, you need to perform the following tasks:

- Determine the preconditions for the usage scenario, specifying information or conditions that must exist before a scenario can be executed.
- Identify the postconditions for the usage scenario, which identify the work or goal completed during the task sequence.
- Break the activity into discrete steps.
- Identify exceptions that might occur for any step. You might need to develop usage scenarios for these exceptions.
- Identify the requirement that this particular usage scenario addresses, for tracking and traceability.
- Identify the source for this usage scenario, for further discussion and clarification.

## *How to Identify Candidate Business Objects*

Objects are defined as the people or things described in the usage scenarios. Objects form the basis for services, attributes, and relationships.

Figure 5.4 shows the overall process of identifying objects, services, attributes, and relationships in the analysis step of logical design.

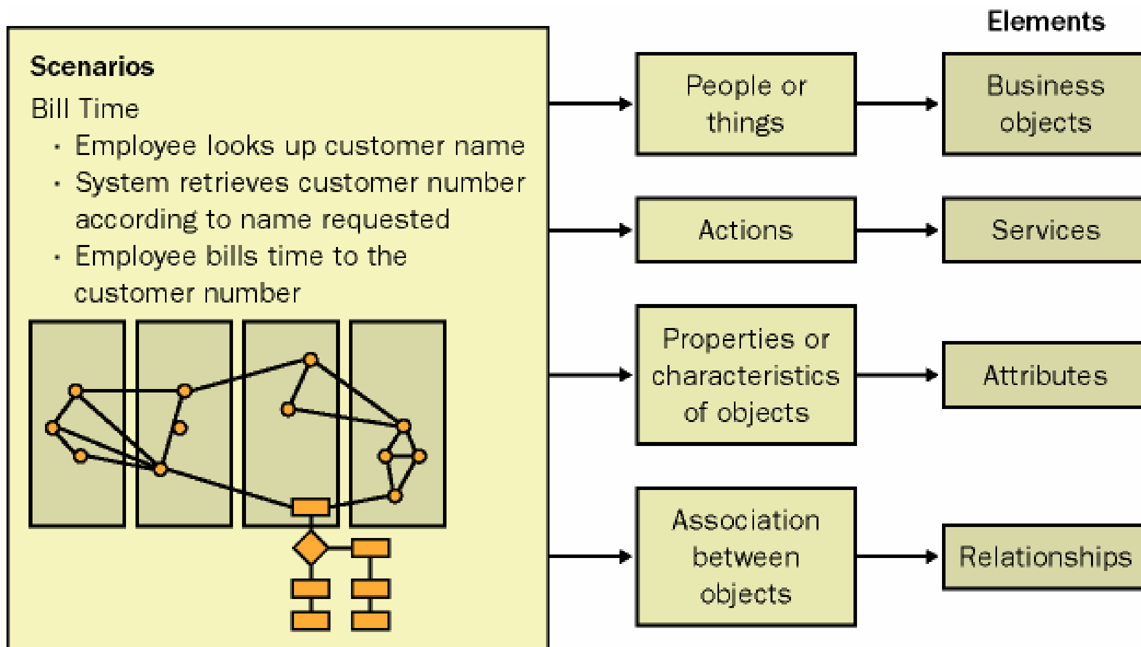



Figure 5-4. Logical design analysis: identifying objects, services, attributes, and relationships

## Identifying objects

You need to identify the business objects, or components, that will provide the functionality for the solution. Look at the usage scenarios you created during the conceptual design process to help you identify these objects. Figure 5.5 illustrates this process. When you have identified an object, you then need to identify the behaviors and attributes of the object in addition to its relationships with other objects.

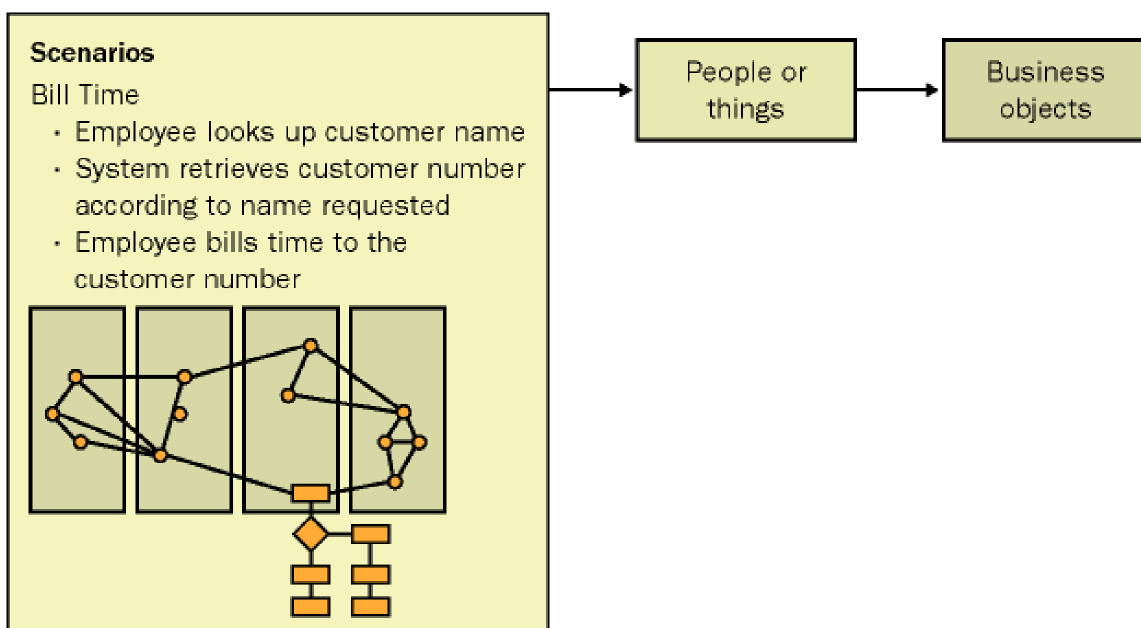



Figure 5-5. Identifying business objects

Some usage scenarios might not explicitly contain objects, even though objects are necessary to perform the required business activities. The objects might be hidden within sentences, depending on how the usage scenario is written. Look for hidden references to structures, systems, devices, things, and events. To identify missing objects, think about the scenario in terms of required information and behavior that is not associated with an object.

### Example of business objects

To clearly understand the concept of business objects, consider the following use cases:

- Employee completes a time sheet by recording the billable hours spent at work.
- Employee creates a contract with the customer.
- Employee reviews prior billings to the customer.
- Employee bills time to the customer number

After studying these use cases, you can identify the following business objects:

- Employee. Performs actions within the system.
- Customer. The recipient of actions performed by the employee.
- Time sheet. The means by which the employee identifies the amount of time spent on a project.
- Contract. The agreement between the employee and the customer.
- Billings. Invoices that the customer has received for work performed.
- Customer number. A means of identifying a specific customer in the system.

## *How to Identify Services*

A service is a specific behavior that a business object must perform. It refers to an operation, a function, or a transformation that can be applied to or implemented by an object. You use services to implement business rules, manipulate data, and access information. A service can perform any activity that can be described by a set of rules.

### Identifying services

To identify services for an object, examine the usage scenario again. To identify a service, determine what the object is supposed to do, the kind of data the object must maintain, and the actions that the object must perform. If an object maintains information, it also performs the operations on the information. Some examples of such actions that an object might perform include:

- Calculate total amount
- Determine the cost of shipping

Figure 5.6 illustrates identifying actions, and therefore services, from a usage scenario.

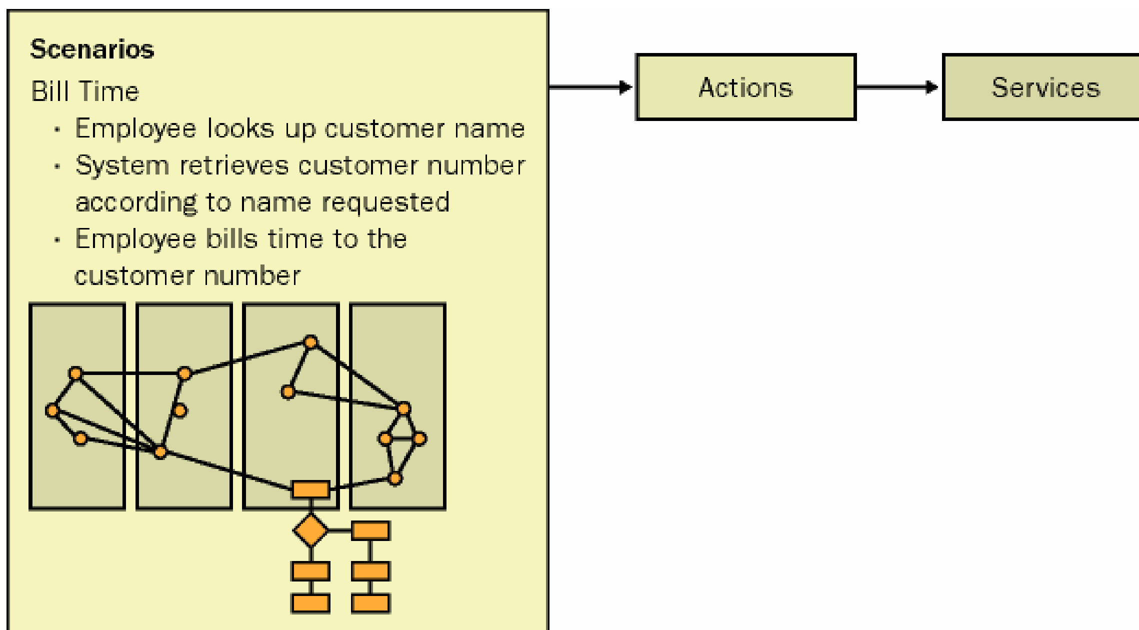



Figure 5-6. Identifying services

Assign the service that you identify to the associated object. This object is either the recipient of the action or is responsible for performing the action in the usage scenario. If it is difficult to identify the correct object for assigning a service, examine all possibilities by a walkthrough of the scenario. A walkthrough of the scenario involves working step by step through every requirement.

State the capabilities and responsibilities of a service in terms as broad as possible. In addition, you need an unambiguous name to identify the service. If you cannot assign a clear name to the service, it indicates that the purpose of the service is not clearly understood and needs more work in the conceptual design.

## Examples of services

Figure 5.7 contains examples of services that can be identified from the sample usage scenario.

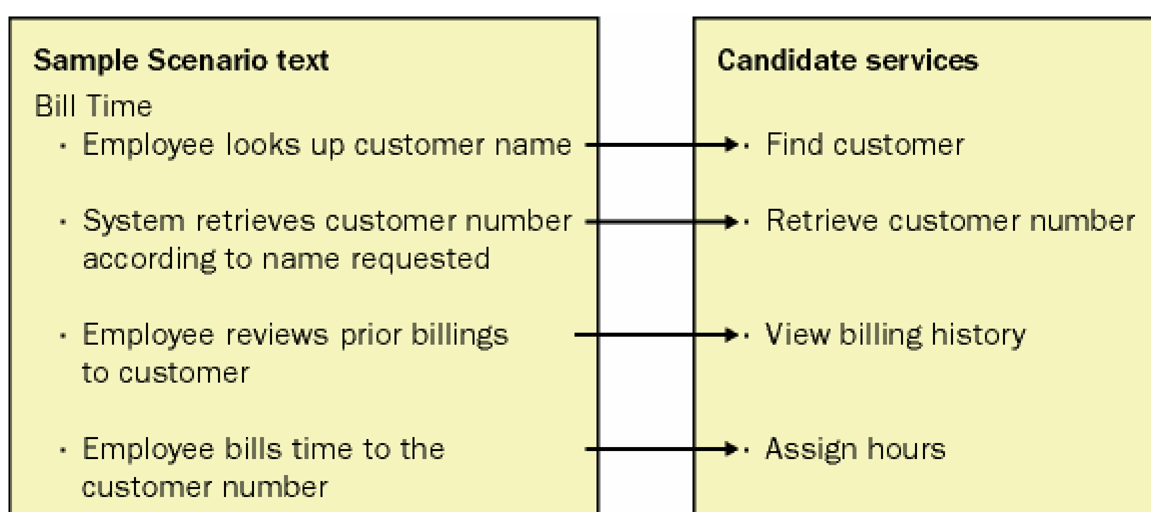



Figure 5-7. Identifying services from a scenario

Consider the following usage scenarios:

- Employee looks up customer name. This usage scenario corresponds to the service find customer for the Employee business object.

- System retrieves customer name. This usage scenario corresponds to the service retrieve customer number for the Customer business object.
- Employee records billable hours on the time sheet. This usage scenario corresponds to the service fill time sheets for the Employee business object.
- Employee reviews prior billings to customer. This usage scenario corresponds to the service view billing history for the Employee business object.
- Employee bills time to the customer name. This usage scenario corresponds to the service assign hours for the Employee business object.

## *How to Identify Attributes*

Attributes of an object are the definitions of data values that the object holds. Attributes are also known as properties. Each instance of an object maintains its own set of values. For example, for the Employee business object, one of the attributes is given name. In a particular instance, the value of the attribute given name is John. In another instance, the value of given name could be Janet. The set of values of an object's attributes at any given time is known as the state of the object.

### Identifying attributes

To identify the attributes for an object, return to the usage scenario. Look for the words or phrases that further identify the object. For example, the length of a bridge, the name of a person, and the brand name and model of a computer indicate attributes.

The actual attributes of an object are often specified in greater detail than the information covered in usage scenarios. To identify the correct attributes, the project team uses its knowledge of the real world and experience in the problem domain. For example, the project team might derive the attribute name from a usage scenario. Based on their knowledge, the team members might modify this attribute to given name and family name. Often this level of detail is done during the physical design process. However, if sufficient information is available to start this process earlier, the team can start during the logical design.

**NOTE**
Although many attributes can be listed for an object, the project team should include only the relevant attributes. For example, the attributes given name and family name can be used for most solutions. However, attributes such as height and weight might only be relevant for, for example, a health care solution. To avoid the risk of omitting important attributes, document all the attributes at this stage of the design process. These attributes can be further refined at a later stage of analysis.

To identify the attributes of an object, the project team should consider each business object and attempt to answer the following questions:

- How is the object described in general and as part of this solution?
- How is the object described in the context of this solution's responsibilities?
- What information does the object contain?
- What information should the object maintain over time?
- What are the states in which the object can exist?

Each attribute is generally identified with an object. You need to clearly label each attribute to avoid confusion with other attributes. In addition, record the structure or type of the attribute, such as text or number.

**TIP**
In many cases, attributes are derived. The computation of such attributes should be recorded as a service of

the object. After compiling the list of attributes, study all the attributes carefully. If some attributes are totally unrelated to the other attributes of a specific object, you might need to create a new business object.

## *How to Identify Relationships*

Relationships illustrate the way in which objects are linked to each other. Unified Modeling Language (UML) defines four types of relationships: dependency, generalizations, associations, and realizations.

- **Dependency**. A relationship between two objects in which a change to one object (independent) can affect the behavior or service of the other object (dependent). Use dependency when you want to show one object using another. For example, a water heater depends on pipes to carry hot water throughout a building. In UML diagrams, dependency between two logical objects is represented by a directional (that is, with an arrow) dashed line.
- **Association**. A structural relationship that describes a connection among objects. *Aggregation* is a special type of association that represents the relationship between a whole and its parts. For example, "Order contains details" is an example of an aggregation relationship. Typically in an aggregation relationship, the whole manages the lifetime of the parts. This form of relationship is called composition. Graphically, association is represented as a solid line. Aggregation is represented by a solid line with a hollow diamond on the end of the line that connects to the whole. *Composition* is represented in UML diagrams by a solid line with a filled-in diamond on the end of the line that connects to the whole.
- **Generalization**. A relationship between a general thing (called the parent) and the specialized or specific thing (called the child). For example, the Manager class is a specific type of the Employee class. A child inherits the properties of its parents, especially their attributes and operations. Generalization means that the child can substitute the parent object anywhere, but the opposite is not true. In UML diagrams, generalization is represented as a solid line with a hollow arrowhead pointing to the parent.
- **Realization**. A relationship between classes, in which one abstract class specifies a contract that another class needs to carry out. When you model, you find a lot of abstractions that represent things in the real world and things in your solution, such as a Customer class in a Web-based ordering system. Each of these abstractions can have multiple instances. In general, the modeling elements that can have instances are called classes. A class has structural and behavioral features. All instances of a class share the same behavior but can have different values for their attributes.

  Realization relationships exist between interfaces and the classes and components that realize these interfaces, and between use cases and collaborations. Graphically, generalization is represented as a cross between generalization and dependency relationship, with dashed lines and hollow arrowhead pointing to the parent.

Figure 5.8 illustrates the graphical representations of the four types of relationships.

FilmClipping
name
+playNow()
+Start()
+stop()
+reset()
Channel
**Dependency**

Person
End1
End2
Company
**Association**

Shape
origin
+move()
+resize()
+display()
Circle
radius: Float
**Generalization**

interface
IRuleAgent
+addRule()
+chargeRule()
+explainAction()
AccountBusinessRules
**Realization**

Figure 5-8. Types of UML relationships

Figure 5.9 illustrates the two types of associations—the aggregation and composition relationships.

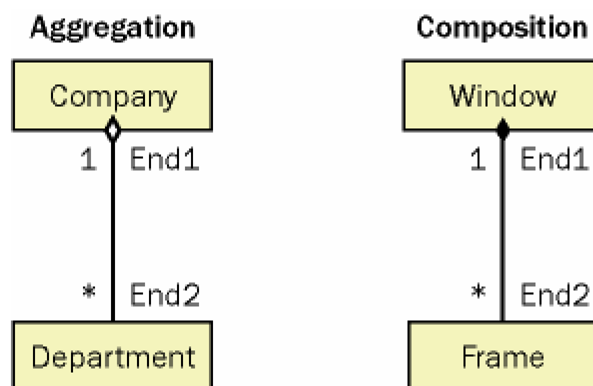



Figure 5-9. Aggregation and composition relationships

## Identifying relationships from use cases

To identify relationships from use cases, look at the usage scenario for information that describes physical location, directed action, communication, or ownership, or that indicates that a condition has been met. The project team reviews the scenario and determines which behaviors are associated with an object, and identifies relationships between two or more objects. Figure 5.10 illustrates how associations between objects indicate relationships.

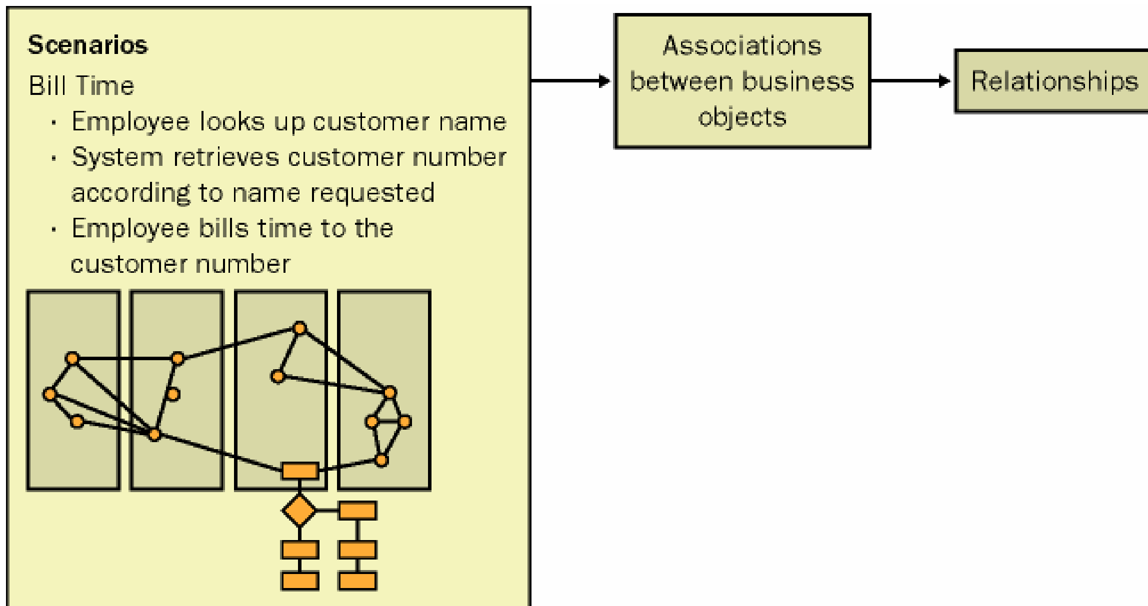



Figure 5-10. Identifying relationships between objects

## Examples of relationships

Relationships represent how the actions of an object or a system affect another object or system, especially if a known state exists between the two. Figure 5.11 illustrates that relationships must exist between objects for a company to produce meaningful work and data.

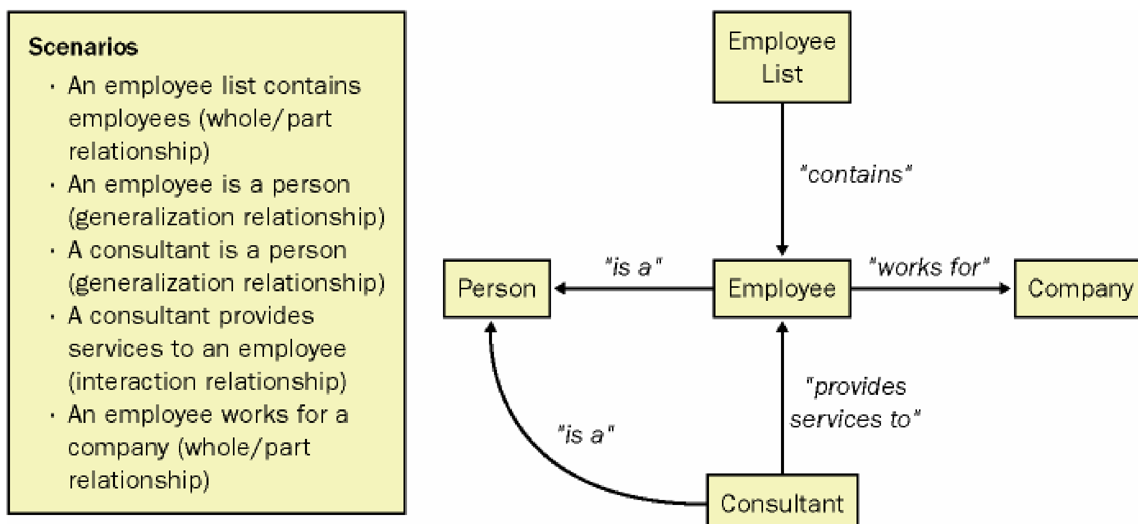



Figure 5-11. Examples of relationships

Some relationships between the objects are:

- The consultant and the customer are related because they are both people.
- The consultant and the customer are related because they interact with each other.

- The consultant performs services for the company, and the company requires the services of the consultant.
- A company has one or more divisions.

Additional relationships might exist in the system. For example, the consultants bring revenue to the company, and the accountants manage the company's money.

## *How to Refine UML Models*

At the end of the logical design, the team has UML models for objects, services, attributes, and relationships in the solution. Typically, the team uses the artifacts that best capture their intent and decisions to manage the complex parts of the project. This includes the following set of deliverables:

- Objects and services inventory
- Class diagrams
- Sequence diagrams
- Activity diagrams
- Component diagrams

During physical design, the team refines these models.

### Objects and services inventory

In the analysis step of the physical design, examine the services inventory for the following tasks:

- Categorizing services based on the MSF services-based application model:
    - User services
    - Business services
    - Data services
    - System services
- Identifying hidden services

  The team tries to identify services that were not apparent during the logical design, such as system services or specific technical services for transforming data. Remember that each hidden service must be synchronized with development goals by validating it against requirements.

### Class diagrams

Class diagrams are used in logical design to represent the static structure of the application object model. During physical design, the team performs the following tasks to refine UML class diagrams:

- Transforming logical objects into class definitions, including their interfaces.
- Identifying objects that were not apparent during logical design, such as services-based objects (also known as common services).
- Consolidating logical objects if necessary.
- Categorizing objects into a services-based model:
    - The logical boundary objects are potential user services.
    - The logical control objects are potential business services.
    - The logical entity objects are potential data services.

There might be exceptions for these recommendations. Therefore, the team must carefully examine all relevant factors before making decisions.

- Refining the methods by focusing on parameters, considering the use of overloads, combining or dividing methods, and identifying how to handle passing values.
- Refining the attributes. Minimize the public attributes as much as possible for applications based on stateless server architecture. During physical design, the team focuses on internal protected attributes, which will be used by derived objects.

Figure 6.5 shows the class diagram for the Order components.

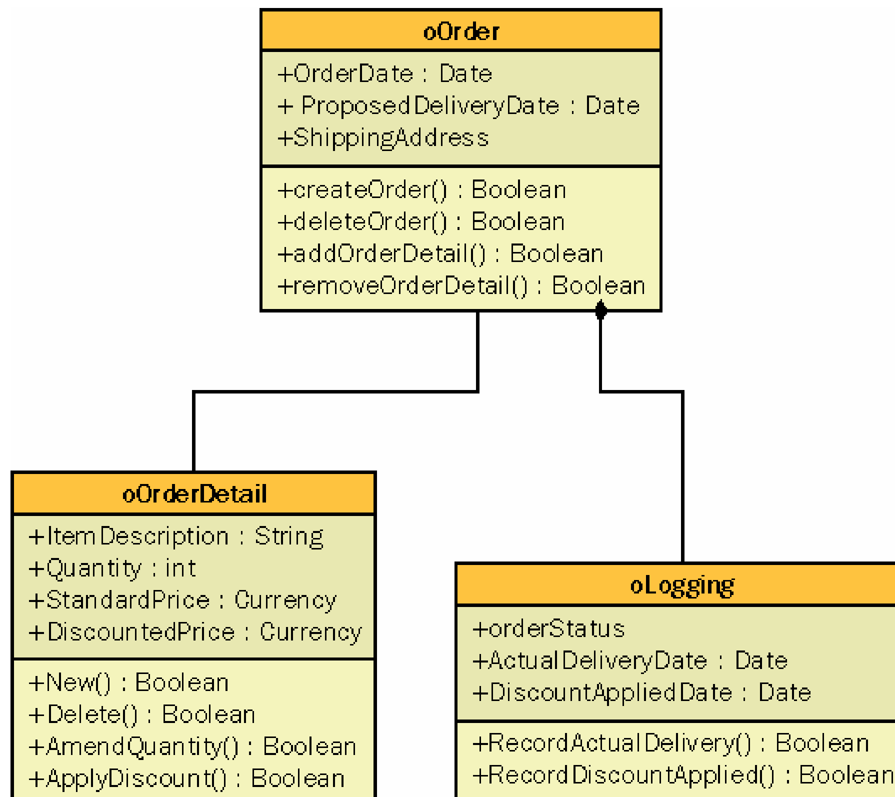
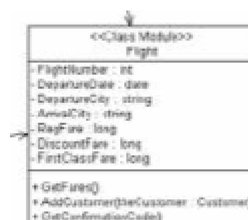



Figure 6-5. Class diagram

## Class diagram artifacts

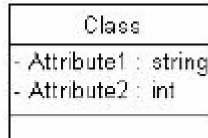
- An ***analysis class*** is an implementation-independent class. The analysis classes are used to document some of the concepts within the system and to create a conceptual view of the system design.

- A ***design class*** is an implementation-specific class within the model. It will correspond to a class in the source code.

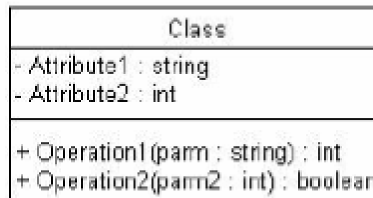


- An ***interface*** is used to expose the public operations of a class without exposing the implementation. An interface contains method signatures, but no implementation.

- An ***attribute*** is a piece of information associated with a class. All objects in a given class will share the same attributes, but each object may have its own attribute values.




- An ***operation*** is a method within the class. In Rose, you can define the operation name, parameters, visibility, return type, and parameter data types.




## Sequence diagrams

Sequence diagrams represent the interaction between objects and the dynamic aspect of the object model. They are usually used to clarify complex class relationships that might not be easily understood by reviewing the static methods and attributes of a group of classes. In the physical design, the team performs the following tasks:

- Updating classes based on the refined physical design classes
- Refining the sequence diagram to include interactions between the classes or services based on physical constraints or technology requirements
- Identifying additional messages (methods) that are triggered by the new physical classes

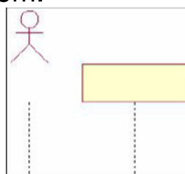
Figure 6.6 shows the sequence diagram for the Product and Catalog objects.

Contact
NewCatalogItem
SQL Database
CatalogItem
«precondition» {isinRole (ProductionClerk)}
Show()
Enter Product Description()
Enter Product Spec()
Enter Product Picture()
Submit()
Get Catalog ID number()
Return Catalog ID number()
Create()
Success()

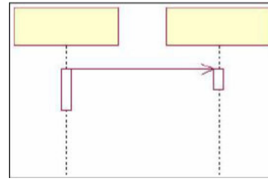
Figure 6-6. Sequence diagram

## Sequence diagram artifacts

- Sequence diagrams are usually created to show the flow of functionality and control throughout the ***objects*** in the system.



- A ***message*** is simply some form of communication between one object or actor and another. Messages can also be *reflexive*, meaning that the object communicates some information to itself.



## Activity diagrams

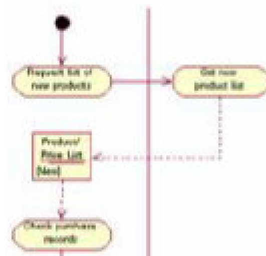
Activity diagrams are used to represent the state transition and flow of an application. You can use activity diagrams instead of sequence diagrams and vice versa. In the physical design, the project refines activity diagrams to:

- Include physical platform and technology requirements.
- Identify potential workflow processes.

An activity diagram that focuses on workflow shows you the people or groups within the workflow, the steps in the process, decision points in the process, areas where steps in the process can occur in parallel, objects affected by the workflow, states of the objects, and transitions between steps in the process. UML contains notation for all of these items.

## Activity diagrams artifacts

- A ***swimlane*** is a vertical section of the diagram that will contain all of the workflow steps that a particular person or group performs. You divide the diagram into many swimlanes, one for each person or group in the process.



- An ***activity*** is a step in the workflow. It can contain actions, which are steps within the activity. The activity is placed in the swimlane of the individual or group that performs the activity.

- A ***transition*** shows how the process moves from one step (activity) to the next. An event triggers the movement from one activity to another. An event can have arguments. A guard condition controls when the transition can or cannot occur; the guard condition must be true for the transition to occur. An action occurs while the process is transitioning from one activity to another. It is typically a quick process that occurs as part of the transition itself. The send target suggests that, as part of the transition, a message is sent to some object. The send target is the object receiving the message. The send event is a message sent to another object. It may have arguments.

- A ***decision point*** in the workflow indicates when the workflow can take two or more different paths.Transition arrows leading from the decision to activities show the different paths that the workflow can follow. Guard conditions on the transitions indicate under which conditions each path will be followed. Guard conditions must be mutually exclusive.

- A ***synchronization*** indicates that two or more steps in the workflow may be completed in parallel. A synchronization bar is used to show where two or more activities may occur simultaneously. These can be very effective in analyzing the efficiency of a workflow; examining the amount of parallel activity can help to optimize a workflow.

## Component diagrams

Component diagrams are used to represent the dependencies between components or component packages. As with sequence and activity diagrams, they are typically used in more complex situations. In the physical design, the project team might create component diagrams to:

- Clarify dependencies between components.
- Further define packaging decisions.

Figure 6.7 illustrates the component diagram for the Order process.

Use Excel on client to manipulate analysis
from SQL server OLAP cubes.
Cube can be on client or server.
Analysis Services
Customer Catalog
Orders
Use groups for divisions within the company
Use roles per titled position
Use WindowsIdentity for employee credentials
Use GenericIdentity for customer
Search (SharePoint Server)
Contact management (Outlook)
Security
Locate Facilities

Figure 6-7. Component diagram

## Component diagram artifacts

- A ***component*** is one of the physical files that make up a system. Source code components will realize many of the various classes contained within the model.