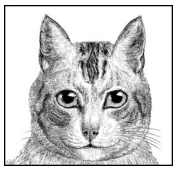
,ch04.17526 Page 71 Wednesday, May 9, 2001 4:02 PM



CHAPTER 4

A Unified Modeling Language Tutorial

This chapter presents a tutorial on the UML. I start with the example briefly mentioned in Chapter 2 and Chapter 3, and examine this problem using the various UML diagrams. This brief tutorial will introduce you to the different diagrams and elements of the UML. The tutorial is not intended to prescribe a particular process for developing models, but provides a learning mechanism by starting with small and intuitive models that are elaborated using the UML. The complete details of each diagram type are found in the reference chapters.

The Unified Modeling Language Diagrams

The UML defines nine types of diagrams: class, object, use case, sequence, collaboration, statechart, activity, component, and deployment diagrams. All of these diagrams are based on the principle that concepts are depicted as symbols and relationships among concepts are depicted as paths (lines) connecting symbols, where both of these types of elements may be named. The example of Chapter 2 and Chapter 3 involved an organization that desires an information system to better utilize employees (resources) on projects.

Use Case Diagrams

Use case diagrams describe the functionality of a system and users of the system. These diagrams contain the following elements:

- *Actors*, which represent users of a system, including human users and other systems.
- *Use cases*, which represent functionality or services provided by a system to users.

Figure 4-1 shows a Resource Manager actor who uses the functionality of the system to manage resources, a Project Manager actor who uses the functionality of the system to manage projects, a System Administrator actor who is responsible for

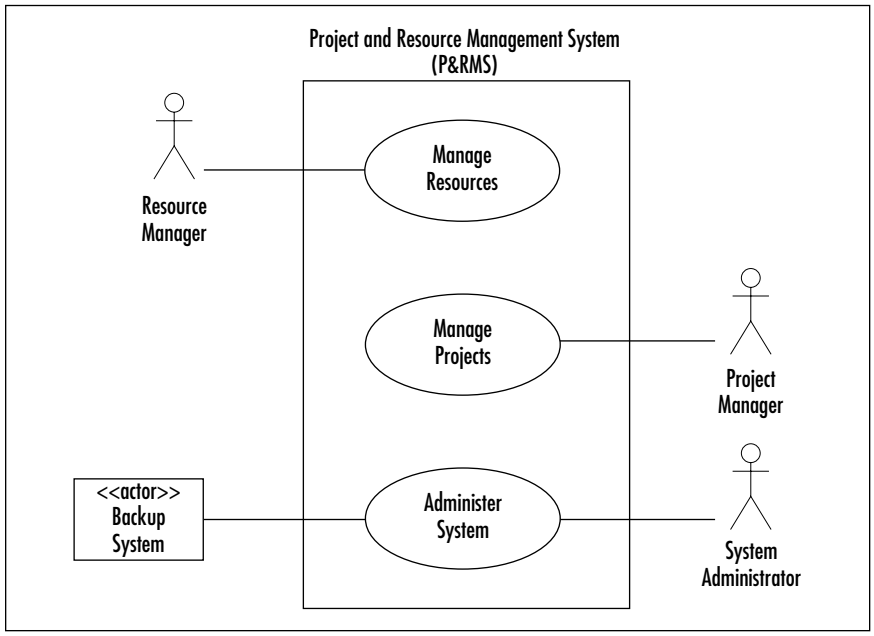


Figure 4-1: High-Level Use Case Diagram

administrative functions of the system, and a Backup System actor that houses backup data for the system. The large rectangle represents the boundary of the system. The name of the system appears above the rectangle. The stick person icon represents an actor who is a human user. The Backup System rectangle icon represents an actor that is another system.

The name of the Backup System actor is preceded by an "<<actor>>" string. This string is called a *stereotype* and indicates that the Backup System is an actor. This stereotype and the stick person icon are equivalent. The stick person icon is usually used to represent humans, whereas the "<<actor>>" string is usually used to represent systems. The ellipses represent use cases. The lines connecting actors and use cases indicate that the actors use or participate in the functionality provided by the use cases. This line is called a *communicates* relationship.

Figure 4-2 elaborates the Manage Resources use case by detailing the functionality a Resource Manager actor expects of the system.

Resource managers may add, remove, or update skills information. Because a skill must be found in the system database before it may be removed or updated, a Find Skill use case is utilized. The arrows from the Remove Skill use case and the Update Skill use case to the Find Skill use case are labeled with a "uses" stereotype to indicate that the Find Skill use case is called or included in the Remove Skill use case and the Update Skill use case. This arrow is called a *uses* relationship.

Resource managers may also add, remove, or update resource information. The Remove Resource and Update Resource use cases utilize a Find Resource use case.

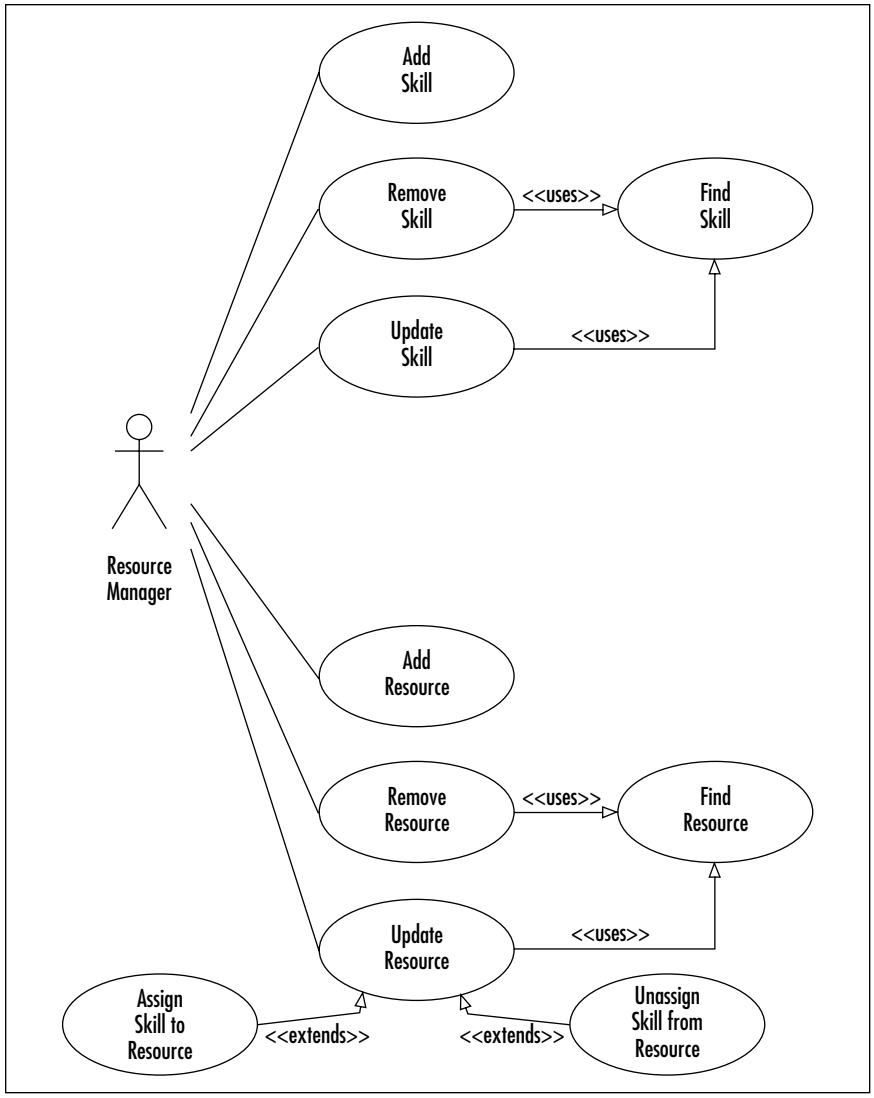


Figure 4-2: Manage Resources Use Case Diagram

Resource managers may assign a skill to a resource or unassign a skill from a resource while updating resource information. An Assign Skill to Resource use case and an Unassign Skill from Resource use case are utilized to depict this functionality. The arrows from these use cases to the Update Resource use case are labeled with an "extends" stereotype to indicate that these use cases are options from the Update Resource use case. This arrow is called an *extends* relationship. Use it when you want to make functionality optional to a use case.

Figure 4-3 elaborates the Manage Projects use case by detailing the functionality a Project Manager actor expects of the system. The elements are similar to Figure 4-2.

Find Project is required by the Remove Project and Update Project use cases, so they are connected by a uses relationship. A number of other use cases are optionally invoked by Update Project, so they are connected to it by an extends relationship.

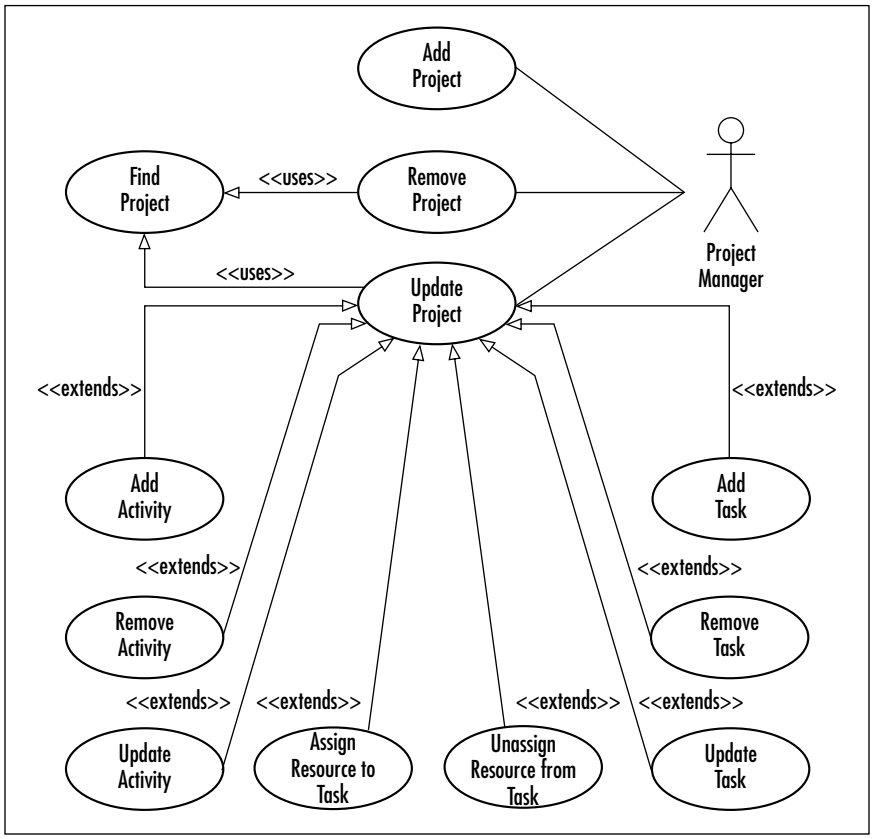


Figure 4-3: Manage Projects Use Case Diagram

Figure 4-4 elaborates the Administer System use case by detailing the functionality a System Administrator actor and Backup System actor expect of the system.

System administrators may start up the system, shut down the system, back up data to the Backup System actor, or restore data from the Backup System actor. Because backing up data is very common before a system is shut down, the Back Up Data use case is a connected option to the Shut Down System use case through an extends relationship. Because restoring data is very common after a system is started up, the Restore Data use case is a connected option to the Start Up System use case through an extends relationship. Because the Backup System actor is involved only in backing up or restoring data and is not directly involved in starting up or shutting down the system, the Backup System actor communicates only with the Restore Data use case and Back Up Data use case. The Back Up Data use case uses the combined functionality of a Back Up Project Data use case and a Back Up Resource Data use case. The Restore Data use case uses the

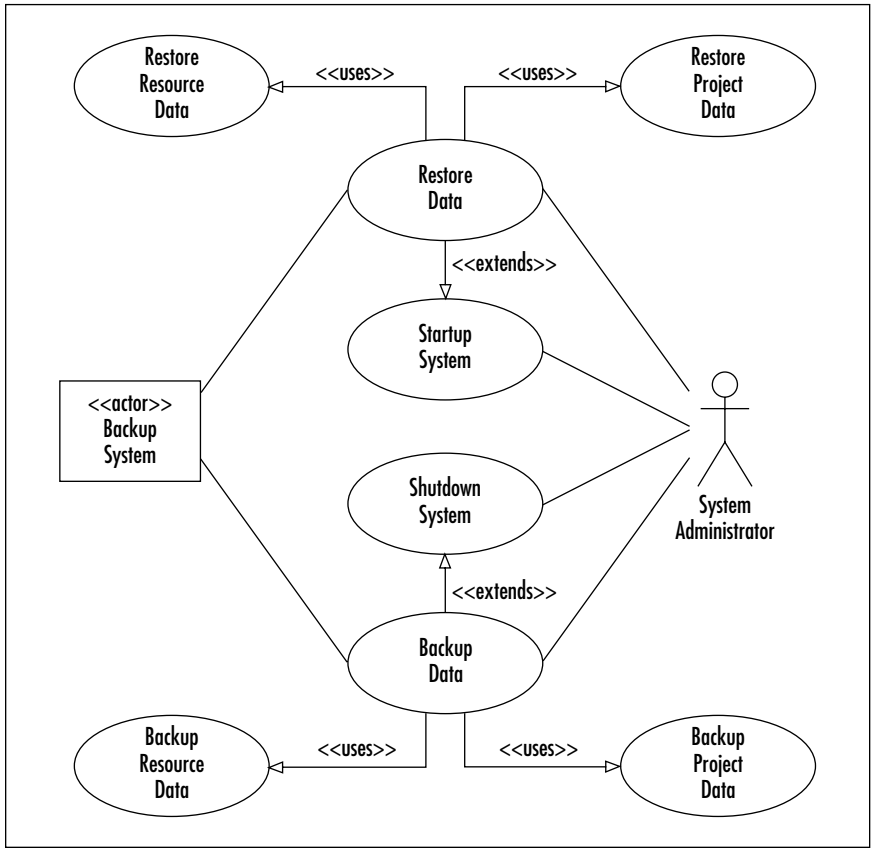


Figure 4-4: Administer System Use Case Diagram

combined functionality of a Restore Project Data use case and a Restore Resource Data use case.

Use cases are usually described in a textual document that accompanies the use case diagram. The combination of these use case diagrams and their supporting documentation is known as a *use case model*.

Use case diagrams are described in greater detail in Chapter 8.

Class Diagrams

Class diagrams describe the static structure of a system, or how it is structured rather than how it behaves. These diagrams contain the following elements:

- *Classes*, which represent entities with common characteristics or features. These features include attributes, operations, and associations.
- *Associations*, which represent relationships that relate two or more other classes where the relationships have common characteristics or features. These features include attributes and operations.

Figure 4-5 shows that the system is concerned with projects, activities, tasks, and resources. The rectangles represent these concepts or classes. The figure also shows that projects are related to activities, activities are related to tasks, and tasks are related to resources. The paths represent relationships or associations. Each class must have a name within the rectangle, and each association may have a name attached.

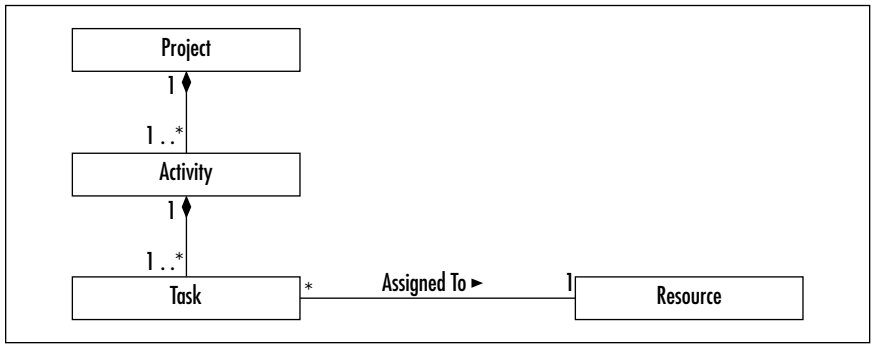


Figure 4-5: High-Level Project Class Diagram

Projects, activities, and tasks are all efforts of work. A project consists of activities, and activities consist of tasks. A project does not have any hours associated with it, but activities and tasks have hours indicating the duration required to do the work. Activities have associated deliverables or work products, whereas tasks have associated resources for doing the work. The filled diamond attached to the Project class of the association between the Project class and the Activity class indicates that the project contains activities and that if the project is removed, so are its associated activities. A similar relationship holds between the Activity class and the Task class.

Thus, the filled diamond is used to indicate a *composition* relationship. If activities could exist without being associated with a project or a project could be removed without having to remove its activities, the diamond may be hollow. Such a hollow diamond is used to indicate an *aggregation* relationship rather than a composition relationship.

Projects contain one or more activities. The "1..*" string attached to the Activity class of the association between the Project class and the Activity class indicates that one or more activities are involved in the relationship. This is called the *multiplicity* of the Activity class within the relationship. The multiplicity specifies how many objects of the class are associated with a single object of the other class in the association. An activity may be associated with only one project.

The "1" string attached to the Project class of the association between the Project class and the Activity class indicates that only one project is involved in the relationship. The multiplicity indicators on this association specify that each project must have one or more activities, and each activity may only belong to a single project. The association between the Activity class and the Task class specifies that each activity must have one or more tasks, and each task may only belong to a single activity.

Resources may be assigned zero or more tasks. The "*" string attached to the Task class of the association between the Task class and the Resource class indicates that zero or more tasks are involved in the relationship. Each task may only be assigned to one resource. Note the difference between the "1..*" multiplicity, which requires at least one instance, and the "*" multiplicity, which permits zero as an option. The "1" string attached to the Resource class of the association between the Task class and the Resource class indicates that only one resource is involved in the relationship.

The association name has a small arrow indicating the direction in which to read the association name. If no arrow is attached, associations are read from left to right and top to bottom. In this case, the arrow is not required. The association between the Task class and the Resource class specifies that zero or more tasks are associated with a single resource, and each resource may be associated with zero or more tasks.

Figure 4-6 shows that zero or more resources may be associated with zero or more skills. Thus, any number of resources may have the same skill. Because we want to maintain information about the relationship between a resource and a skill, a Resource-Skill class is used. The dashed line connecting the Resource-Skill class to the actual path indicates that the class maintains information about the relationship. This class is called an *association class*. Association classes are used to maintain information about the relationship rather than the objects involved in the relationship.

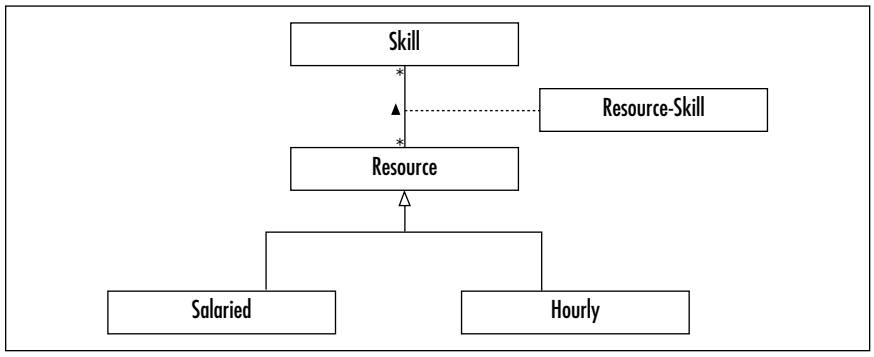


Figure 4-6: High-Level Resource Class Diagram

There are two kinds of resources, salaried and hourly. The arrows from the Salaried class and the Hourly class to the Resource class indicate that the Salaried class and the Hourly class inherit or receive all the characteristics of the Resource class. This also indicates that whenever a Resource class object is used, a Salaried class object or Hourly class object may be used in its place. The Resource class is reused by the Salaried class and Hourly class. This arrow is called a *generalization* relationship. In object-oriented languages, the term *inheritance* is often used instead of generalization relationship, and the particular relationship between the Resource class and the others would be called an *is-a-kind-of* inheritance.

Figure 4-7 details Figure 4-5 by defining what constitutes projects, activities, and tasks. Projects have a name, description, and a start date. The compartment below

the name represents the *attributes* of the class. Attributes are *structural features* of the class, that is, information about the class. The name attribute will hold a string value, the description attribute will hold a string value, and the start date attribute will hold a date value. The attributes are all inaccessible outside the class, as indicated by the preceding "-" visibility character. This is known as *private* visibility.

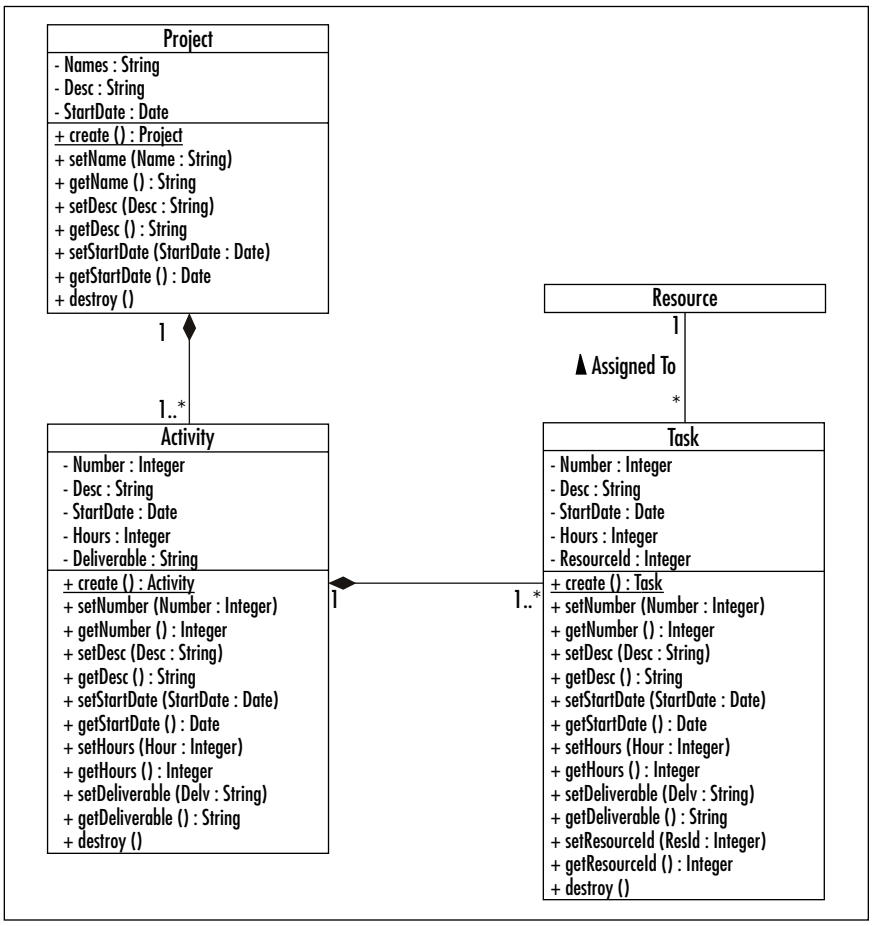


Figure 4-7: Detailed Project Class Diagram

Projects must provide services for setting and getting the values of their attributes. The compartment below the attribute compartment represents the *operations* of the class. Operations are *behavioral features* of the class, that is, the services (often known as *methods* in object-oriented languages) provided by the class. Thus, the setName operation requires a string parameter to set the value of the Name attribute, and the getName operation returns the value of the Name attribute. There are operations for setting and getting the values of all of the attributes.

The Project class must also provide services for creating and destroying objects of the class. The "create" operation is used to create an object of the class. This oper-

ation requires no arguments and returns a Project. This operation is known as a *constructor* operation. Because the operation is applied to the class rather than an object of the class, it is underlined. The "destroy" operation is used to destroy objects of the class. This operation requires no parameters and does not return anything. This operation is know as a *destructor* operation. The operation may be applied to an object of the Project class just like the other operations. All of these operations are accessible outside the class, as indicated by the preceding "+" visibility character. This is known as *public* visibility.

When we want to make an attribute or operation inaccessible outside a class, but accessible to classes with a generalization relationship to the class, we use *protected* visibility. Protected visibility is indicated by a preceding "#" character.

Activities have a number to describe the order in which they are done within a project, a description, a start date, the number of hours they require, and the deliverable product that results from the activity. The Activity class provides appropriate operations for setting and getting the values of these attributes, and operations for creating and destroying objects of the class.

Tasks have a number to describe the order in which they are done within an activity, a description, a start date, the number of hours they require, and an identification number of the resource who is responsible for the task. The Task class provides appropriate operations for setting and getting the values of these attributes, and operations for creating and destroying objects of the class.

Because activities and tasks are very similar, with the only distinction being that activities have an associated deliverable and tasks have an associated resource, a Work Effort class is used as a base from which to define activities and tasks. Figure 4-8 shows that a Work Effort class represents everything common to activities and tasks, whereas the Activity class and the Task class have a generalization relationship to the Work Effort class. The Work Effort class is reused to define the Activity class and the Task class

Figure 4-9, which is part of this chapter's hypothetical system, details Figure 4-6 by defining what constitutes skills and resources. Skills have two attributes: a name and a description. The Skill class provides appropriate operations for setting and getting the values of these attributes, and operations for creating and destroying objects of the class.

Resources have a name. The Resource class provides appropriate operations for setting and getting the values of this attribute, and operations for creating and destroying objects of the class. Salaried resources have an associated salary, and hourly resources have an associated hourly rate. The Salaried class and Hourly class also provide similar operations.

The information maintained about the relationship between a resource and a skill includes the proficiency of the resource and the years experience the resource has with the skill. The Resource-Skill class has a Prof attribute that may have the values of Expert or Novice, and a YearsExpr attribute for representing the number of years of experience. The class also provides appropriate operations for setting and getting the values of these attributes, and operations for creating and destroying objects of the class.

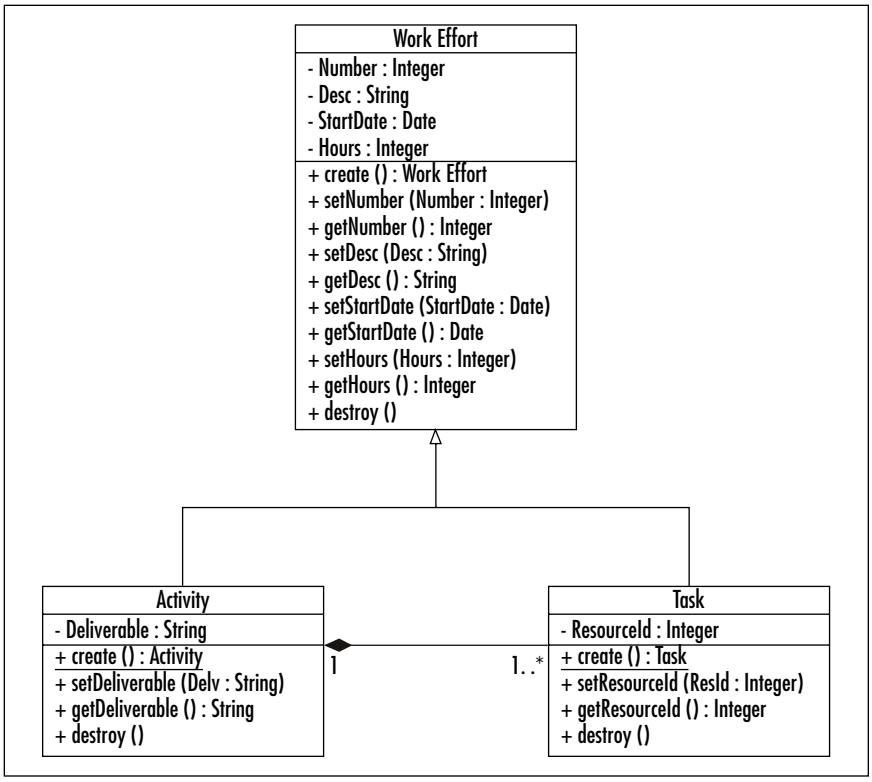


Figure 4-8: Detailed Activities and Tasks Class Diagram

Figure 4-10 details Figure 4-9 by defining how the relationship between skills and resources will be implemented. The Resource-Skill class must provide the capability to associate any number of resources with any number of skills.

We introduce the Identifiable class as an abstract class that simply provides an identification integer. The class is specified as abstract by including the constraint string "abstract" in curly braces following the class name. *Abstract classes* cannot have instances; they merely provide attributes and services shared by the classes that inherit them.

The Skill class and the Resource class both inherit the characteristics of the Identifiable class; that is, objects of these classes are now identifiable via their Id attribute. The Resource-Skill class stores the Id attributes of the associated Resource class object and Skill class object pairs. The SkillId attribute of the Resource-Skill class stores the Id attribute of the associated Skill class object, and the ResourceId attribute of the Resource-Skill class stores the Id attribute of the associated Resource class object. Because we now know enough detail about how the Resource-Skill class maintains information about the association, the association may be refined.

To determine if a particular resource has a particular skill, the Resource class object and a SkillId attribute value of the Resource-Skill class are used to find the

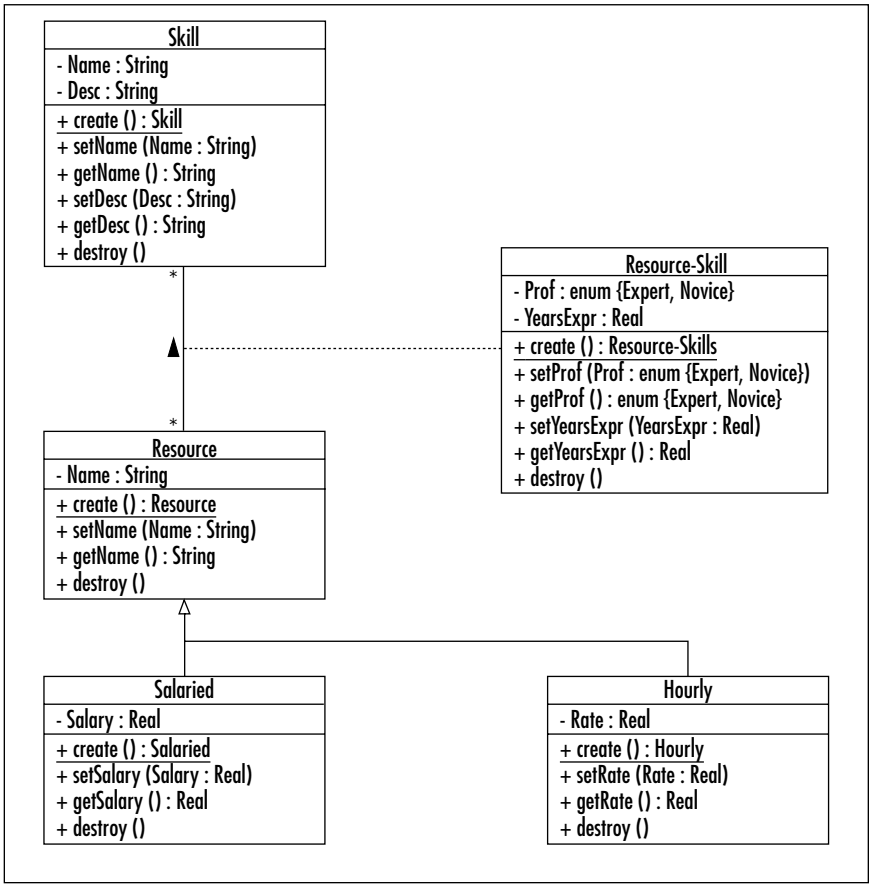


Figure 4-9: Detailed Resource Class Diagram

associated Skill class object. The SkillId attribute of the Resource-Skill class is attached, as a small rectangle, to the Resource class to indicate that a Resource class object and a SkillId value are used to partition or select zero or one Skill class object. The SkillId attribute attached to the Resource class is called a *qualifier*. The multiplicity on the association is changed to "0..1" to specify that pairing a Resource class object and a qualifier value may or may not select a Skill class object. That is, not every Resource class object must be associated with a Skill class object. There are situations where a particular resource simply does not have an associated skill.

This collection of class diagrams is used to understand the system's structure. They are elaborated with more and more detail until they can be virtually implemented. Classes, associations, attributes, and operations are usually described in a textual document that accompanies the class diagrams. The combination of these class diagrams and their supporting documentation is known as a *class model*. This model is often referred to as an *object model* because it represents the possible configurations of objects within the system.

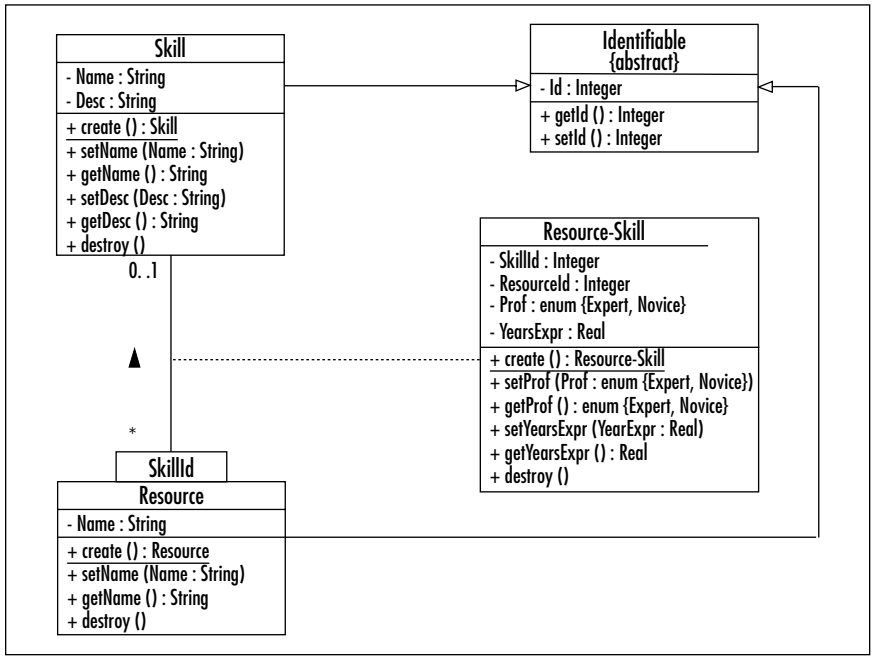


Figure 4-10: Detailed Skills and Resources Class Diagram

Class diagrams are described in greater detail in Chapter 7.

Object Diagrams

Object diagrams describe the static structure of a system at a particular time. Whereas a class model describes all possible situations, an object model describes a particular situation. Object diagrams contain the following elements:

- *Objects*, which represent particular entities. These are instances of classes.
- *Links*, which represent particular relationships between objects. These are instances of associations.

Figure 4-11 shows a project named "Human Resource System Development" that consists of three activities. The first activity is named "Scope" and consists of two tasks. The first task is named "Review," and the second task is named "Scope." As you can see, the same name can be assigned to two different elements (here, an activity and a task) without confusion in the UML.

The second activity is named "AnalysisDesign" and consists of two tasks. The first task is named "Analysis," and the second task is named "Design." The third activity is named "Implementation" and consists of three tasks. The first task is named "Code," the second task is named "Test," and the third task is named "Deploy." The rectangles in this diagram represent objects, and the paths represents links. The object names are underlined and indicate the class of the object. Each object has actual values for their attributes.

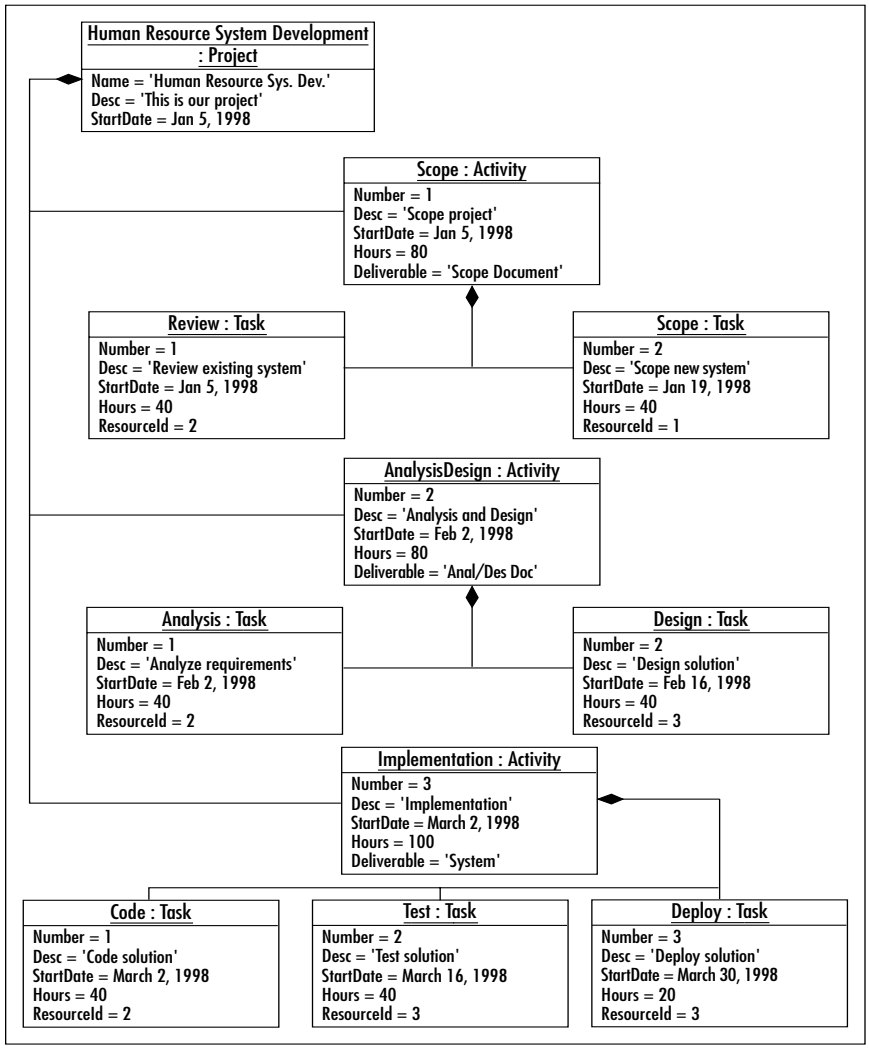


Figure 4-11: Project Object Diagram

All the multiplicity rules identified in the class diagram are adhered to; that is, each task belongs to only one activity, each activity contains one or more tasks, each activity belongs to one project, and each project contains one or more activities.

Figure 4-12 shows resources and their skills. To understand the connections, follow the lines from each Resource class object to the related Skill class object. Each Resource class object represents a person in this figure, and the attributes of each person's skills are shown in the Resource-Skill class objects connected by a dotted line to the relationships to Skill class objects.

For instance, Andy has three years' experience in gathering requirements, three years' experience in doing analysis and design, and seven years' experience in

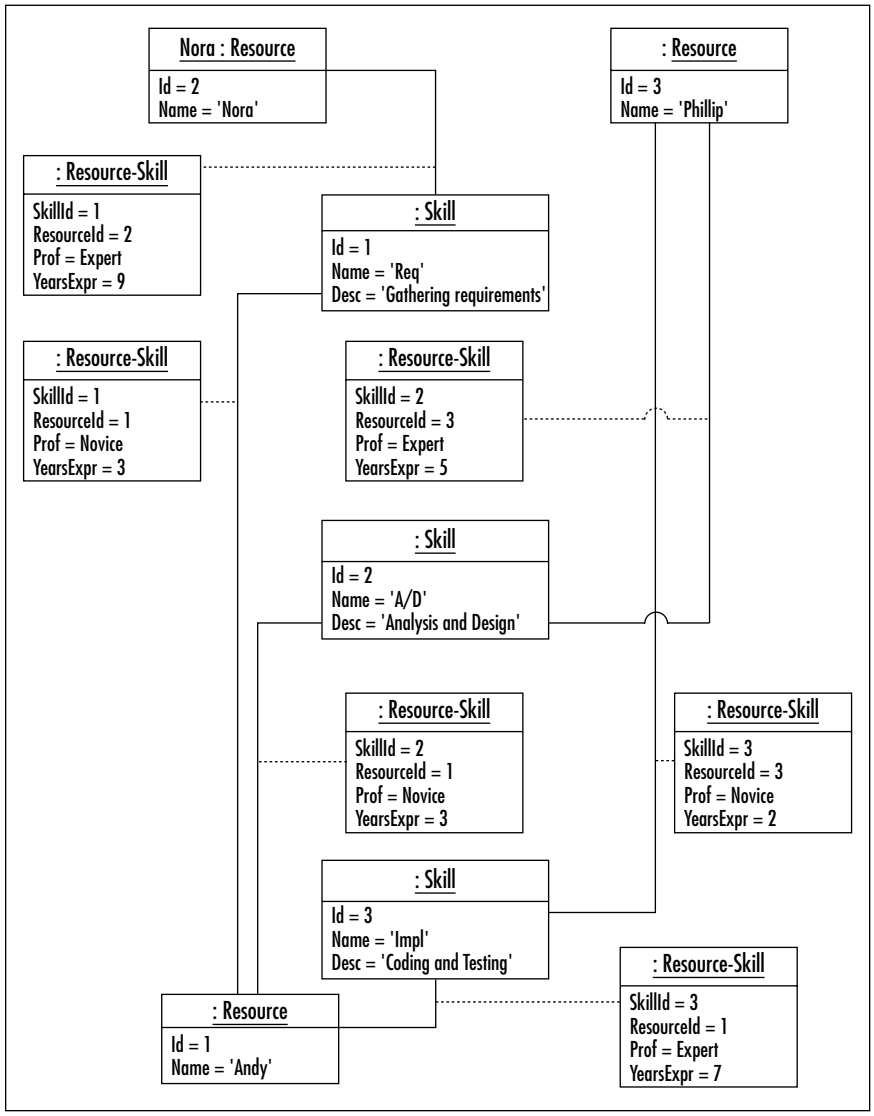


Figure 4-12: Resource Object Diagram

coding and testing. Andy is represented as an object. Notice that the object has no name, but is identified by its attributes. Such an object is called an *anonymous object*. This emphasizes the idea that an object need not be identified by its name, but instead may be identified by its characteristics (here, Id and Name). A name is useful if we want to refer to an object independent of the values of its characteristics. Each of Andy's skills are objects of the Skill class, and each relationship between the object representing Andy and his skills has an associated Resource-Skill class object.

Phillip has two years' experience in coding and testing, and five years' experience in doing analysis and design. Notice that the Skill class objects are shared between the Resource class objects, but the information about the relationships is actually maintained within Resource-Skill class objects.

Nora has nine years' experience in gathering requirements. Notice that the Resource class object representing Nora is named "Nora." This is optional.

Object diagrams are often used to validate the class model of a system. If we discover an object diagram that invalidates the class model, we must either change the class model or determine that the object diagram is invalid. For example, Figure 4-13 shows a project named "Payroll System Development" in which the Task class object named "Design" belongs to the Activity class object named "AnalysisDesign" and the Activity class object named "Implementation." This invalidates the rule, within the class model, that a task may only belong to one activity. We either must change the class model to support this possibility or determine that the object diagram is invalid. In this case, the object diagram is ill formed, and the rule is valid.

Object diagrams are described in greater detail in Chapter 7.

Sequence Diagrams

Sequence diagrams describe interactions among classes. These interactions are modeled as exchanges of messages. These diagrams focus on classes and the messages they exchange to accomplish some desired behavior. Sequence diagrams are a type of interaction diagram. Sequence diagrams contain the following elements:

- *Class roles*, which represent roles that objects may play within the interaction.
- *Lifelines*, which represent the existence of an object over a period of time.
- *Activations*, which represent the time during which an object is performing an operation.
- *Messages*, which represent communication between objects.

Figure 4-14 elaborates the Assign Skill to Resource use case. It shows how a resource manager uses the system to assign a skill to a resource, and how the classes within the system work together to provide this functionality. The objects at the top of the diagram represent class roles. They are named just like objects since they represent the objects that participate in the interaction. The dashed lines that extend from each object represent lifelines. The thin rectangles placed on lifelines represent activations. The horizontal arrows between lifelines indicate the messages exchanged between objects. The horizontal arrows are labeled with the message that is sent between the class roles. A message triggers an operation in the receiving object.

A resource manager will use the resource manager window, which is a user interface, to find a resource, find a skill, and assign the skill to the resource. The resource manager window will find a resource using a Resource class object, and a skill using a Skill class object. The window will assign a skill to a resource if the skill is not already assigned to the resource. This condition is called a *guard condition*

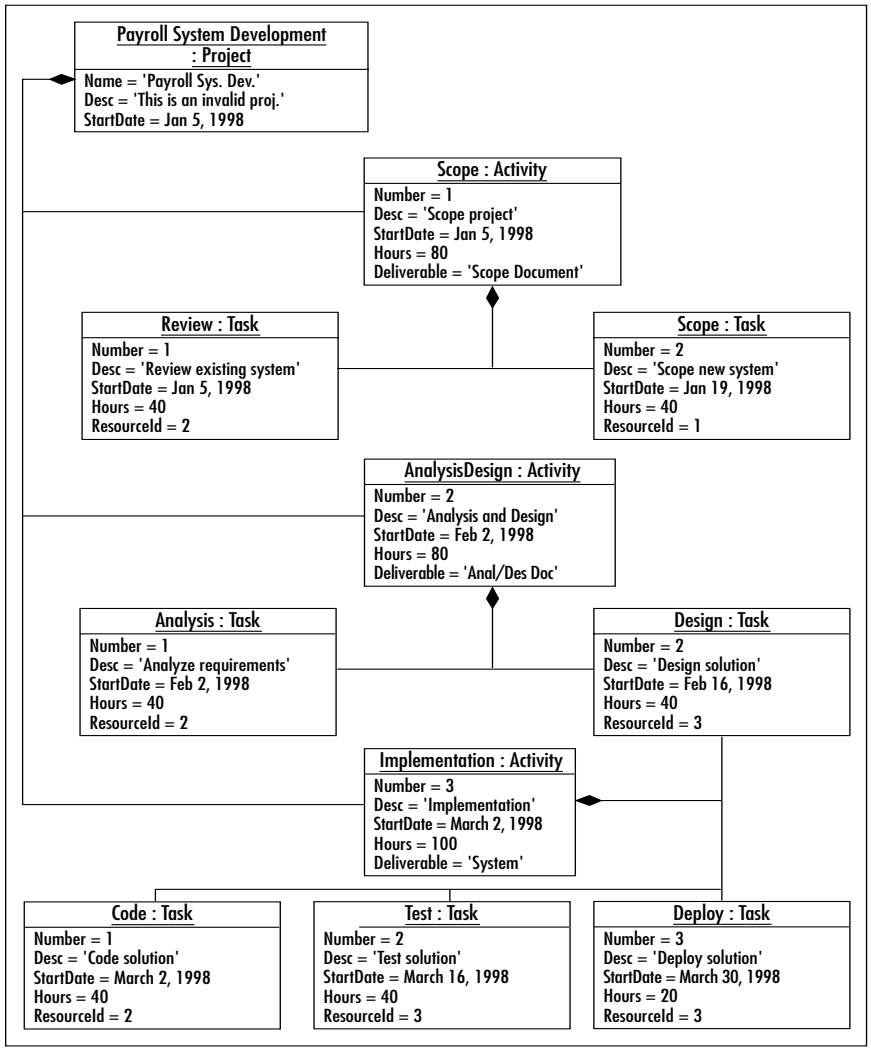


Figure 4-13: Invalid Project Object Diagram

and is indicated within square brackets on the message originating from the window lifeline to the Resource-Skill class role.

Figure 4-15 shows how a resource manger assigns the "A/D" skill to Nora. The class roles represent actual objects, which are anonymous in this case. The messages are replaced with actual messages sent between the objects. Notice that the condition on the message originating from the window lifeline to the Resource-Skill class role is satisfied and the message is sent. This diagram is called a *scenario* since it shows one possible use of the sequence diagram in Figure 4-14.

Figure 4-16 shows a scenario in which a resource manager assigns the "Impl" skill to Andy. The message originating from the window lifeline to the Resource-Skill

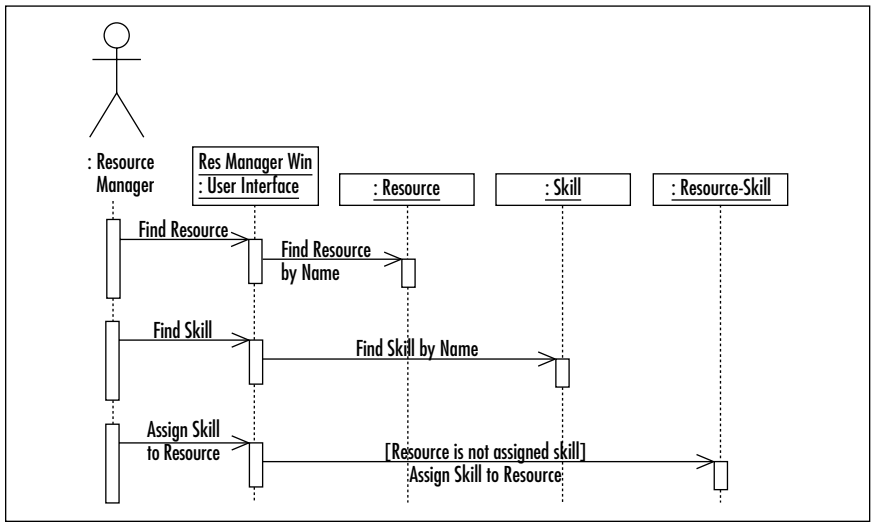


Figure 4-14: Assign Skill to Resource Sequence Diagram

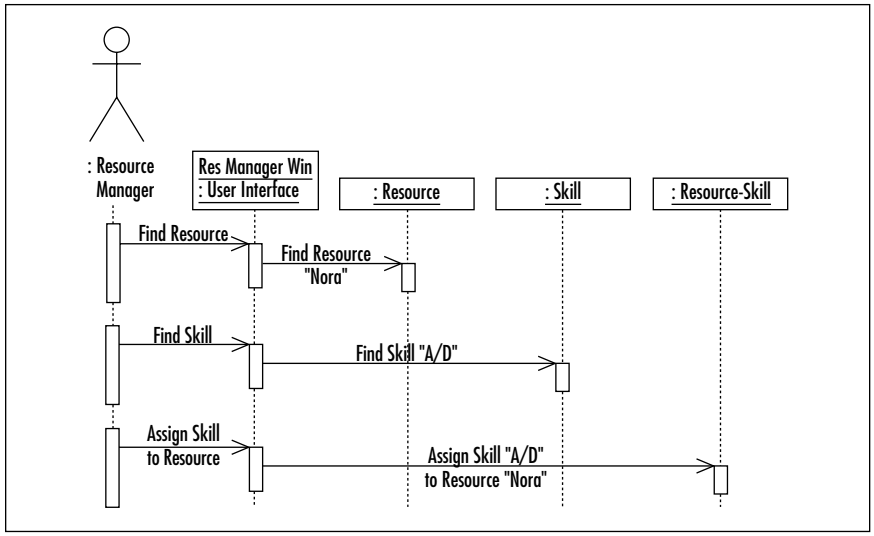


Figure 4-15: Assign the A/D Skill to Nora Scenario (Sequence Diagram)

class role is not shown since this skill is already assigned to Andy and the guard condition is not satisfied. This scenario and the scenario of Figure 4-15 are both satisfied by the sequence diagram of Figure 4-14. Just as object diagrams may be used to validate class models, scenarios validate sequence diagrams.

Figure 4-17 elaborates the Remove Project use case. It shows how a project manager uses the system to remove a project from the database, and how the classes within the system work together to provide this functionality. The rectangles enclosing a

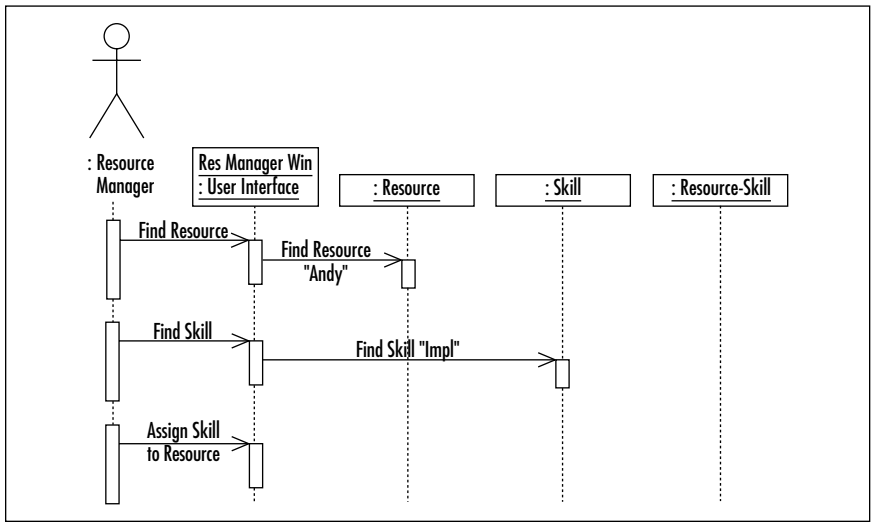


Figure 4-16: Assign the Impl Skill to Andy Scenario (Sequence Diagram)

set of messages represent *repetition,* or a loop. The condition for exiting a loop is shown at the bottom of the rectangle ("No More Tasks" or "No More Activities").

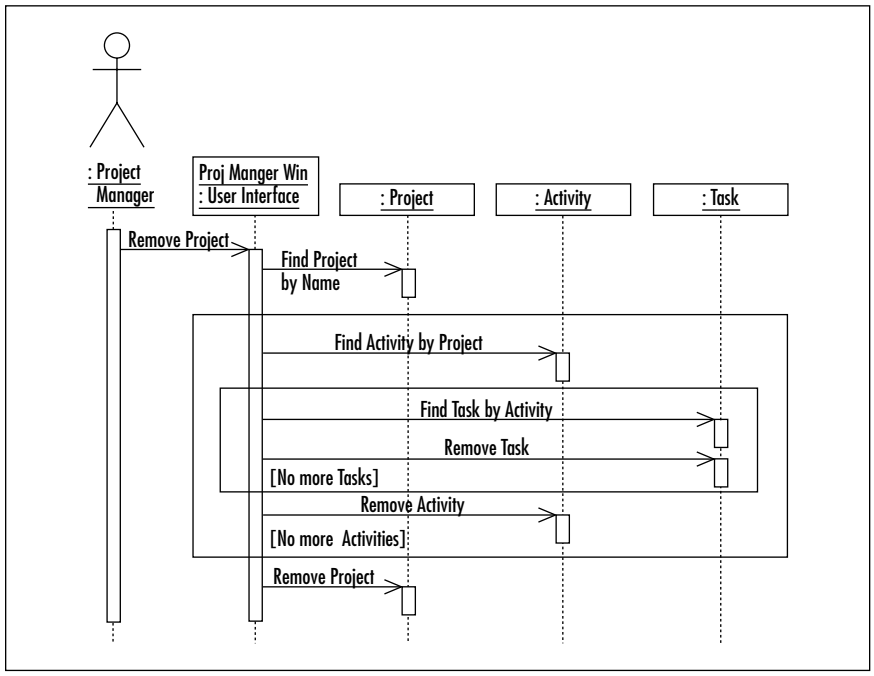


Figure 4-17: Remove Project Sequence Diagram (Version 1)

A project manager will use the project manager window, which is a user interface, to specify a project to remove. Then the project manager window will find the project using a Project class object. Once the project is found, the project manager window will repeatedly find the activities within the project, remove their tasks, and remove the activities. The window uses an Activity class object to find and remove activities, and a Task class object to find and remove tasks. The repetition continues until all the tasks and activities are removed. The window completes processing by removing the project entry from the database.

Figure 4-18 is equivalent to Figure 4-17. The only difference is in how the repetition is depicted. In this case, the loops are not represented using rectangles but using a dashed line with an attached constraint string. The constraint string specifies the extent of the loop, and the dashed line specifies those messages that are sent within the loop. The two diagrams are simply two alternative conventions for representing the same loops.

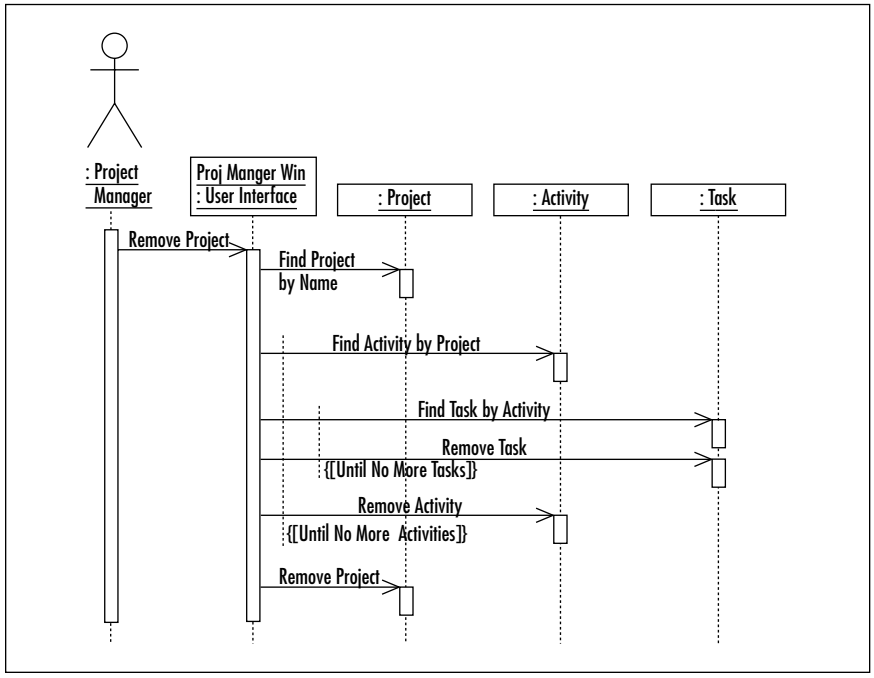


Figure 4-18: Remove Project Sequence Diagram (Version 2)

Figure 4-19 shows a scenario in which a project manager removes the "Human Resource System Development" project, modeled by Figure 4-11, from the system. Notice that all the loops within the diagram are removed, and the messages sent within the loops are shown. Again, this scenario is used to validate the sequence diagrams of Figures 4-17 and 4-18.

Figure 4-20 elaborates the Backup System use case. It shows how a system administrator uses the system to back up the database, and how the Backup System actor is used within this interaction. Because the Backup System actor is actually another system and not part of the original system, the two are concurrently active and

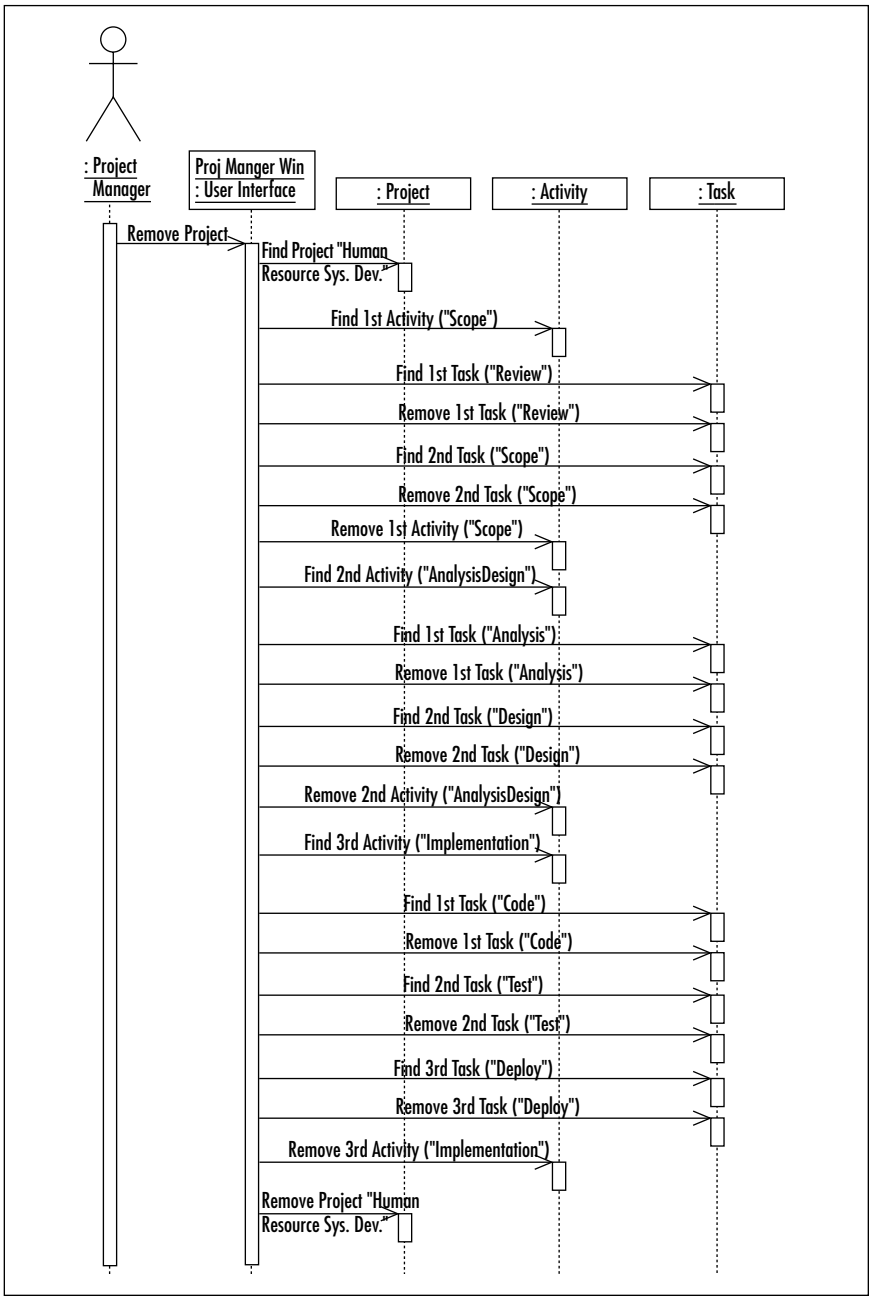


Figure 4-19: Remove Project Scenario (Sequence Diagram)

communicate in an asynchronous manner. Both systems are active, as indicated by the "active" constraint attached to their names. An active object or system may

request services from other objects. Furthermore, two active objects may communicate asynchronously. The half-arrowhead messages represent *asynchronous messages* sent between objects. An asynchronous message indicates that the sender does not wait for the receiver to return control before continuing. *Synchronous messages* (represented by full arrowheads) indicate that the sender actually passes control to the receiver and waits until control is returned before continuing. The diagram shows how the systems will send messages back and forth asynchronously to coordinate the backing up of all the projects, their activities, and their tasks.

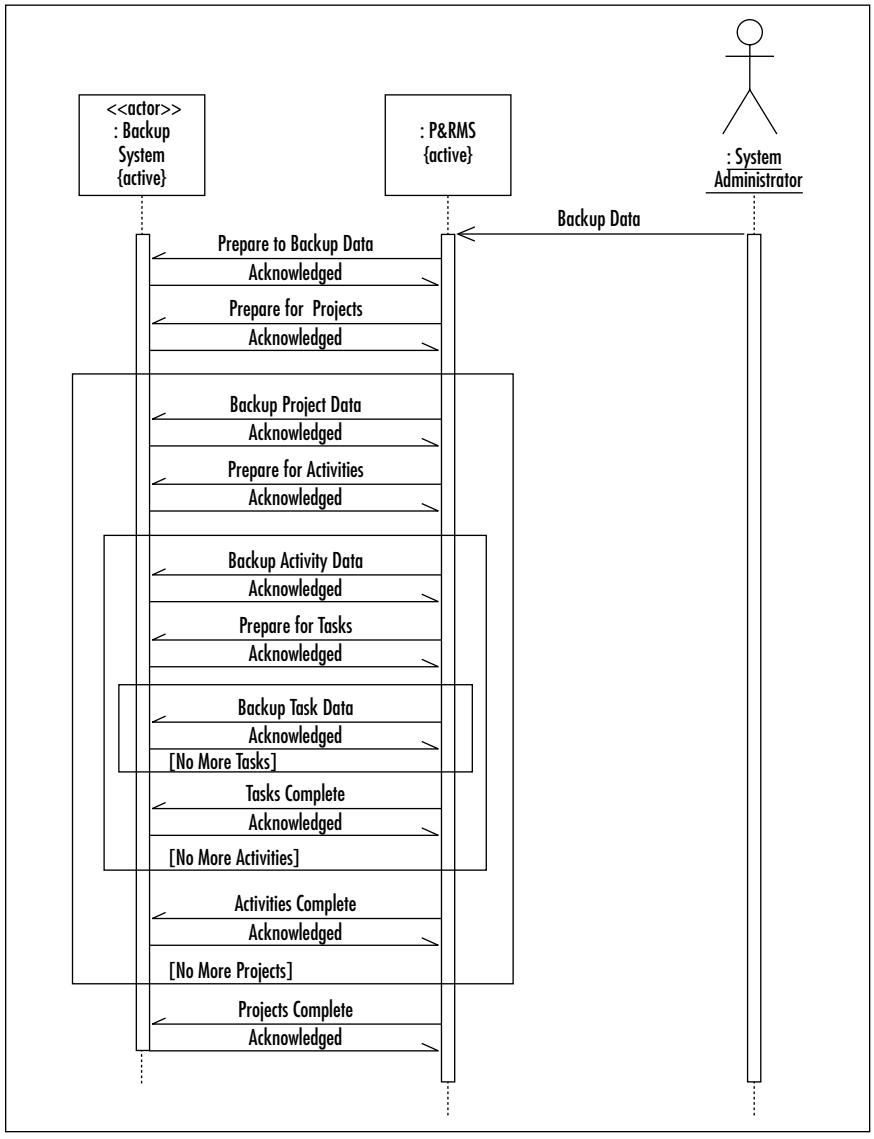


Figure 4-20: Back Up Project Data Sequence Diagram (Version 1)

Figure 4-21 is equivalent to Figure 4-20. The only difference is in how the repetition is depicted, using dashed lines rather than rectangles.

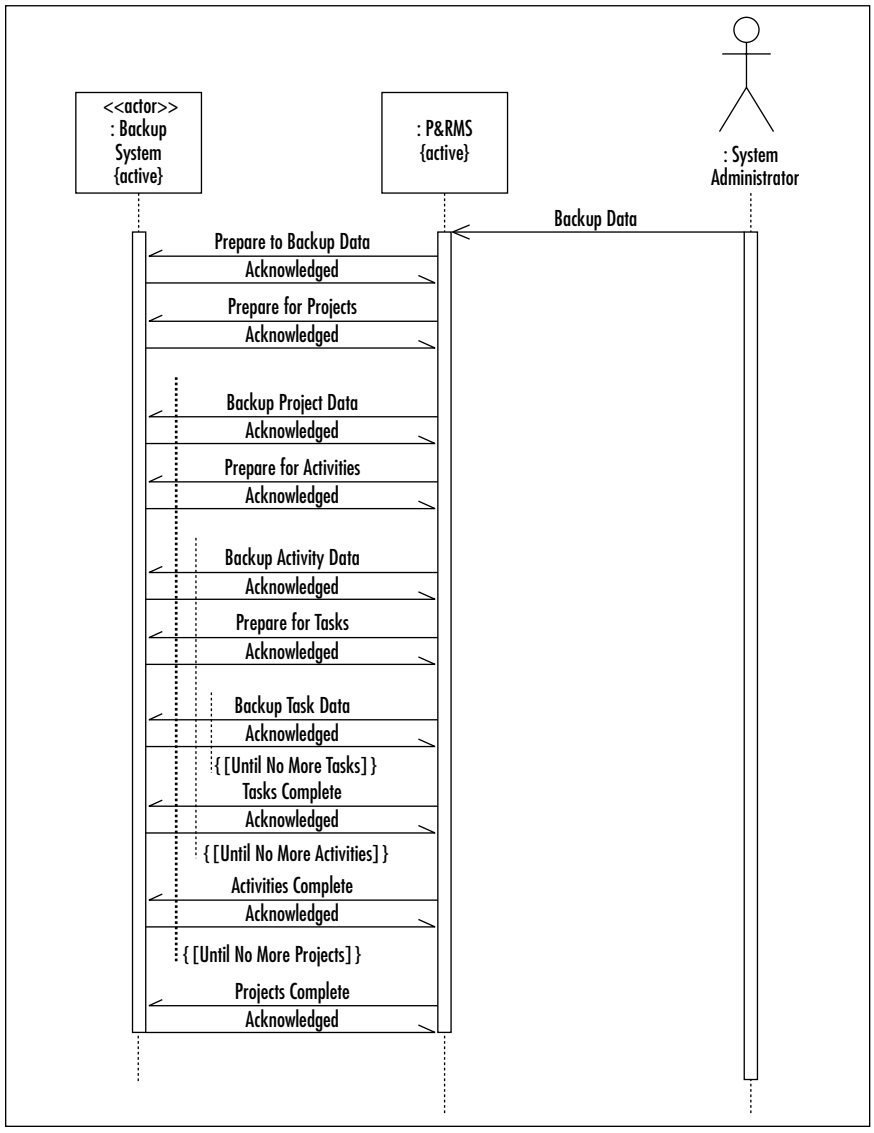


Figure 4-21: Back Up Project Data Sequence Diagram (Version 2)

Figure 4-22 shows a scenario in which a system administrator backs up the "Human Resource System Development" project data. All the loops within the diagram are removed, and the messages sent within the loops are shown. Messages are sent asynchronously. Again, this scenario is used to validate the sequence diagrams of Figures 4-20 and 4-21.

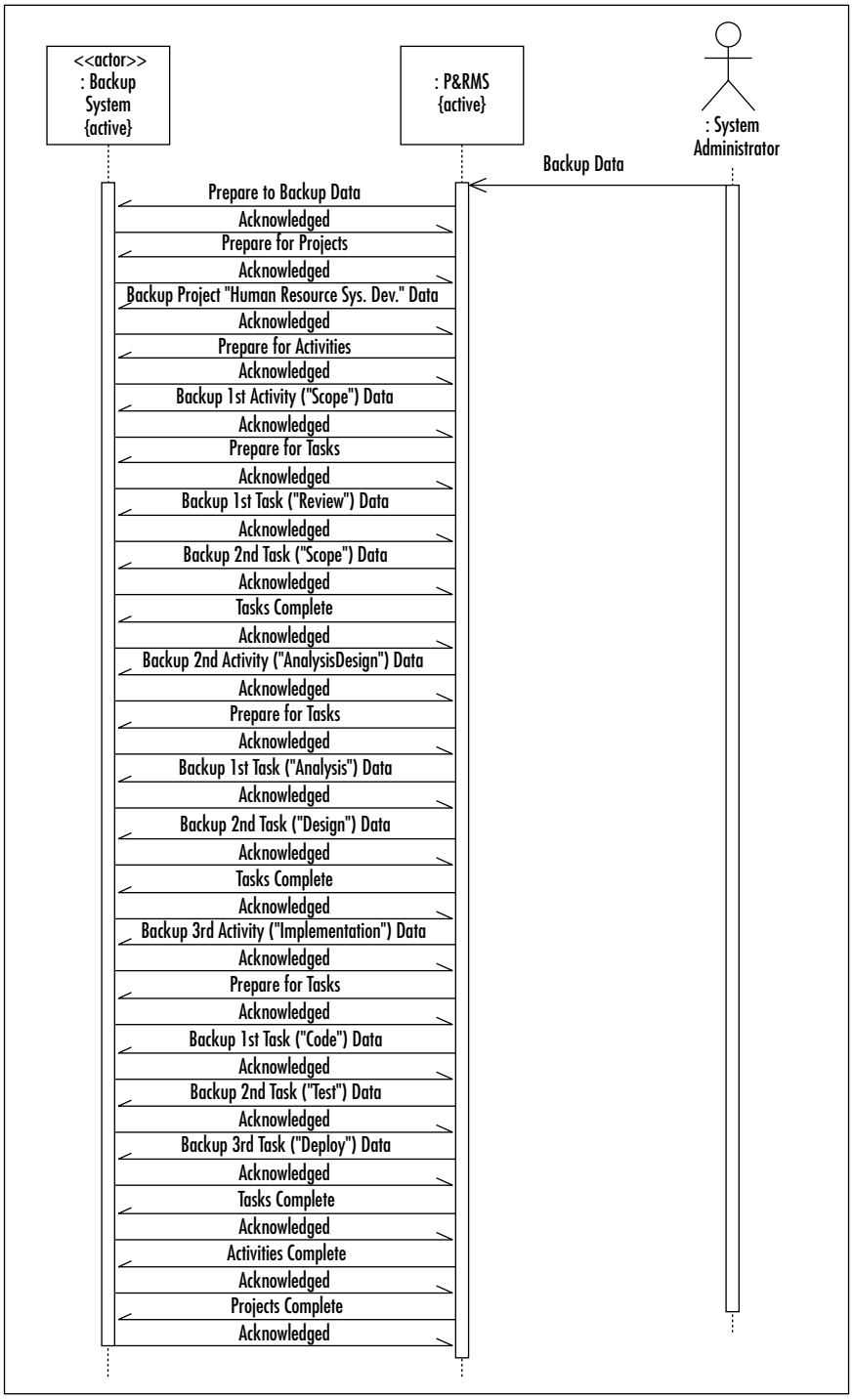


Figure 4-22: Back Up Project Data Scenario (Sequence Diagram)

Sequence diagrams are used to understand the messages (and operations) that classes within a system must support in order to perform some desired behavior or functionality. Scenarios are used to validate sequence diagrams, and a collection of sequence diagrams is part of the *dynamic model* of a system. Sequence diagrams are said to *specify* use cases because they specify the messages exchanged between class roles to realize use cases. Sequence diagrams are described in greater detail in Chapter 9.

Collaboration Diagrams

Collaboration diagrams describe interactions among classes and associations. These interactions are modeled as exchanges of messages between classes through their associations. Collaboration diagrams are a type of interaction diagram. Collaboration diagrams contain the following elements:

- *Class roles*, which represent roles that objects may play within the interaction.
- *Association roles*, which represent roles that links may play within the interaction.
- *Message flows*, which represent messages sent between objects via links. Links transport or implement the delivery of the messages.

Figure 4-23 elaborates the Assign Skill to Resource use case. It shows how a resource manager uses the system to assign a skill to a resource, and how the classes and their associations within the system work together to provide this functionality. The objects in rectangles represent class roles. The links (lines or paths) represent association roles. Class roles are named just like objects since they represent the objects that participate in the interaction. The arrows between class roles indicate message flows exchanged between objects. The message flows are labeled with both the sequence number of the message and the message that is sent between the class roles. A message triggers an operation in the receiving object.

The sequence numbers indicate the sequence of messages within the next higher level of nesting or passing of control. Each collaboration diagram starts with message 1. Messages that differ in one integer form a sequence at the same level of nesting. Within nesting level 1, message 1.1 precedes message 1.2, and message 1.2 precedes message 1.3, etc. Among nesting levels (or when messages originate from different class roles), messages are concurrent; that is, message 1 and message 2 are concurrent when originating from different class roles. Other numbering schemes may also be used, but they must be defined in the context in which they are used. Figure 4-23 is equivalent to Figure 4-14.

A resource manager will use the resource manager window, which is a user interface, to find a resource, find a skill, and assign the skill to the resource. The resource manager window will find a resource using a Resource class object, and a skill using a Skill class object. The window will assign a skill to a resource if the resource is not already assigned the skill. This condition is called a *guard condition* and is indicated within square brackets on the message flow originating from the window class role to the Resource-Skill class role.

Figure 4-24 shows how a resource manger assigns the “A/D” skill to Nora. The class roles represent actual objects, which are anonymous in this case. The

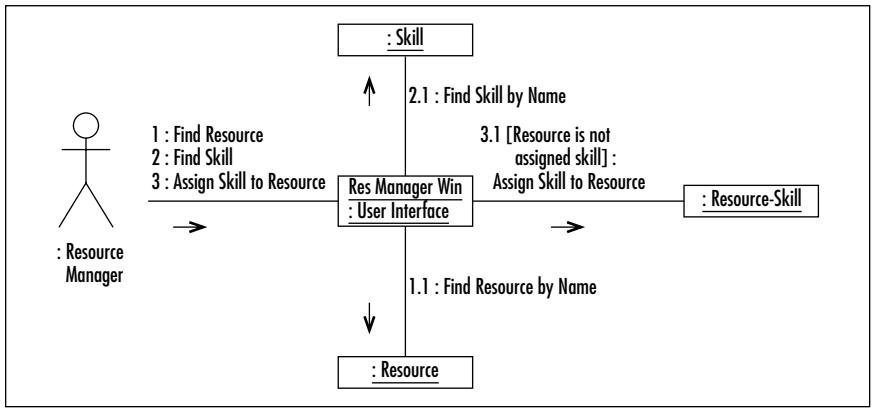


Figure 4-23: Assign Skill to Resource Collaboration Diagram

message flows are replaced with actual messages sent between the objects. Notice that the condition on the message originating from the window class role to the Resource-Skill class role is satisfied and the message is sent. This diagram is called a *scenario* since it shows one possible use of the collaboration diagram.

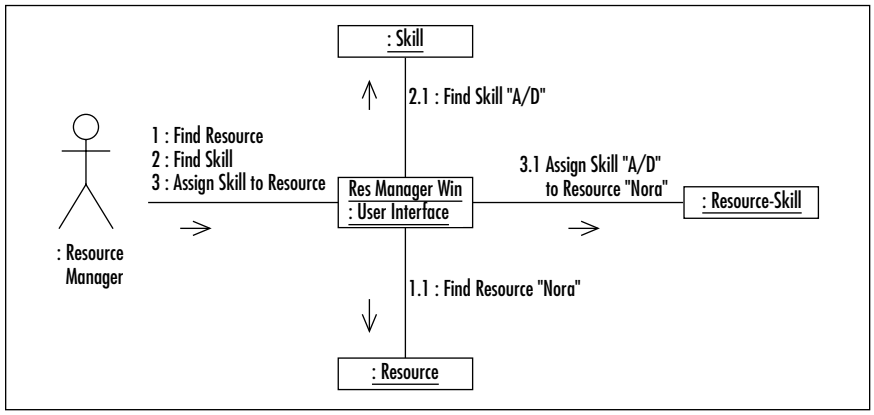


Figure 4-24: Assign the A/D Skill to Nora Scenario (Collaboration Diagram)

Figure 4-25 shows a scenario in which a resource manager assigns the "Impl" skill to Andy. The message flow originating from the window class role to the Resource-Skill class role is not shown since this skill is already assigned to Andy and the guard condition is not satisfied. This scenario and the scenario of Figure 4-24 are both satisfied by the collaboration diagram of Figure 4-23. Just as object diagrams may be used to validate class models, scenarios may be used to validate collaboration diagrams as well as sequence diagrams. The validation involves both the class roles used within the collaboration and the association roles that exist between the class roles.

Figure 4-26 elaborates the Remove Project use case. It shows how a project manager uses the system to remove a project from the database, and how the

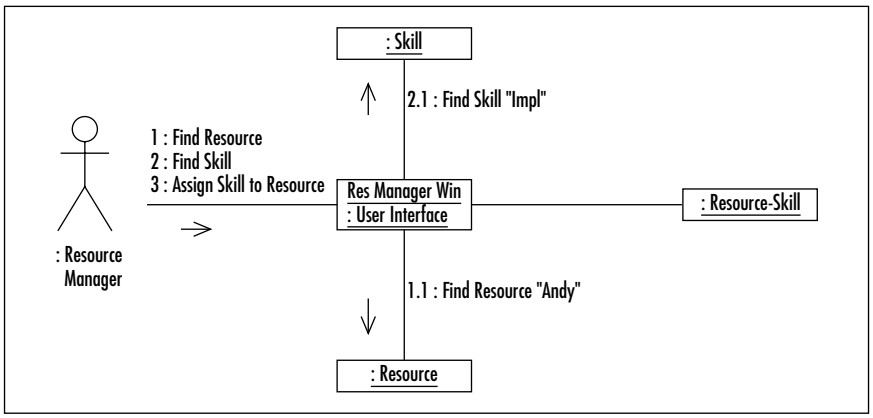


Figure 4-25: Assign the Impl Skill to Andy Scenario (Collaboration Diagram)

classes and their associations within the system work together to provide this functionality. The constraint strings enclosing an asterisk represent repetition, or a loop. The specification of the loop is enclosed in square brackets.

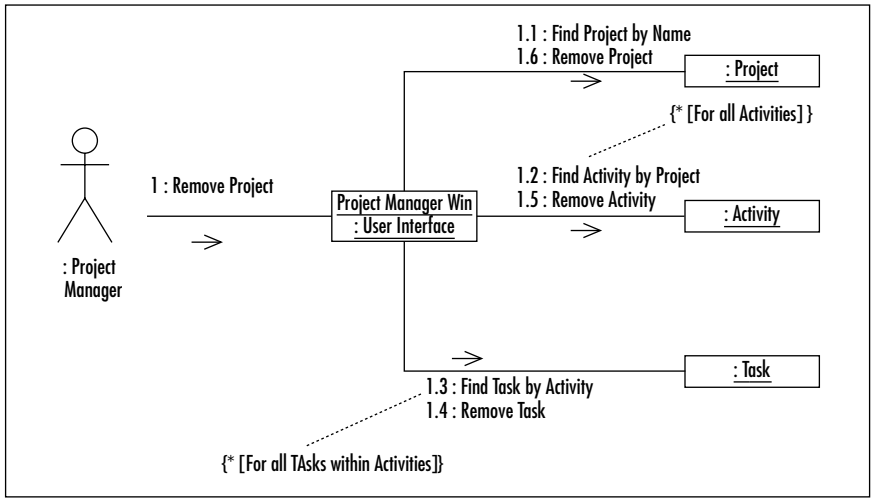


Figure 4-26: Remove Project Collaboration Diagram

A project manager will use the project manager window, which is a user interface, to specify a project to remove. Then the project manager window will find the project using a Project class object. Once the project is found, the project manager window will repeatedly find the activities within the project, remove their tasks, and remove the activities. The window uses an Activity class object to find and remove activities, and a Task class object to find and remove tasks. The repetition continues until all the tasks and activities are removed. The window completes processing by removing the project entry from the database.

Figure 4-27 shows a scenario in which a project manager removes the "Human Resource System Development" project from the system. Notice that all of the

loops are removed, and the messages sent between class roles are shown. Again, this scenario is used to validate the collaboration diagram of Figure 4-26.

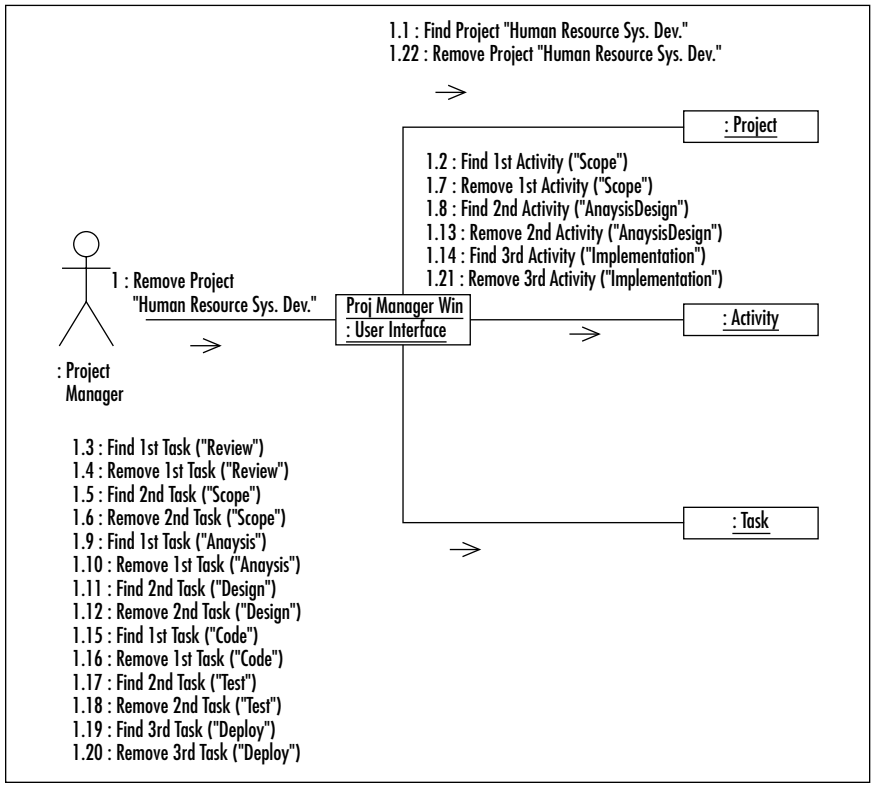


Figure 4-27: Remove Project Scenario (Collaboration Diagram)

Figure 4-28 elaborates the Backup System use case. It shows how a system administrator uses the system to back up the database, and how the Backup System actor is used within this interaction. Because the Backup System actor is actually another system and not part of the original system, the two are concurrently active and communicate in an asynchronous manner. Both systems are active, as indicated by the "active" constraint attached to their names. The diagram shows how the systems will send messages back and fourth asynchronously to coordinate the backing up of all the projects, their activities, and their tasks.

Some of the sequence numbers are preceded by other sequence numbers and a forward slash. The sequence numbers before the forward slash specify the messages that must have occurred to enable the message flow. This is used to synchronize concurrent active objects. All messages originating from the Backup System actor are synchronized with the messages from the other system, and all messages originating from the other system are synchronized with the messages from the Backup System actor.

Figure 4-29 shows a scenario in which a system administrator backs up the "Human Resource System Development" project data. All the loops within the diagram are

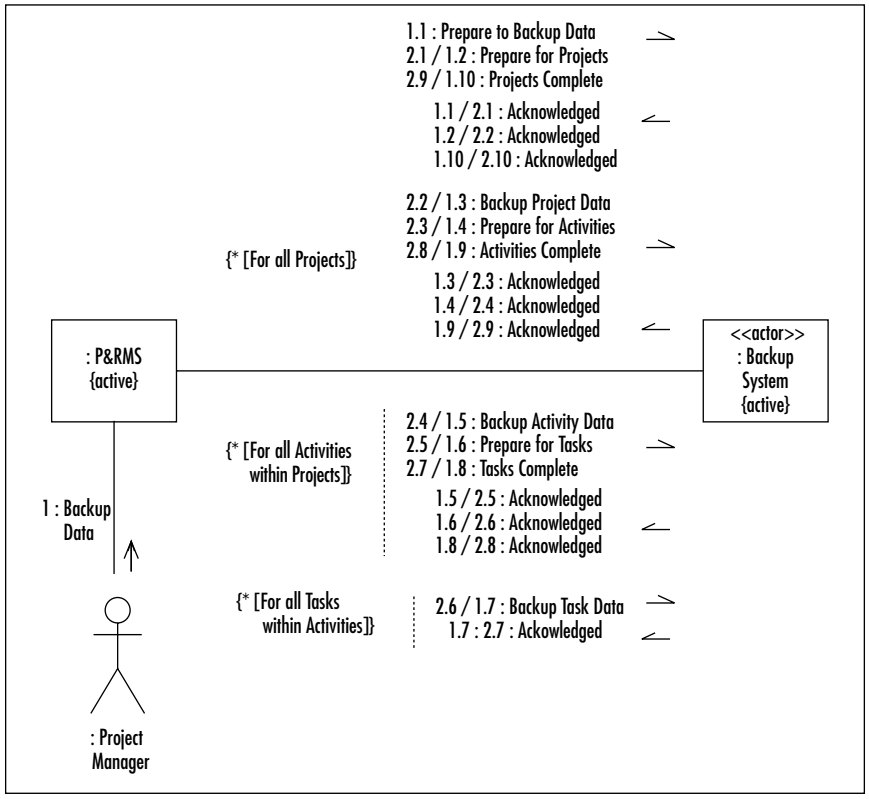


Figure 4-28: Back Up Project Data Collaboration Diagram

removed, and the message flows are shown with their synchronization. Again, this scenario is used to validate the collaboration diagram of Figure 4-28.

Collaboration diagrams are used to understand the messages (and operations) that classes within a system must support in order to perform some desired behavior or functionality, and the associations that must exist between classes to enable the messages to flow among them. Scenarios are used to validate collaboration diagrams, and a collection of collaboration diagrams are part of the dynamic model of a system. Collaboration diagrams are said to *realize* use cases because they specify the messages exchanged between class roles and their association roles. A collaboration diagram is really the result of overlaying the messages from a sequence diagram onto a class diagram. Collaboration diagrams are described in greater detail in Chapter 10.

Statechart Diagrams

Statechart (or state) diagrams describe the states and responses of a class. Statechart diagrams describe the behavior of a class in response to external stimuli. These diagrams contain the following elements:

- *States*, which represent the situations during the life of an object in which it satisfies some condition, performs some activity, or waits for some occurrence.

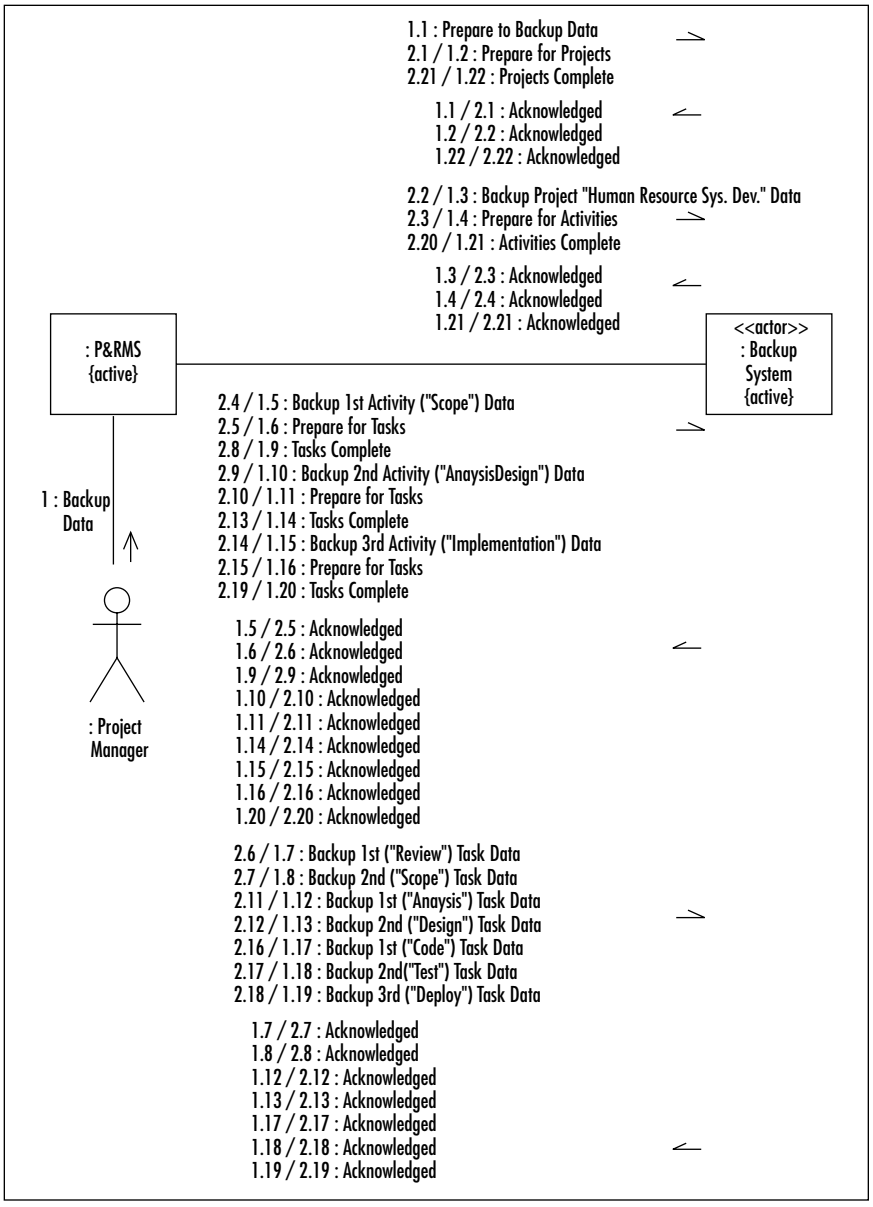


Figure 4-29: Back Up Project Data Scenario (Collaboration Diagram)

- *Transitions*, which represent relationships between the different states of an object.

Figure 4-30, a statechart diagram that might be associated with our sample project, shows that a resource may be assigned no tasks or may be assigned some tasks. The rectangles with rounded corners represent states. Each state has a name. The figure also shows that a resource is initially not assigned to any tasks. The small filled circle with an arrow points to the *initial state*.

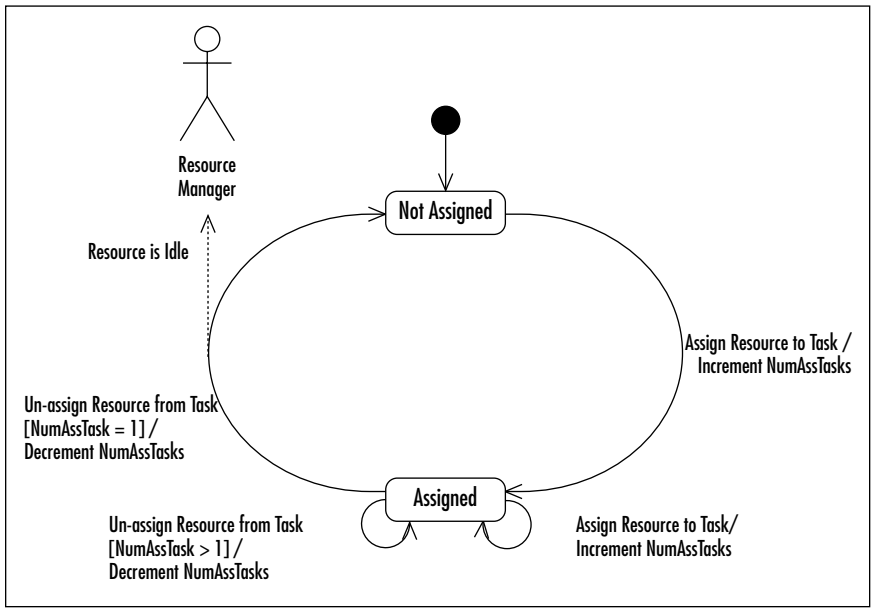


Figure 4-30: Resource Statechart Diagram

Assigning a task to a resource changes it from the Not Assigned state to the Assigned state. The solid arrow between these two states represents a transition. The transition is labeled with an event that triggers the transition, and an action that results from the transition. In this case, the NumAssTasks is incremented to keep track of the number of tasks assigned to the resource. When a resource is assigned, the resource may be assigned more tasks, in which case the resource remains in the Assigned state and the NumAssTasks attribute is incremented.

When a resource is unassigned from a task, the NumAssTasks attribute is decremented. An old value greater than 1 leaves the resource in the Assigned state, whereas an old value of exactly 1 causes a state transition so that the resource is in the Not Assigned state. The effect of unassigning a resource thus depends on a condition, which is specified in square brackets following the event that triggers the transition. When the resource returns to the Not Assigned state, a message is sent to inform the resource manager that the resource is idle. The dashed arrow from the transition to the Resource Manager actor represents a message sent to the object when the transition fires.

Figure 4-31 shows that when the overall system to which all the models belong (P&RMS) and the Backup System are deployed, they will both be inactive, and the P&RMS's state will depend on the state of the Backup System. The short heavy bar with two transitions leaving it represents a *splitting of control*. That is, when the incoming transition to "Deploy Systems" fires, two resulting transitions fire and multiple states become active.

When both systems are inactive, we may retire the systems concurrently. The short heavy bar with two transitions entering it represents a *synchronization of control*.

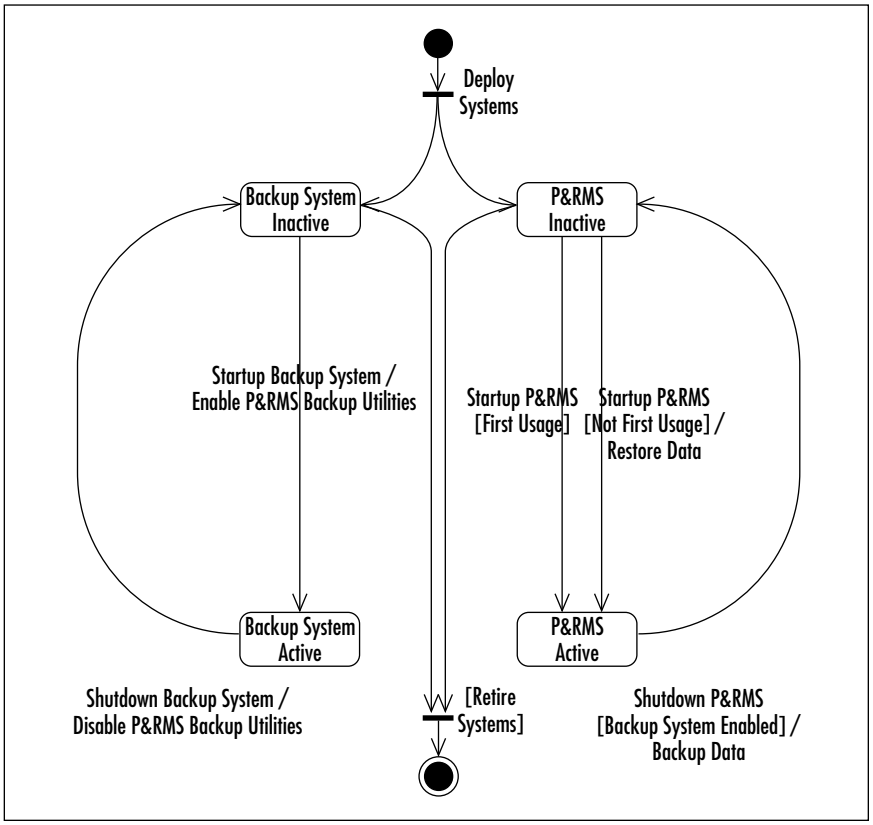


Figure 4-31: System Statechart Diagram

That is, when all the incoming transitions fire, the resulting transition fires. Once the two systems are retired, nothing else occurs. The circle surrounding a smaller filled circle represents the *final state*. Once the final state is reached, the objects whose states are depicted in the diagram are destroyed.

The left side shows that when the Backup System is started, the P&RMS backup utility is enabled. When the Backup System is shut down, the P&RMS backup utility is disabled. The right side of Figure 4-31 shows that when the P&RMS is started for the first time, the system simply becomes active. However, when the P&RMS is shut down and the Backup System is enabled, the Back Up Data action occurs and the system enters the P&RMS Inactive state. Then, when the P&RMS system is restarted (not the first use), the system becomes active and data is restored from the Backup System.

Figure 4-32 elaborates the P&RMS Active state by detailing the states of the different parts of the system.

When the P&RMS becomes active, the resource manager functionality, project manager functionality, and system administrator functionality are all inactive. The P&RMS Active state is called a *superstate* and has its name displayed at the top of

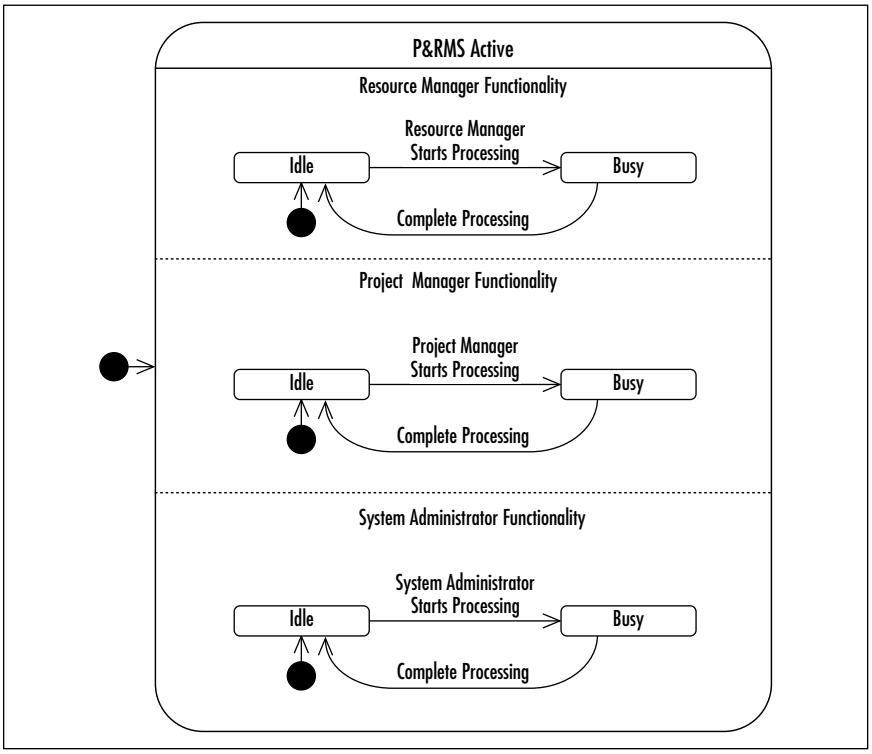


Figure 4-32: System Substate Statechart Diagram

the state. The superstate is divided into three concurrent *substates* for each part of the system. When a resource manager starts processing, the parts of the system that provide the resource manager functionality become busy until the processing is complete. Similar effects take place when a project manager starts processing or a system administrator starts processing.

Statechart diagrams are used to understand the messages (and operations) and attributes that classes within a system must support in order to perform some desired behavior or functionality. These diagrams are part of the dynamic model of a system. Statechart diagrams are described in greater detail in Chapter 11.

Activity Diagrams

Activity diagrams describe the activities of a class. These diagrams are similar to statechart diagrams and use similar conventions, but activity diagrams describe the behavior of a class in response to internal processing rather than external events as in statechart diagrams. These diagrams contain the following elements:

- *Swimlanes*, which represent responsibilities of one or more objects for actions within an overall activity; that is, they divide the activity states into groups and assign these groups to objects that must perform the activities.

- *Action states*, which represent atomic, or noninterruptible, actions of entities or steps in the execution of an algorithm.
- *Action flows*, which represent relationships between the different action states of an entity.
- *Object flows*, which represent the utilization of objects by action states and the influence of action states on objects.

Figure 4-33 shows how the Assign Skill to Resource use case is performed. The figure shows how the classes within the system work together and perform actions to provide this functionality: A Resource class object will find the resource by name, a Skill class object will find the skill by name, and a Resource-Skill class object will assign the skill to the resource. The regions denoted by vertical solid lines are swimlanes, and the objects at the top of the swimlanes are responsible for the action states within the swimlane. The shapes with straight top and bottom lines and convex arcs on the sides are action states.

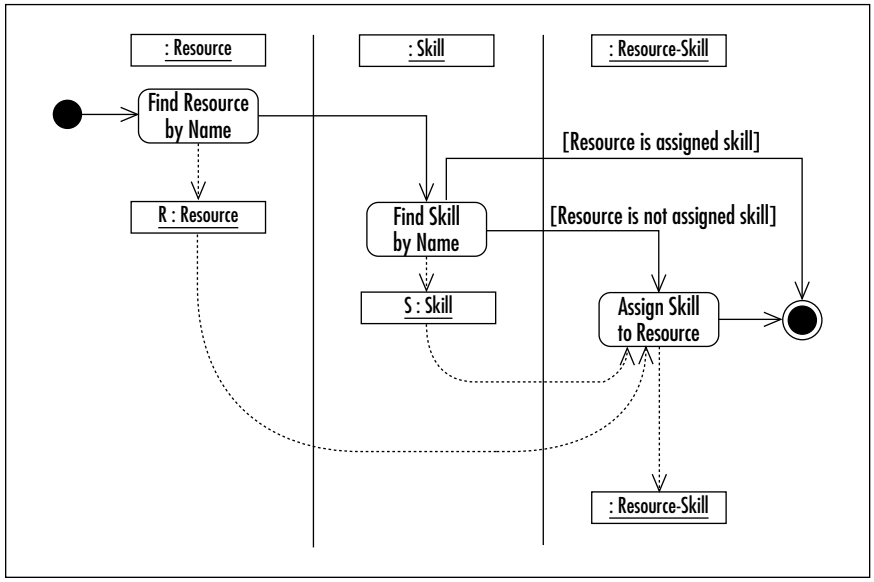


Figure 4-33: Assign Skill to Resource Activity Diagram

After a resource is found by name, the skill is found by name and is assigned to the resource. The solid arrows represent action flows that are automatically triggered at the completion of the action states. Notice that an initial and a final state are used, as in statechart diagrams.

Once a resource is found by name, a resource object is created. Once a skill is found by name, a skill object is created. Assigning a skill to a resource uses the two objects created by the previous steps. The dashed arrows represent object flows. Object flows originating from an action state indicate that the action creates or influences the object. Object flows from an object to an action state indicate that the action state uses the object.

Activity diagrams are similar to statechart diagrams. These diagrams are used to understand the operations that classes within a system must support in order to perform some desired behavior or functionality, and to understand how these objects interact with other objects in the system. These diagrams are part of the dynamic model of a system. Activity diagrams are described in greater detail in Chapter 12.

Component Diagrams

Component diagrams describe the organization of and dependencies among software implementation components. These diagrams contain *components,* which represent distributable physical units, including source code, object code, and executable code.

Figure 4-34 shows the development-time and run-time physical objects implementing the system. The rectangles with two small rectangles projected from the side represent components. The small labeled circle attached via a line to a component represents an interface.

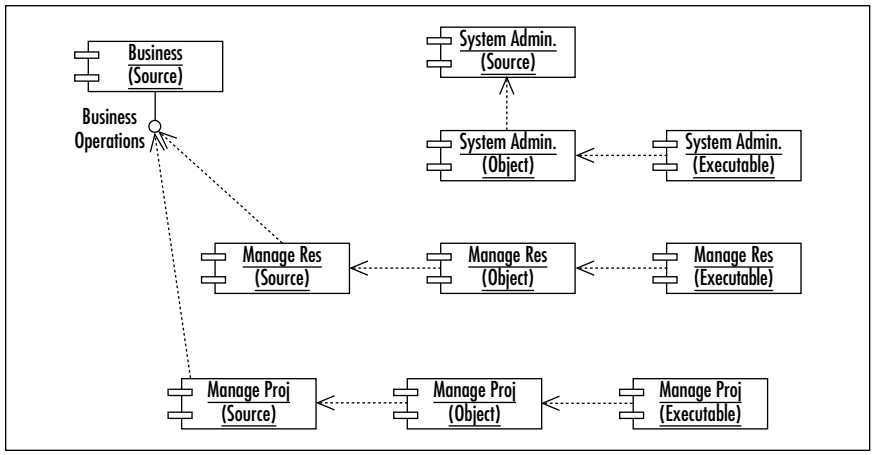


Figure 4-34: System Component Diagram

The object code components are dependent on their associated source code components. The dashed arrows represent *dependencies* between components. Thus, in Figure 4-34, the System Administrator executable program is dependent on the System Administrator object code component, and the System Administrator object code component is dependent on the System Administrator source code file. Similar dependencies are shown for the Manage Resource and Manage Project executable programs. Their source code files use the business operations provided by the Business component.

Component diagrams are described in greater detail in Chapter 13.

Deployment Diagrams

Deployment diagrams describe the configuration of processing resource elements and the mapping of software implementation components onto them. These

diagrams contain components and *nodes,* which represent processing or computational resources, including computers, printers, and so forth.

Figure 4-35 shows that the resource manager application will execute on a resource manager desktop client computer. The three-dimensional cubes represent nodes. This figure also shows that the project manager application will execute on a project manager desktop client computer. Both types of computers have access to printers and an application server. The solid paths represent lines of communication between nodes.

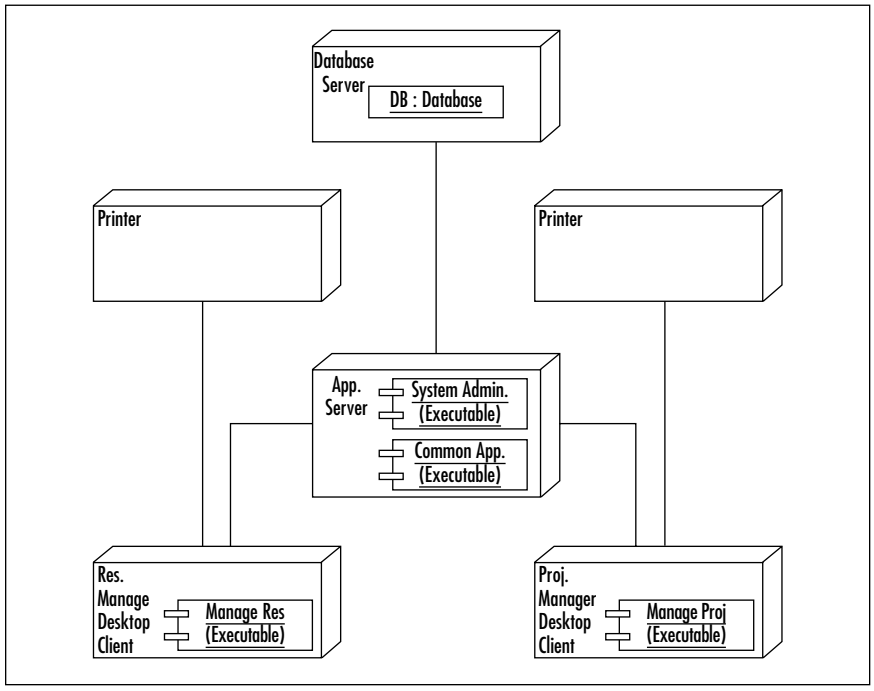


Figure 4-35: System Deployment Diagram

The system administrator application and other common application components will execute on the server and access a database server containing the database object named "DB."

Deployment diagrams are described in greater detail in Chapter 14.

Other Notation and Information

Figure 4-36 shows how model elements may be grouped into packages and related. The large rectangles with a small rectangle attached on one corner represent packages. *Packages* are general-purpose grouping mechanisms used to organize model elements. They are used within the other diagrams.

In Figure 4-36, the User Interface package contains classes that implement the user interface of a system. The Business package contains classes that represent business objects. The Communication package contains classes that are used to

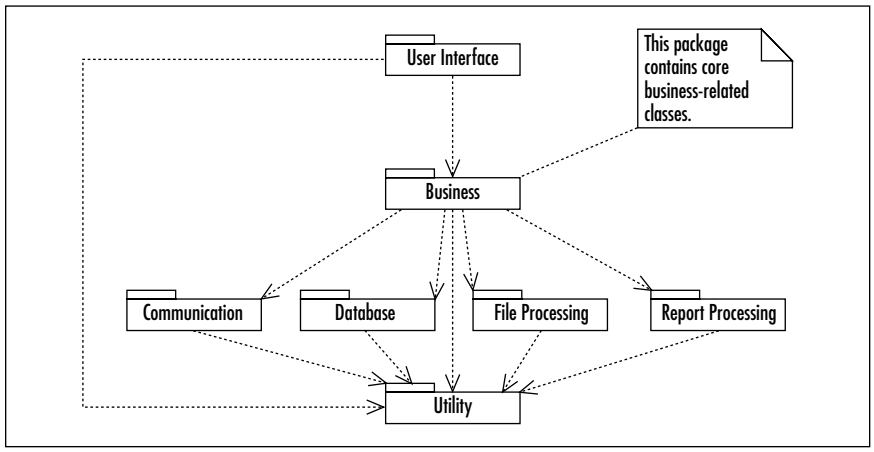


Figure 4-36: Packages

communicate with other systems. The Database package contains classes used to access a database. The File Processing package contains classes used for processing files. The Report Processing package contains classes used for generating reports. The Utility package contains general utility classes.

The User Interface package is dependent on the Business package and the Utility package. The Business package is dependent on the Communication package, Database package, File Processing package, Report Processing package, and Utility package. The Communication package, Database package, File Processing package, and Report Processing package are dependent on the Utility package.

Notes may be placed on diagrams to attach text to model elements. Rectangles with a bent upper right corner represent notes. They may be attached to specific model elements using dashed lines or attached to the whole diagram rather than any specific model element.

Packages, notes, and general diagramming are described in greater detail in Chapter 6.

Figure 4-37 shows a class diagram with rules or constraints attached to different elements. All projects in this diagram must start before January 1, 1999. The *constraint* string attached to the Project class represents this rule, to which all objects of the class must adhere.

All activities must belong to projects that start after January 1, 1998. The constraint string attached to the Activity class represents this rule. The term "project" references the Project class object of or linked with an Activity object, and the StartDate references the StartDate attribute of the Project class object. Note that there may only be one Project class object for any Activity class object (as specified by the multiplicity).

All resources must have less than 15 tasks assigned to them. The constraint string attached to the Resource class represents this rule. The term "task" references the collection of Task class objects of the Resource object, and the "size" is a predefined attribute of a collection (as explained in Chapter 16). Note that there

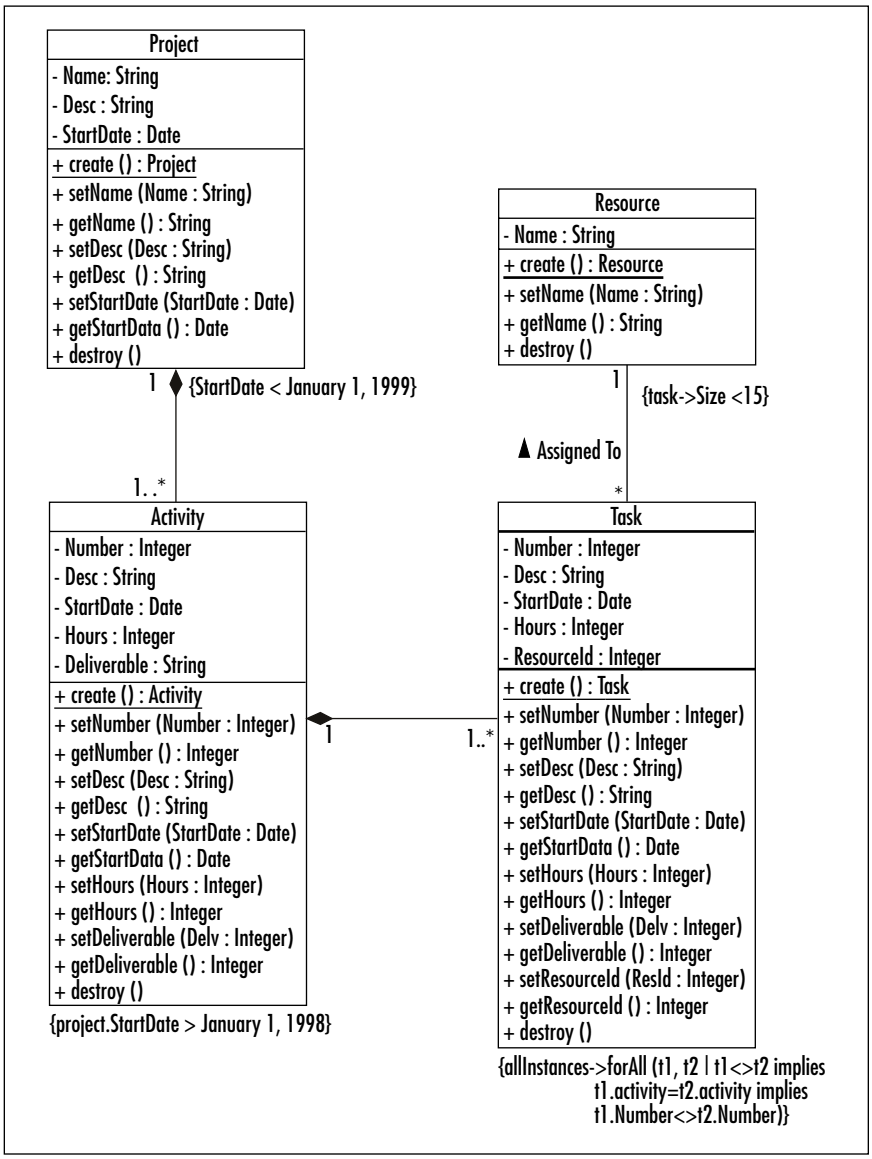


Figure 4-37: Detailed Project Class Diagram with OCL Expressions

may be any number of Task class objects associated with any Resource class objects (as specified by the multiplicity); this is why the arrow operator is used.

No two tasks within the same activity may have the same number. The constraint string attached to the Task class represents this rule. The "forAll" specifies that all objects of the class are iterated over. The "t1" variable references one object, and the "t2" variable references another object. "t1<>t2" dictates that we have two

different tasks. If we do, "t1.activity=t2.activity" dictates that the tasks belong to the same activity, and, if so, the task numbers should be different. If this rule fails for any objects, the object model is ill formed.

The Object Constraint Language (OCL) is a language for expressing conditions attached to model elements. When attached to a class, these expressions specify rules to which objects of the class must adhere. The OCL is described in greater detail in Chapter 16. Some expressions can be very complex, and a natural language may be used in place of the OCL.

Figure 4-38 shows four different ways to represent an actor. An actor is a stereotype of a class. The string "actor" enclosed in guillemets (<< >>), or double angle brackets, preceding the name of the element indicates the type of the element.

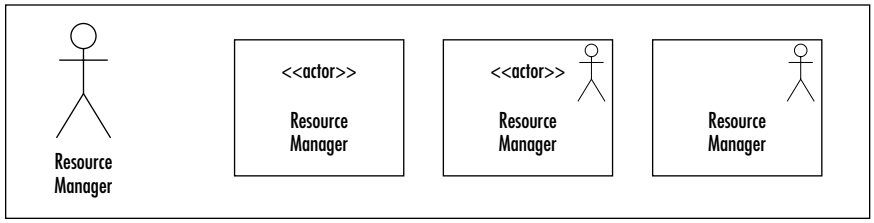


Figure 4-38: Stereotypes

The UML allows new types of elements to be established using a stereotype. Once a new type of element is established, there are four ways to depict an instance of the element: an icon, a stereotype string, both an icon and a stereotype string, or an icon with a class rectangle.

Figure 4-39 shows a project manager who is removing a project. Stereotype icons are used within the diagram to emphasize the types of objects involved. The Project Manager Window is an interface class. The Remove Project is a control class that is responsible for controlling interactions among a collection of objects. The Project, Activity, and Task classes are entity stereotypes representing database information. These stereotypes are predefined in the UML, and are explained in Chapter 15.

Figure 4-40 shows a Skill class with two properties or characteristics. The first property is a constraint specifying that an object of the class is created, and the second property is called a *keyword-value pair* or *tagged value*. The second property indicates that objects of the class have a keyword "documentation" and a value associated with this keyword. *Properties* of model elements are used by modeling tools and enable us to associate information with model elements.

The UML stereotypes enable the UML to be extended with new types of modeling elements, and properties enable the UML to be customized to support specialized characteristics of model elements. These are called *extension mechanisms*. They are described in greater detail in Chapter 15, including the standard stereotypes, constraints, and tagged values.

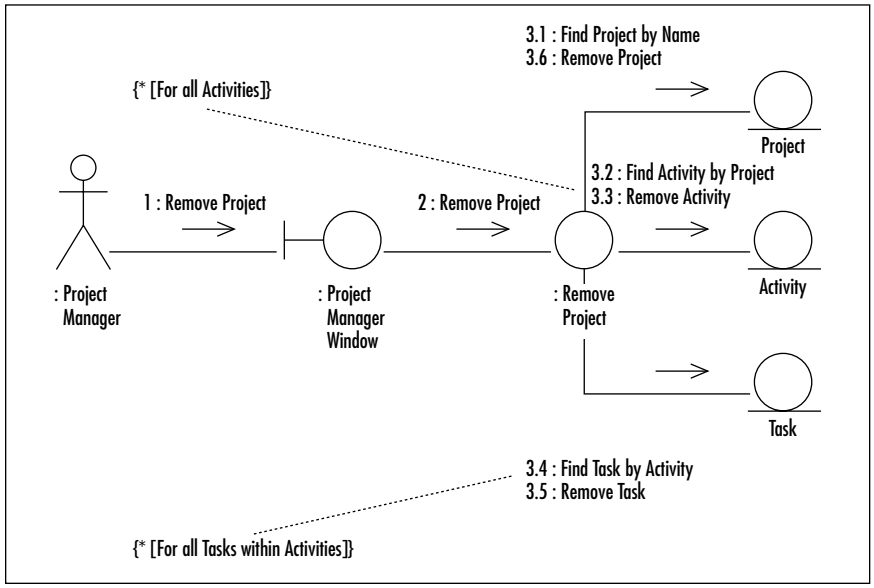


Figure 4-39: Remove Project Collaboration Diagram

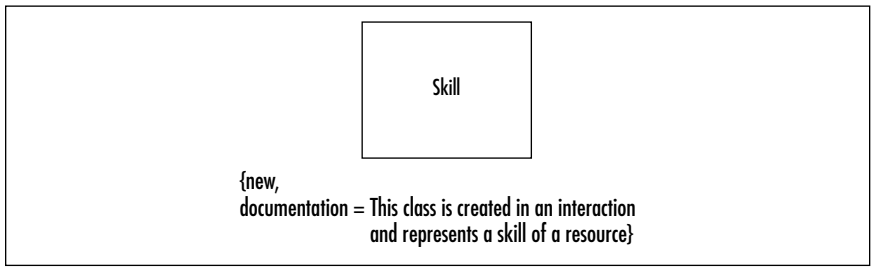


Figure 4-40: Constraints and Tagged Values

,ch04.17526 Page 110 Wednesday, May 9, 2001 4:02 PM