1. Introduction to Software Engineering: Solutions

1-1 What is the purpose of modeling?

The purpose of modeling is to reduce complexity by building a simplified representation of reality which ignores irrelevant details. What is relevant or not is defined by the questions the model will be used to answer.

1–2 A programming language is a notation for representing algorithms and data structures. List two advantages and two disadvantages of using a programming language as sole notation throughout the development process.

Advantages:

- Developers need only learn one notation for all development activities.
- Traceability among models and between models and code is made easier since they are written in the same notation.

Disadvantages:

- A programming language is a low level notation which is difficult to use for representing user requirements, for example.
- A programming language enables and encourages developers to represent implementation details too early.

1–3 Consider a task you are not familiar with, such as designing a zero-emissions car. How would you attack the problem?

This is an open ended question whose purpose is for students think about problems they cannot solve without help. Answers should contain two or more of the following points:

- Define the problem precisely by gathering information from potential users.
- Discover the boundaries of the solution space by gathering information from application domain experts.
- Brainstorm ideas with other people, including experts and non experts
- Evaluate ideas using prototypes, simulations, and candidate users.

1–4 What is meant by "knowledge acquisition is not sequential"? Provide a concrete example of knowledge acquisition that illustrates this.

Knowledge acquisition is nonlinear in the sense that the acquisition of a new piece of knowledge may invalidate prior knowledge. In other terms, knowing one more piece of information may lead you to realize that what you thought you knew is invalid. Galileo Galilei invalidated the earth centric model of the universe by observing the moons of Jupiter and the phases of Venus.

1–5 Hypothesize a rationale for the following design decisions:

This exercise tests if the student understands the difference between a decision and its rationale. The exact rationale provided by the student is not important as long as it is rationale (e.g., the answer to the first bullet could have been to allow snow white's seven dwarves to purchase train tickets).

- *"The ticket distributor will be at most one and a half meters tall."*

Enable children and persons in wheelchair to purchase tickets.

- *"The ticket distributor will include two redundant computer systems."*

To achieve a high level of availability such that ticket distribution is not interrupted (and thus, ticket sales not lost in the case of the failure of one computer).

- *"The ticket distributor will include a touch screen for displaying instructions and inputing commands. The only other control will be a cancel button for aborting a transaction."*

Enable substantial modifications to the interface (e.g., increase the number of tariff zones or the number of products) without changes to the hardware.

1–6 Specify which of the following statements are functional requirements and which are nonfunctional requirements:

- *"The ticket distributor must enable a traveler to buy weekly passes."*
- *"The ticket distributor must be written in Java."*
- *"The ticket distributor must be easy to use."*

The first requirement is functional, the third is nonfunctional. Using the definitions in Chapter 1, the second requirement is nonfunctional. In Chapter 4, we will call this requirement a pseudo requirement as it constrains aspects of the system that are not visible to the user.

1–7 Specify which of the following decisions were made during requirements or system design:

- *"The ticket distributor is composed of a user interface subsystem, a subsystem for computing tariff, and a network subsystem managing communication with the central computer."*
- *"The ticket distributor will use PowerPC processor chips."*
- *"The ticket distributor provides the traveler with an on-line help."*

The first decision is a system design decision. The second decision is also a system design decision if made by developers (otherwise, it is a requirements decision). The third decision is a requirements decision.

1–8 In the following description, explain when the term account is used as an application domain concept and when as a solution domain concept:
"Assume you are developing an online system for managing bank accounts for mobile customers. A major design issue is how to provide access to the accounts when the customer cannot establish an online connection. One proposal is that accounts are made available on the mobile computer, even if the server is not up. In this case, the accounts show the amounts from the last connected session."

The first two occurrences of "account" are application domain concepts while the last two occurrences are solution domain concepts. The phrases "accounts are made available on the mobile computer" and "the accounts show the amounts from the last connected session" denote a solution domain concept that gives users the illusion that they are accessing their bank accounts on their mobile computer. However, the actual bank account is not on the mobile computer.

1–9 What is the difference between a task and an activity?

An activity is composed of a number of tasks. Both represent work, but tasks cannot conveniently be decomposed any further.

1–10 A passenger aircraft is composed of several millions of individual parts and requires thousands of persons to assemble. A four-lane highway bridge is another example of complexity. The first version of Word for Windows, a word processor released by Microsoft in November 1989, required 55 person-years, resulted into 249,000 lines of source code, and was delivered 4 years late. Aircraft and highway bridges are usually delivered on time and below budget, whereas software is often not. Discuss what are, in your opinion, the differences between developing an aircraft, a bridge, and a word processor, which would cause this situation.

This is an open question whose purpose is to have students realize that software systems are not the only complex systems out there. Answers should include two or more of the following points:

- To estimate the budget and schedule for a new bridge or aircraft, engineers use actual data from previous bridges and aircraft. Word for Windows was an innovative piece of software with few or no precedents.
- Many bridges and aircraft are simply refinements of other existing artifacts. This reduces the proportion of the overall effort that is dedicated to design (which is the most difficult to estimate).
- Bridges and aircraft are often associated with severe financial penalties when late or over budget.

- Bridges and aircraft have safety requirements associated with them. This leads to a conservative approach to development including the use of mature technologies and well defined processes.
- Bridges and aircraft are sometimes delivered late too.

2. Modeling with UML: Solutions

2-1 Consider an ATM system. Identify at least three different actors that interact with this system.

An actor is any entity (user or system) that interacts with the system of interest. For an ATM, this includes:

- Bank Customer
- ATM Maintainer
- Central Bank Computer
- Thief

The last actor is often referred to as a "misactor" in the literature, because it is an actor that interacts with the system but shouldn't.

2–2 Can the system under consideration be represented as an actor? Justify your answer.

The system under consideration is not external to the system and shouldn't be represented as an actor. There are a few cases, however, when representing the system as an actor may clarify the use case model. These include situations where the system initiates uses cases, for example, as time passes (Check for Outdated Articles, Send Daily Newsletter).

2–3 What is the difference between a scenario and a use case? When do you use each construct?

A scenario is an actual sequence of interactions (i.e., an instance) describing one specific situation; a use case is a general sequence of interactions (i.e., a class) describing all possible scenarios associated with a situation. Scenarios are used as examples and for clarifying details with the client. Use cases are used as complete descriptions to specify a user task or a set of related system features.

2–4 Draw a use case diagram for a ticket distributor for a train system. The system includes two actors: a traveler, who purchases different types of tickets, and a central computer system, which maintains a reference database for the tariff. Use cases should include: `BuyOneWayTicket`, `BuyWeeklyCard`, `BuyMonthlyCard`, `UpdateTariff`. *Also include the following exceptional cases:* `Time-Out` *(i.e., traveler took too long to insert the right amount),* `TransactionAborted` *(i.e., traveler selected the cancel button without completing the transaction),* `DistributorOutOfChange`, *and* `DistributorOutOfPaper`.

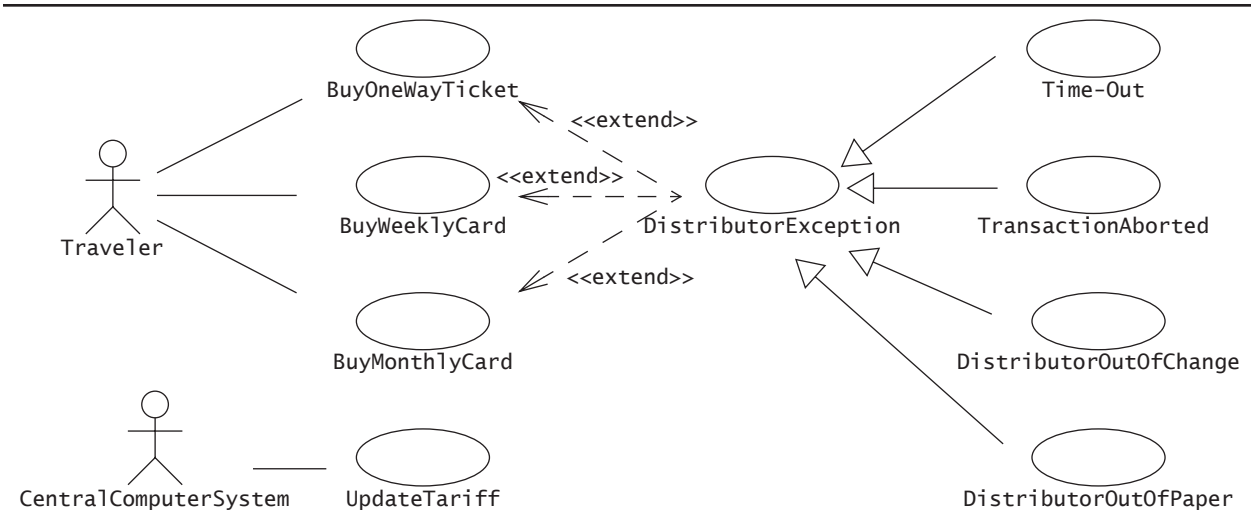


Figure 2-1 Example solution to Exercise 2–4.

This questions can have several correct answers, Figure 2-1 being a possible answer. The following elements should be present:

- The relationship between an actor and a use case is a communication relationship (undirected solid line).
- The relationship between exceptional use cases and common use cases is an `<<extend>>` relationship.
- The exceptional use cases described in the exercise only apply to the use cases invoked by the traveler.

The following elements should be present in a "good" answer:

- All exceptions apply to all traveler use cases. Instead of drawing 3x4 relationships between these use cases, an abstract use case from which the exceptional use case inherit can be used, thus reducing the number of `<<extend>>` relationships to 3 at the cost of introducing 4 generalization relationships.
- The student can introduce exceptional use cases not specified in the exercise that apply to the `CentralComputerSystem` use cases.

2–5 Write the flow of events and specify all fields for the use case UpdateTariff that you drew in Exercise 2–4. Do not forget to specify any relationships.

Figure 2-2 depicts a possible solution for this exercise.

Use case name	`UpdateTariff`
Participating actor	Initiated by `CentralComputerSystem`
Flow of events	1. The `CentralComputerSystem` activates the "UpdateTariff" function of the ticket distributors available on the network. 2. The ticket distributor disables the traveler interface and posts a sign indicating that the ticket distributor is under maintenance. 3. The ticket distributor waits for the new database from the `CentralComputerSystem`. 4. After waiting a minute for the ticket distributors to reach a waiting state, the `CentralComputerSystem` broadcasts the new database. 5. The ticket distributor system receives the new database of tariff. Upon complete, the ticket distributor sends an acknowledgement to the `CentralComputerSystem`. 6. After acknowledgment, the ticket distributor enables the traveler interface and can issue tickets at the new tariff. 7. The `CentralComputerSystem` checks if all ticket distributors have acknowledged the new database. If not, the CentralComputerSystem invokes the `CheckNonRespondingDistributors` use case.
Entry condition	• The ticket distributor is connected to a network reachable by the CentralComputerSystem.
Exit condition	• The ticket distributor can issue tickets under the new tariff, OR • The ticket distributor is disabled and displays a sign denoting that it is under maintenance.
Quality requirements	• The ticket distributor stays offline at most 2 minutes and is considered out-of-order otherwise.

Figure 2-2 A possible solution for the `UpdateTariff` use case.

2–6 Draw a class diagram representing a book defined by the following statement: "A book is composed of a number of parts, which in turn are composed of a number of chapters. Chapters are composed of sections." Focus only on classes and relationships.

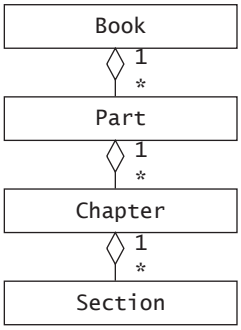


Figure 2-3 Example solution for Exercise 2–6.

This exercise checks the student's understanding of basic aspects of class diagrams, including:

- Classes are represented with rectangles.
- The attribute and operations compartment can be omitted.
- Aggregation relationships are represented with diamonds.
- Class names start with a capital letter and are singular.

2–7 Add multiplicity to the class diagram you produced in Exercise 2–6.

See Figure 2-3. Aggregation does not imply multiplicity, thus the `1..*` multiplicity is necessary.

2–8 Draw an object diagram representing the first part of this book (i.e., Part I, Getting Started). Make sure that the object diagram you draw is consistent with the class diagram of Exercise 2–6.

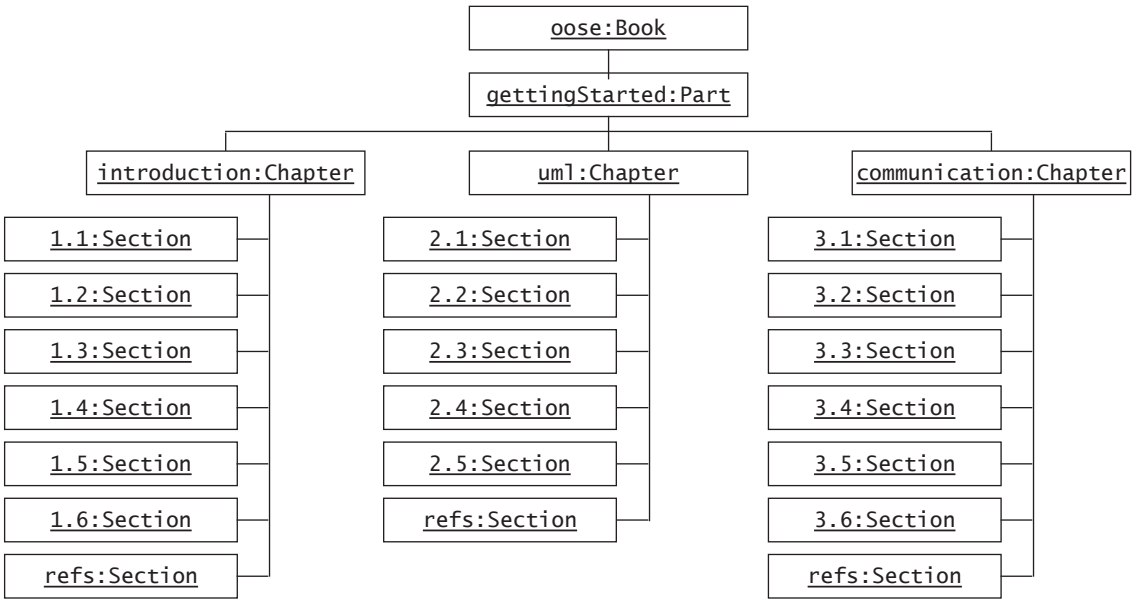


Figure 2-4 Example solution for Exercise 2–8.

This exercise checks the student's understanding of basic aspects of object diagrams, including:

- Objects are represented with rectangles and underlined labels.
- The class of an object is included in the label of the object (e.g., `uml:Chapter` is of class `Chapter`).
- Links are represented with solid lines

2–9 Extend the class diagram of Exercise 2–6 to include the following attributes:

- *a book includes a publisher, publication date, and an ISBN*
- *a part includes a title and a number*
- *a chapter includes a title, a number, and an abstract*
- *a section includes a title and a number*

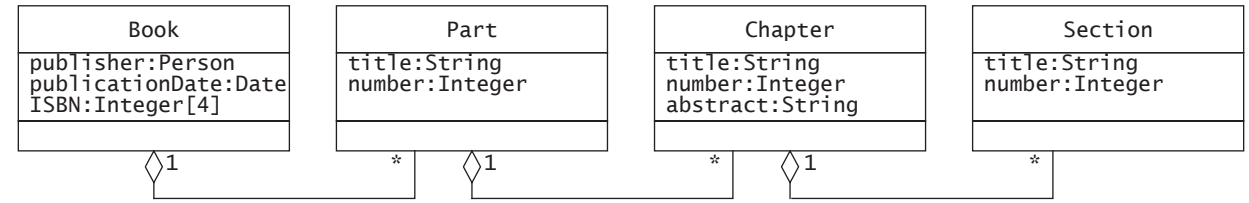


Figure 2-5 Example solution for Exercise 2–9.

This exercises checks the student's knowledge of attributes and their representation in UML (page 45).

2–10 Consider the class diagram of Exercise 2–9. Note that the `Part`, `Chapter`, *and* `Section` *classes all include a title and a number attribute. Add an abstract class and a generalization relationship to factor out these two attributes into the abstract class.*

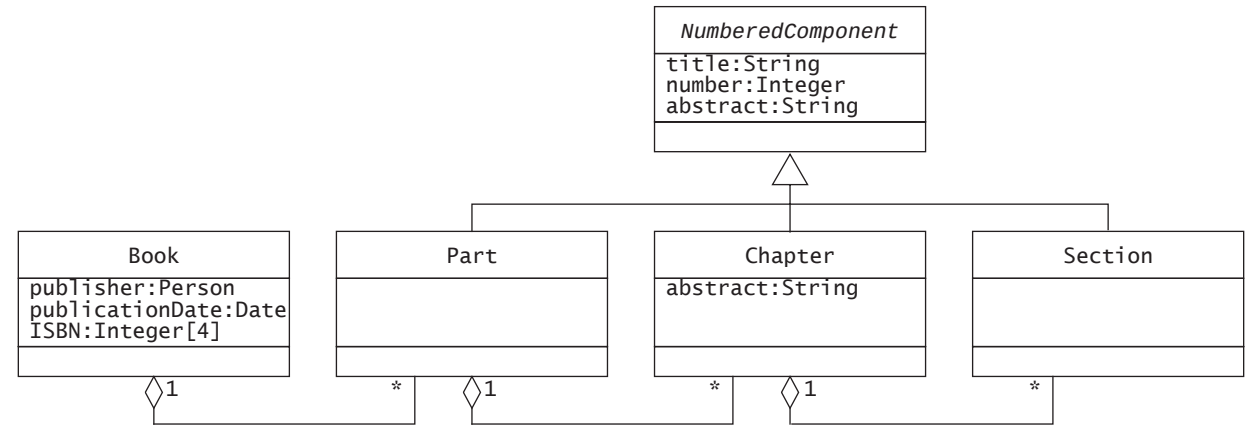


Figure 2-6 Example solution for Exercise 2–10.

This exercise checks the student's knowledge of abstract classes and inheritance.

2–11 Draw a class diagram representing the relationship between parents and children. Take into account that a person can have both a parent and a child. Annotate associations with roles and multiplicities.

Figure 2-7 depicts a canonical solution. This exercise is meant to emphasize the difference between a relationship, a role, and a class. In the above sentence, parent and child are roles while person is the class under consideration. This results in a class diagram with a single class and a single association with both ends to the class.

A common modeling mistake by novices is to draw two classes, one for the parent and one for the child.

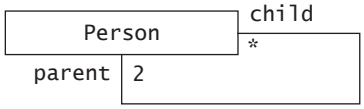


Figure 2-7 Example solution for Exercise 2–11.

2–12 Draw a class diagram for bibliographic references. Use the references in Appendix C, Bibliography, to test your class diagram. Your class diagram should be as detailed as possible.

The domain of bibliographic references is rich and complex. Consequently, students should deepened their understanding of the domain before they attempt to draw a class diagram (similar to the requirements analysis of a system). Figure 2-8 depicts an incomplete sample solution that could be accepted as sufficient from an instructor. The class diagram should minimally include the following concepts:

- An abstract class *`Publication`* (or *`BibliographicReference`*)
- A many to many relationship between `Author` and *`Publication`*
- At least three or more concrete classes refining *`Publication`*
- At least one aggregation relationship (e.g., between `Journal` and `Article` or `Proceedings` and `ConferencePaper`). Both ends of the aggregation should also be subclasses of *`Publication`*.

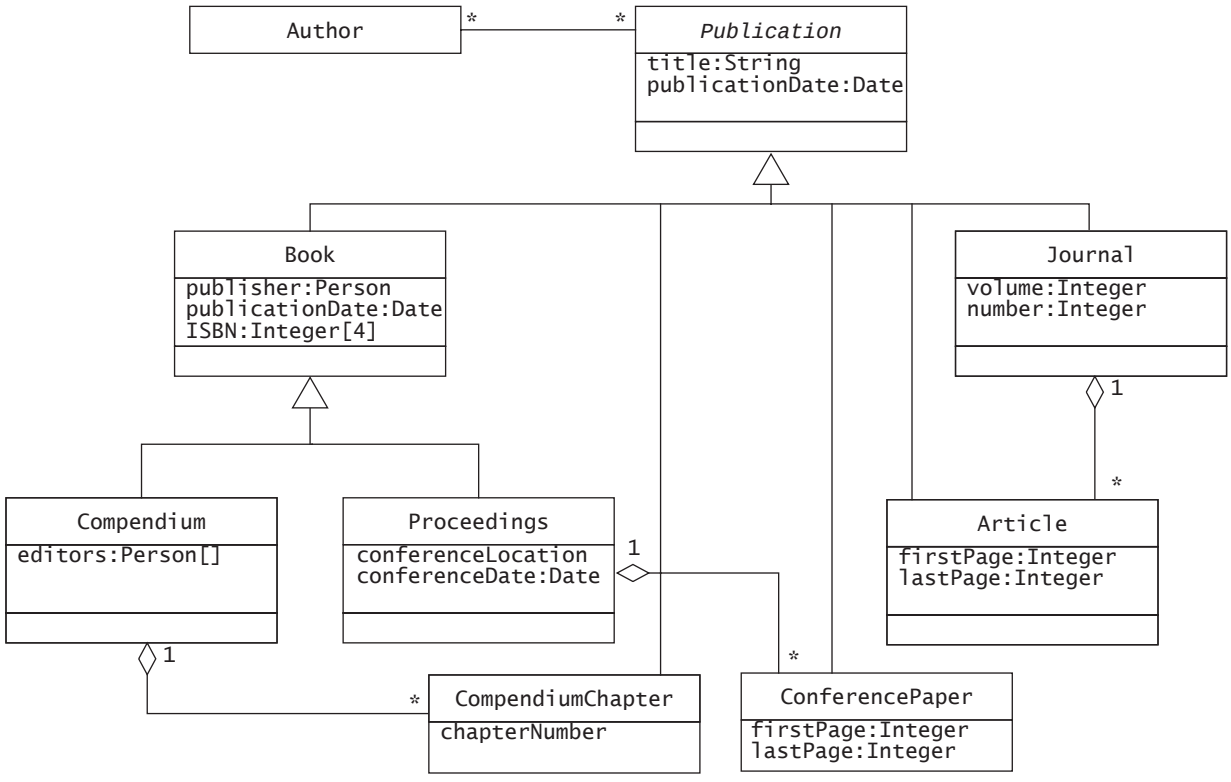


Figure 2-8 Example solution for Exercise 2–12.

2–13 Draw a sequence diagram for the `warehouseOnFire` *scenario of Figure 2-15. Include the objects* `bob`, `alice`, `john`, `FRIEND`, *and instances of other classes you may need. Draw only the first five message sends.*

This exercise checks the student's knowledge of sequence diagrams when using instances. This exercise can have several correct answers. In addition to the UML rules on sequence diagrams, all correct sequence diagrams for this exercise should include one or more actors on the left of the diagram who initiate the scenario, one or more objects in the center of the diagram which represent the system, and a dispatcher actor on the right of the diagram who is notified of the emergency. All actors and objects should be instances. Figure 2-9 depicts a possible answer.

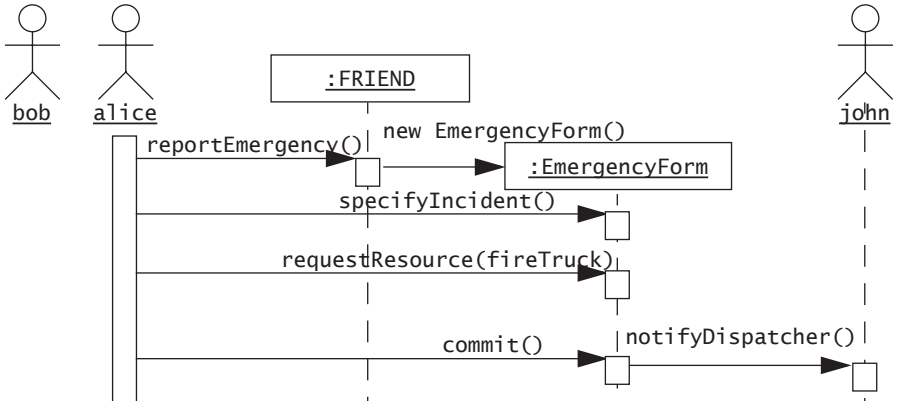


Figure 2-9 Example solution for Exercise 2–13.

2–14 Draw a sequence diagram for the `ReportIncident` use case of Figure 2-14. Draw only the first five message sends. Make sure it is consistent with the sequence diagram of Exercise 2–13.

This exercise tests the student's knowledge of sequence diagram when using classes. Like the previous exercise, this exercise can have several correct solutions. A correct solution should include a FieldOfficer actor on the left, a Dispatcher actor on the right, and one or more classes in the middle representing the FRIEND system. The answer to this exercise should be consistent with the answer to the previous exercise, in the sense that all class and operation names should be the same. Figure 2-10 depicts a possible answer.

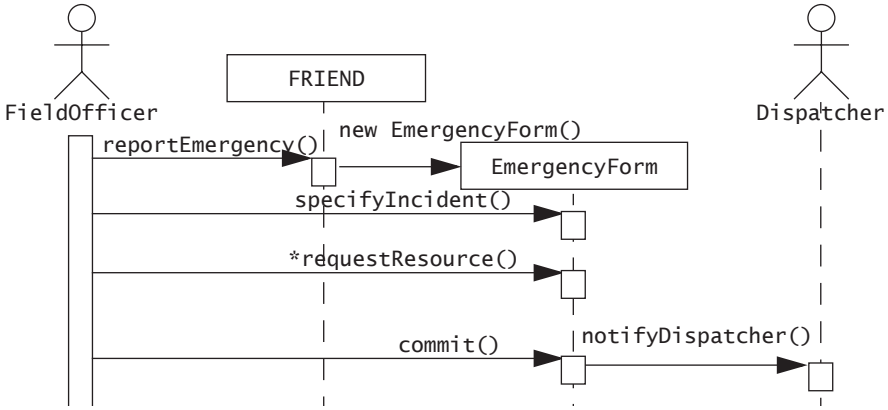


Figure 2-10 Example solution for Exercise 2–14.

2–15 Consider the process of ordering a pizza over the phone. Draw an activity diagram representing each step of the process, from the moment you pick up the phone to the point where you start eating the pizza. Do not represent any exceptions. Include activities that others need to perform.

Figure 2-11 is an example solution for this exercise. The following elements should be present in the solution:

- Activity names should be verb phrases indicating what the initiating actor is attempting to accomplish. Roles should be indicated with swimlanes

- Activities that are concurrent or which do not need to happen in a sequential order should be indicated with complex transitions.

2–16 Add exception handling to the activity diagram you developed in Exercise 2–15. Consider at least three exceptions (e.g., delivery person wrote down wrong address, deliver person brings wrong pizza, store out of anchovies).

This exercise checks the student's knowledge of decision points. Figure 2-12 depicts an example solution. Students modeling a realistic example will also think about cascaded exceptions and learn to represent them with multiple decision points.

2–17 Consider the software development activities which we described in Section 1.4. Draw an activity diagram depicting these activities, assuming they are executed strictly sequentially. Draw a second activity diagram depicting the same activities occurring incrementally (i.e., one part of the system is analyzed, designed, implemented, and tested completely before the next part of the system is developed). Draw a third activity diagram depicting the same activities occurring concurrently.

This exercise tests the student's knowledge of the activity diagram syntax, not the knowledge of software engineering activities. In the sample solution provided in Figure 2-13, we assumed that requirements elicitation need to be completed before any subsystem decomposition can be done. This leads to a splitting of the flow of control in the third diagram where all remaining activities occur concurrently. The instructor could accept a solution where requirements elicitation also occurs incrementally or concurrently with the other development activities.

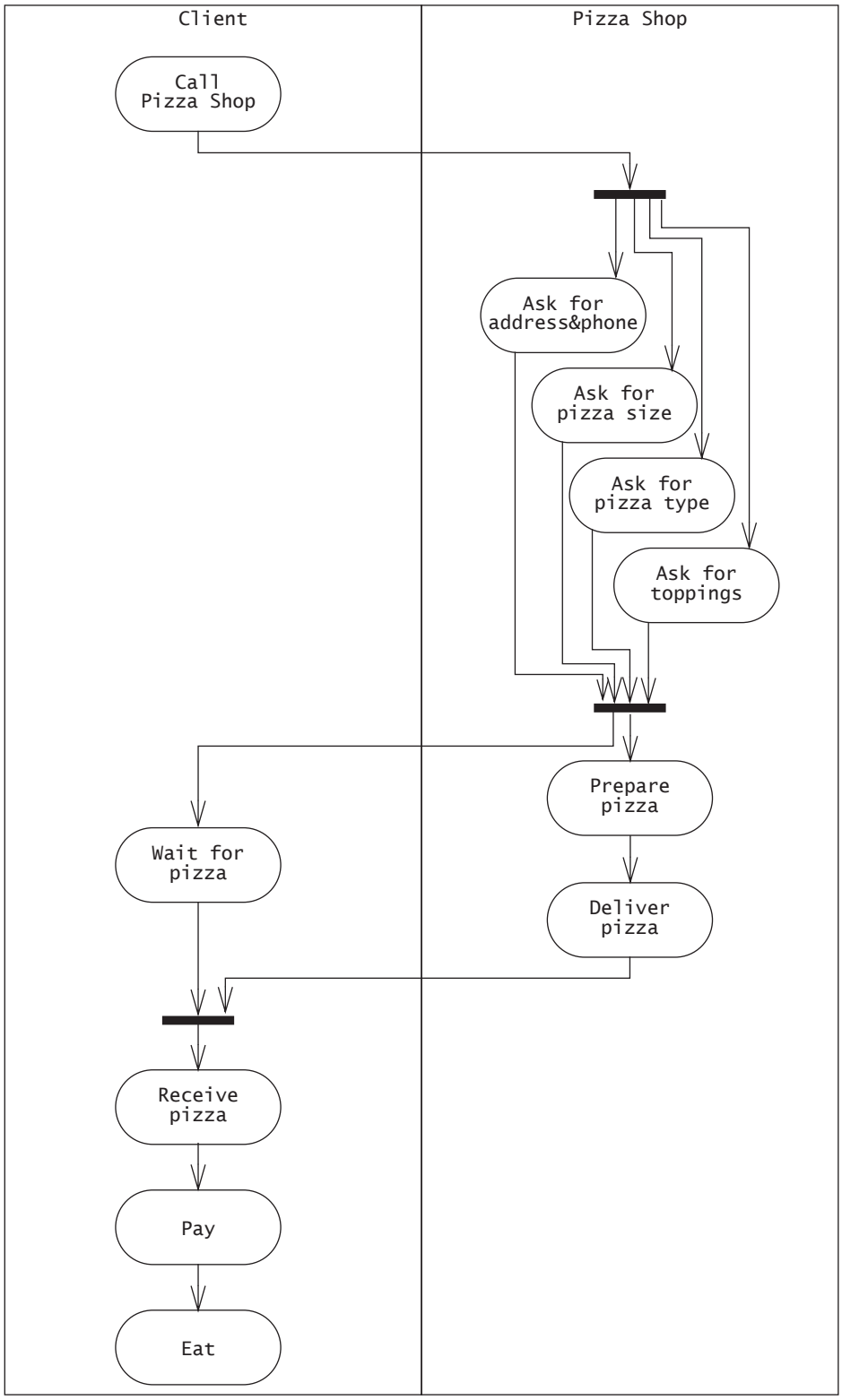


Figure 2-11 An example of pizza order activity diagram with exception handling.

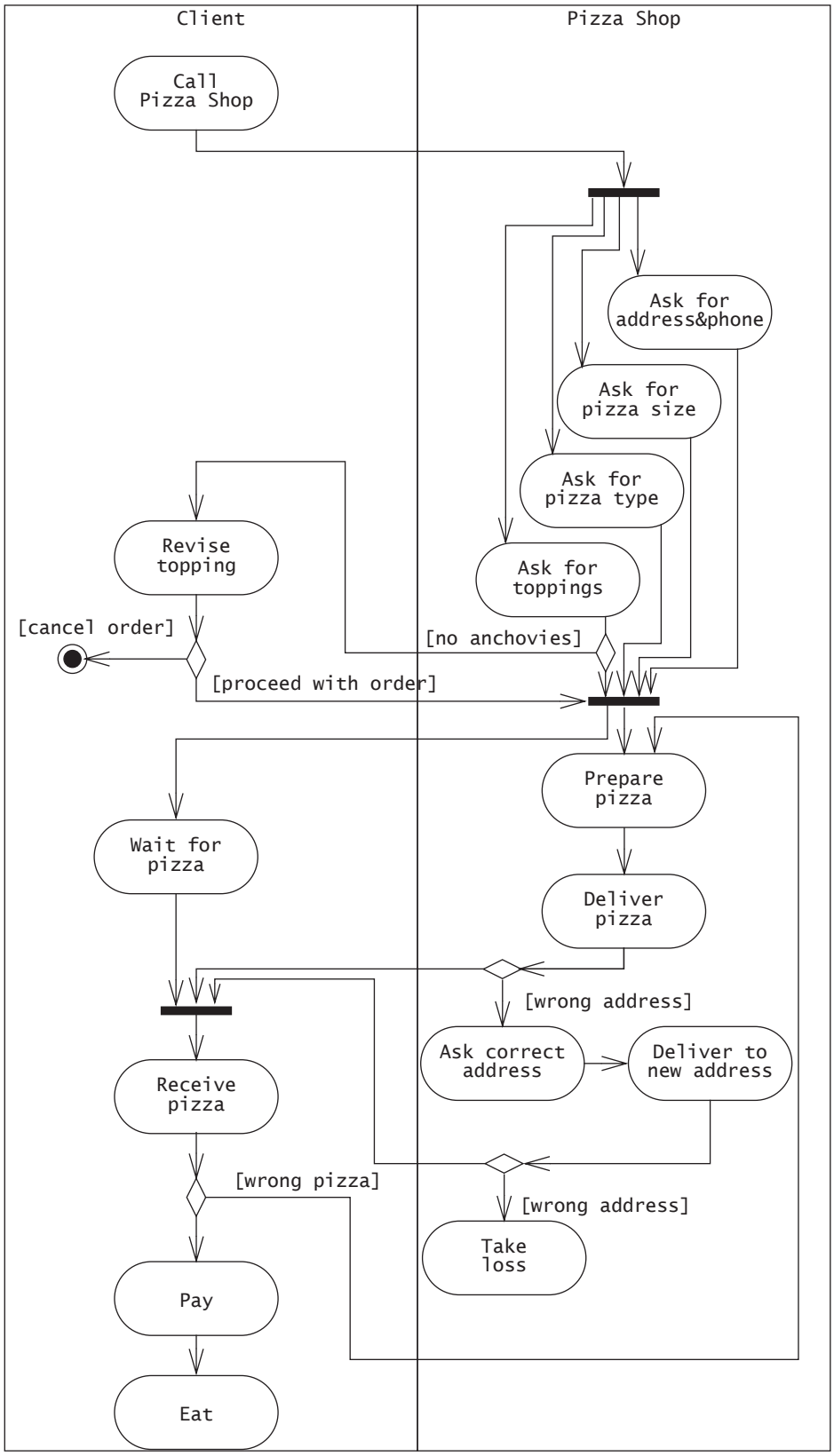


Figure 2-12 An example of pizza order activity diagram.

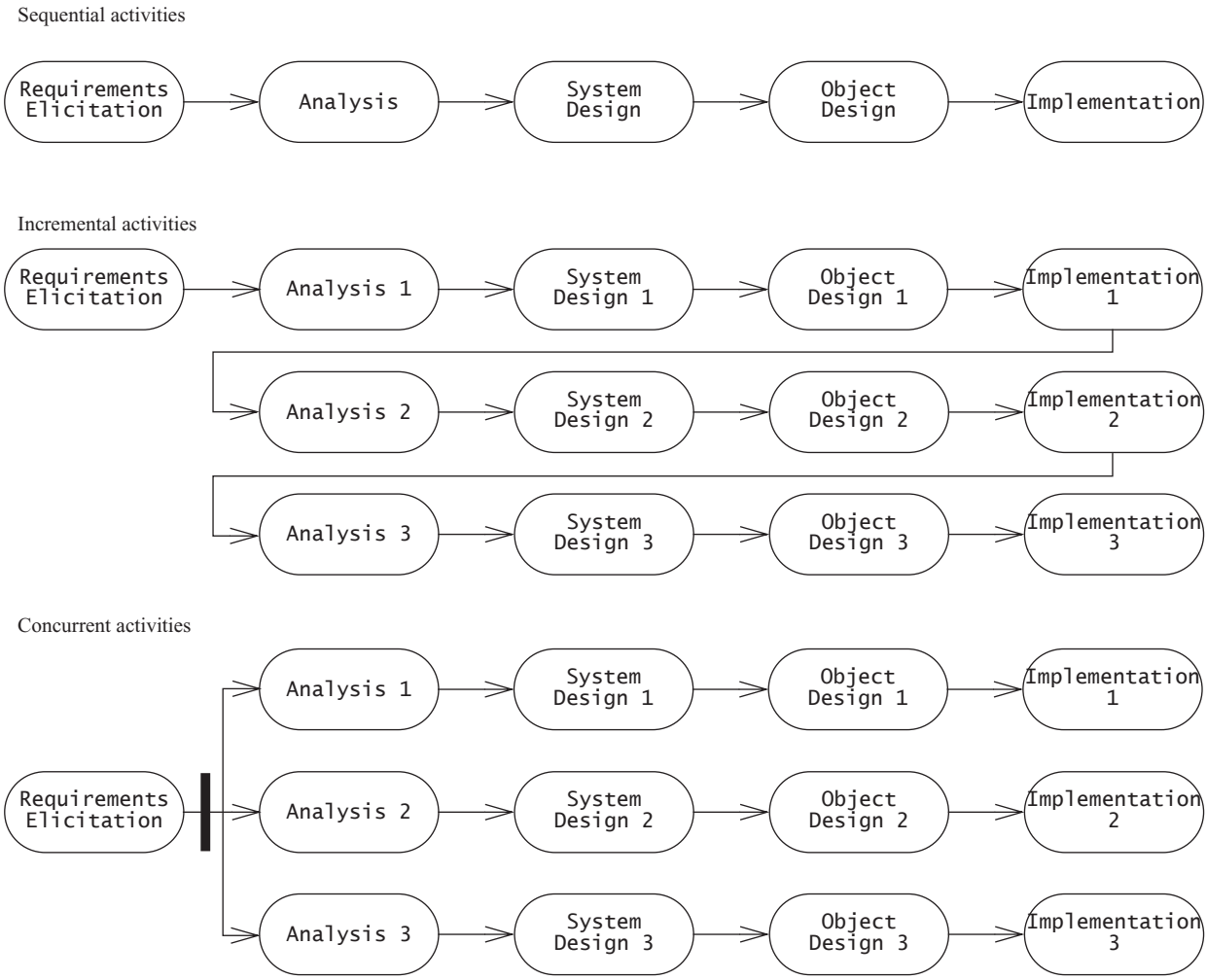


Figure 2-13 Example solution for Exercise 2–17.

3. Project Communication: Solutions

3-1 What is the difference between a role and a participant?

A participant is an individual. A role corresponds to a set of responsibilities. A single participant can assume many roles. Some roles may be shared among more than one participant.

3–2 Can a role be shared between two or more participants? Why or why not?

Some roles are assigned to more than one participant. For example, a team of developers can be assigned to the same subsystem. Other roles, such as chief architect or project manager, can only be assigned to a single participant in the project. In the end, the issue is one of assigning clear and non-overlapping tasks to single participants: When two or more participants are responsible for exactly the same task, none of them will feel personally responsible for the success of the task.

3–3 What is the difference between a client and an end user?

The client contracts the development of a system. The client is responsible for defining the scope of the system and for financing the project. The end users are the participants who actually uses the system to accomplish their work. In some project, the role of end user and client may be shared by the same participant. In general, however, this is not the case.

3–4 To which roles would you assign the following tasks?

- *Change a subsystem interface to accommodate a new requirement.* [system architect]
- *Communicate the subsystem interface change to other teams.* [API engineer (liaison)]
- *Change the documentation as a result of the interface change.* [editor]
- *Design a test suite to find defects introduced by the change.* [tester]
- *Ensure that the change is completed on schedule.* [project manager or team leader]

3–5 You are responsible for coordinating the development of a system for processing credit applications for a bank. In what roles would the following project participants be able to contribute most to the project?

- *a bank employee responsible for processing credit applications* [end user]
- *the manager of the information technology group at the bank, who contracted the system* [client]
- *a free-lancer who developed similar systems in the past* [developer]
- *a technical writer [editor]*

3–6 Draw a UML activity diagram representing the meeting process described in Section 3.4.3. Focus in particular on the work products generated before and after the meeting, such as the agenda and the meeting minutes. Use swimlanes to represent roles.

Figure 3-1 depicts an example solution. This exercise tests the student's knowledge of the meeting process and of activity diagrams described in Chapter 2.

3–7 What is the difference between a work package and a work product?
A work package is a description of work. A work product is the actual result of work (e.g., a document, a piece of code, etc.).
When is a work package defined?
During planning.
Consider an assignment where two students collaborate to plan and develop a system for sorting lists of names using two different sort algorithms. The deliverables for the assignment are the source code, the system documentation, and a manual for other developers explaining how new sorting algorithms can be integrated into the code. Give examples of work packages and work products in this project.

Examples of work packages:

- Plan development

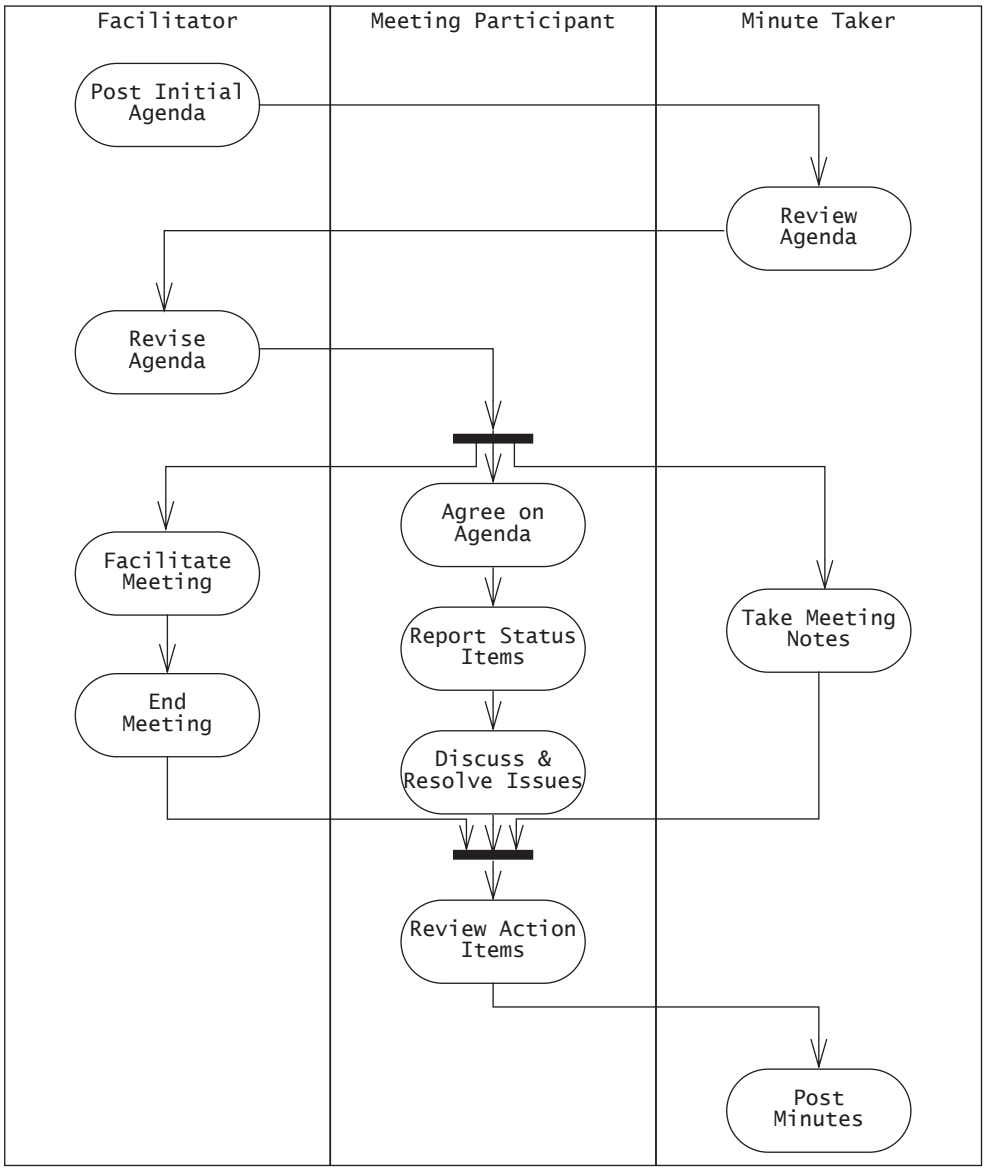


Figure 3-1 An example meeting process (UML activity diagram).

- Define and document interface between system and sort algorithms
- Implement algorithms
- Document system

Examples of work products:

- development plan
- source code
- system documentation
- manual for explaining how to integrate new sorting algorithms

3–8 What is the difference between a cross-functional team and a subsystem team? Provide examples and justify your choices.

A subsystem team is completely responsible for the development of a subsystem.A cross-functional team is made out of liaison of subsystem teams and is responsible for support tasks, such as documentation and configuration management. Also, the system architecture can is often defined by a cross-functional team made out of liaison from all subsystem teams.

3–9 As many critical communication events are planned (e.g., client reviews, project reviews, peer reviews), why is there still a need for unplanned communication events (e.g., request for clarification, request for change, issue resolution)?

There are many reasons for unplanned communication. Some of the more important ones are:

- Reduce latency: While developers could wait for the next project review to request clarification on their tasks or on interfaces provided by other teams, this would slow down development significantly and introduce much rework (e.g., correcting code that was developed under incorrect assumptions).
- React to unexpected events: Any system of a realistic size cannot be completely specified and understood a priori. This results in requirements and design changes during development, which often have to be communicated and negotiated among several stakeholders. While change processes can be set in place to manage this communication, much unplanned and informal communication complements the formal communication events to motivate, clarify, understand, gauge, and negotiate the underlying issues introduced by the change.
- Team building: Formal communication events are rarely conducive to a team atmosphere. It is during unplanned and informal events that participants get know each other and form a community working towards a common goal.

3–10 Select a random working day in your work week. Log all activities that qualify as communication activities (e.g., talking to friends over coffee, obtaining information from a fellow student, providing information, negotiating, advertising, browsing the web). Which fraction of your working day does communication represent?

Typically, the fraction of the day spent of communication can easily reach 50% or more. However, there will be a wide variance in the numbers reported by the students, depending on whether or not they are currently doing project work.

3–11 You are a member of the user interface team. You are responsible for designing and implementing forms collecting information about users of the system (e.g., first name, last name, address, E-mail address, level of expertise). The information you are collecting is stored in the database and used by the reporting subsystem. You are not sure which fields are required information and which are optional. How do you find out?

This is an open-ended question whose purpose is to have the student think of all the communication paths available. The side effect of this question is that the student will also have to think about the information sources associated with a project. A correct answer should contain at least two of the following points:

- The user interface team member should first check the requirements analysis document, described in Chapter 1.
- If the answer is not in the requirements analysis document, or if the answer is not clear, the document is incomplete or does not exist yet, in which case the user interface team member should ask the responsible analyst on the project. This is a request for change or a request for clarification.
- If the analyst does not have the answer, he or she should investigate with a potential user and possibly get approval from the client. This is also a request for clarification.
- In all cases, the answer should eventually make its way into the requirements analysis document.

3–12 You have been reassigned from the user interface team to the database team due to staff shortages and replanning. The implementation phase is well underway. In which role would you be most proficient given your knowledge of the user interface design and implementation?

Liaison of the database team to the user interface team.

3–13 Assume the development environment is Unix workstations, and the documentation team uses the Macintosh platform for writing documentation. The client requires the documents to be available on Windows platforms. Developers produce the design documentation using FrameMaker. The documentation team uses Microsoft Word for the user-level documentation. The client submits corrections on hardcopies and does not need to modify the delivered documents. How could the information flow between the developers, the technical writers, and the client be set up (e.g., format, tools, etc.) such that duplication of files is minimized while everybody's tool preferences and platform requirements are still satisfied?

This is a simple real life question with complicated answers. Several approaches are possible, including:

- The project uses a printable format as a standard interchange format, such as Postscript or PDF (Portable Display Format). This allows every participant to access and view documents. Only the author of a document is able to make changes, however. This makes it difficult to move the document responsibility to move from one team to another (e.g., from a subsystem team to the documentation team).
- The project uses an interchange format such as RTF or XML for all documents. Both formats are recognized by recent versions of both Microsoft Word and FrameMaker. This allows document responsibilities to be transferred between developers and technical writers. Unfortunately, this is currently not a practical solution as both tools understanding of the interchange formats is different and imperfect, resulting in some loss of formatting.

3–14 Which changes in the organization and communication infrastructure would you recommend for a successor of the Ariane 5 project as a consequence of the Ariane 501 failure described at the beginning of this chapter?

This is an open-ended question as many solutions exist and detailed knowledge of the current team organization are not available. Note that this question is only interested in communication or organizational changes, not technical or methodological changes. A correct answer could include the following points:

- Increase communication between the team responsible for system testing and the software developers. This can be done by introducing liaisons on both teams.
- Increase communication between the team responsible for unit testing and for system testing.
- Change documentation requirements for reused components.

4. Requirements Elicitation: Solutions

4-1 Consider your watch as a system and set the time 2 minutes ahead. Write down each interaction between you and your watch as a scenario. Record all interactions, including any feedback the watch provides you.

This scenario could be written using any watch. Below is an example of a watch with a digital display and two buttons. Any solution should be detailed enough that an ignorant user can execute the scenario with somebody else's watch.

Scenario name	`setWatch2MinutesAhead`
Participating actor instances	`allen:WatchOwner`
Flow of events	1. The `WatchOwner` presses both `Watch` buttons simultaneously. 2. The `Watch` enters "set time" mode and indicates this by blinking the hour digits. 3. The `WatchOwner` presses the left button once. 4. The `Watch` stops blinking the hour digits and starts blinking the minutes. 5. The `WatchOwner` presses the right button twice. 6. The `Watch` increments the minutes by two. 7. The `WatchOwner` presses both buttons simultaneously. 8. The `Watch` stops blinking.

Figure 4-1 The `setWatch2MinutesAhead` scenario.

4–2 Consider the scenario you wrote in Exercise 4-1 Identify the actor of the scenario. Next, write the corresponding use case `SetTime`*. Include all cases, and include setting the time forward, backward, setting hours, minutes, and seconds.*

The following use case is generalized from the scenario of Figure 4-1. The level of detail of the use case should be the same as the level of detail of the scenario in the previous exercise.

Use case name	`SetTime`
Participating actor	Initiated by `WatchOwner`
Entry condition	1. The `Watch` is in "read time" mode.
Flow of events	2. The `WatchOwner` presses both `Watch` buttons simultaneously. 3. The `Watch` enters "set hour" mode and indicates this by blinking the hour digits. 4. In the "set hour" mode, the `WatchOwner` can advance the hour digits by pressing the right button. Each time the right button is pressed, the hours are incremented by one. If the hours reach 23 and the right button is pressed, the hours are reset to 0. The date is not changed. 5. At any time in the "set hour" mode, the `WatchOwner` can switch to the "set minutes" mode by selecting the left button. To indicate this, the `Watch` stops blinking the hour digits and starts blinking the minute digits. 6. In the "set minutes" mode, the `WatchOwner` can advance the minute digits by pressing the right button. Each time the right button is pressed, the minutes are incremented by one. If the minutes reach 59 and the right button is pressed, the minutes are reset to 0. The hours are not changed.

Figure 4-2 `SetTime` use case.

	7. At any time in the "set minutes" mode, the `WatchOwner` can switch to the "set seconds" mode by selecting the left button. To indicate this, the `Watch` stops blinking the minute digits and starts blinking the second digits. 8. In the "set seconds" mode, the `WatchOwner` can advance the second digits by pressing the right button. Each time the right button is pressed, the seconds are incremented by one. If the seconds reach 59 and the right button is pressed, the seconds are rest to 0. The hours and the minute digits are not changed. 9. At any time in the "set seconds" mode, the `WatchOwner` can switch to the "set hours" mode by pressing the left button. To indicate this, the `Watch` stops blinking the second digits and starts blinking the hour digits. 10. At any time in the "set hours," "set minutes," and "set seconds" mode, the `WatchOwner` can return to "read time" mode by pressing both `Watch` buttons simultaneously.
Exit condition	11. The `Watch` is in "read time" mode with the new time set.
Special requirements	None.

Figure 4-2 `SetTime` use case.

4–3 Assume the watch system you described in Exercises 4-1 and 4–2 also supports an alarm feature. Describe setting the alarm time as a self-contained use case named `SetAlarmTime`.

The difference between setting the time of a watch and setting the alarm time should be minimal. A typical answer can be produced by copying and pasting solution of Exercise 4–2 and modifying the beginning and the end of the use case. The rationale behind this similarity is that any user interface should provide similar interfaces for addressing similar functionality (i.e., user interface consistency). Below is an example generated from the use case of Figure 4-2.

Use case name	`SetAlarmTime`
Participating actor	Initiated by `WatchOwner`
Entry condition	1. The `Watch` is in "read time" mode.
Flow of events	2. The `WatchOwner` presses both `Watch` buttons simultaneously. The `Watch` enters the "set hour" mode. The `WatchOwner` continues to press both buttons. After one second, a small alarm bell symbol appears on the display and blinks, indicating that the hours of the alarm are being set (as opposed to the normal time). 3. In the "set hour" mode, the `WatchOwner` can advance the hour digits by pressing the right button. Each time the right button is pressed, the hours are incremented by one. If the hours reach 23 and the right button is pressed, the hours are reset to 0. The date is not changed. 4. At any time in the "set hour" mode, the `WatchOwner` can switch to the "set minutes" mode by selecting the left button. To indicate this, the `Watch` stops blinking the hour digits and starts blinking the minute digits. 5. In the "set minutes" mode, the `WatchOwner` can advance the minute digits by pressing the right button. Each time the right button is pressed, the minutes are incremented by one. If the minutes reach 59 and the right button is pressed, the minutes are reset to 0. The hours are not changed. 6. At any time in the "set minutes" mode, the `WatchOwner` can switch to the "set seconds" mode by selecting the left button. To indicate this, the `Watch` stops blinking the minute digits and starts blinking the second digits. 7. In the "set seconds" mode, the `WatchOwner` can advance the second digits by pressing the right button. Each time the right button is pressed, the seconds are incremented by one. If the seconds reach 59 and the right button is pressed, the seconds are rest to 0. The hours and the minute digits are not changed.

Figure 4-3 `SetAlarmTime` use case.

	8. At any time in the "set seconds" mode, the `WatchOwner` can switch to the "set hours" mode by pressing the left button. To indicate this, the `Watch` stops blinking the second digits and starts blinking the hour digits. 9. At any time in the "set hours," "set minutes," and "set seconds" mode, the `WatchOwner` can return to "read time" mode by pressing both `Watch` buttons simultaneously. The digits stop blinking and the alarm bell stays on and stops blinking.
Exit condition	10. The `Watch` is in "read time" mode with the new alarm time set.
Special requirements	None.

Figure 4-3 `SetAlarmTime` use case.

4–4 Examine the `SetTime` and `SetAlarmTime` use cases you wrote in Exercises 4–2 and 4–3 Eliminate any redundancy by using an include relationship. Justify why an include relationship is preferable to an extend relationship in this case.

The solution to this exercise should include three use cases: `SetTime`, `SetAlarmTime`, and a third use case (which we called `SpecifyTime`) made out of the common parts of `SetTime` and `SetAlarmTime`. The essential point is to factor out as much common functionality as possible into the third use case. The rationale is that, by eliminating redundancy, fewer inconsistencies are introduced in the future when the use cases are modified. Below is a sample solution for this exercise.

Use case name	`SetTime`
Participating actor	Initiated by `WatchOwner`
Entry condition	1. The `Watch` is in "read time" mode.
Flow of events	2. The `WatchOwner` presses both `Watch` buttons simultaneously. The `Watch` enters the "set hour" mode and indicates this by blinking the hour digits. 3. The SpecifyTime is included here. A the end of the SpecifyTime use case, the `WatchOwner` has specified a time. 4. The `WatchOwner` presses both buttons simultaneously to return to the "read time" mode. The new time is set.
Exit condition	5. The `Watch` is in "read time" mode with the new time set.
Special requirements	None.

Figure 4-4 `SetTime` use case.

Use case name	`SetAlarmTime`
Participating actor	Initiated by `WatchOwner`
Entry condition	1. The `Watch` is in "read time" mode.

Figure 4-5 `SetAlarmTime` use case.

Flow of events	2. The `WatchOwner` presses both `Watch` buttons simultaneously. The `Watch` enters the "set hour" mode and indicates this by blinking the hour digits. The `Watch` enters the "set hour" mode. The `WatchOwner` continues to press both buttons. After one second, a small alarm bell symbol appears on the display and blinks, indicating that the hours of the alarm are being set (as opposed to the normal time). 3. The SpecifyTime is included here. A the end of the SpecifyTime use case, the `WatchOwner` has specified a time for the alarm. 4. The `WatchOwner` presses both buttons simultaneously to return to the "read time" mode. The new alarm time is set.
Exit condition	5. The `Watch` is in "read time" mode with the new alarm time set.
Special requirements	None.

Figure 4-5 `SetAlarmTime` use case.

Use case name	`SpecifyTime`
Participating actor	Initiated by `WatchOwner`
Entry condition	1. The `Watch` is in "set alarm" or "set time" mode. The hour digits are blinking.
Flow of events	2. In the "set hour" mode, the `WatchOwner` can advance the hour digits by pressing the right button. Each time the right button is pressed, the hours are incremented by one. If the hours reach 23 and the right button is pressed, the hours are reset to 0. The date is not changed. 3. At any time in the "set hour" mode, the `WatchOwner` can switch to the "set minutes" mode by selecting the left button. To indicate this, the `Watch` stops blinking the hour digits and starts blinking the minute digits. 4. In the "set minutes" mode, the `WatchOwner` can advance the minute digits by pressing the right button. Each time the right button is pressed, the minutes are incremented by one. If the minutes reach 59 and the right button is pressed, the minutes are reset to 0. The hours are not changed. 5. At any time in the "set minutes" mode, the `WatchOwner` can switch to the "set seconds" mode by selecting the left button. To indicate this, the `Watch` stops blinking the minute digits and starts blinking the second digits. 6. In the "set seconds" mode, the `WatchOwner` can advance the second digits by pressing the right button. Each time the right button is pressed, the seconds are incremented by one. If the seconds reach 59 and the right button is pressed, the seconds are rest to 0. The hours and the minute digits are not changed. 7. At any time in the "set seconds" mode, the `WatchOwner` can switch to the "set hours" mode by pressing the left button. To indicate this, the `Watch` stops blinking the second digits and starts blinking the hour digits. 8. At any time in the "set hours," "set minutes," and "set seconds" mode, the `WatchOwner` can return to "read time" mode by pressing both `Watch` buttons simultaneously. The digits stop blinking and the alarm bell stays on and stops blinking.
Exit condition	9. The `Watch` is in "read time" mode.
Special requirements	None.

Figure 4-6 `SpecifyTime` use case.

4–5 Assume the `FieldOfficer` can invoke a Help feature when filling an `EmergencyReport`. The `HelpReportEmergency` feature provides a detailed description for each field and specifies which fields are required. Modify the `ReportEmergency` use case (described in Figure 4-10) to include this help functionality. Which relationship should you use to relate the `ReportEmergency` and `HelpReportEmergency`?

The ReportEmergency use case does not need to be modified when the HelpReportEmergency is written as an extension. The advantage of writing behavior such as help and exceptions as extensions is that they can be reused in different use cases or in different steps of a single use case. Below is a sample solution with the `<<extend>>` relationship.

Use case name	`HelpReportEmergency`
Participating actor	Initiated by `FieldOfficer`
Entry condition	1. The Emergency Report Form is visible on the `FieldOfficer`'s screen and has not been submitted yet. This use case extends steps 2 and 3 of ReportEmergency.
Flow of events	2. The `FieldOfficer` selects a field of the `Emergency Report Form` (one of "emergency type," "location," "incident description," "resource request," and "hazardous material") and presses the help key. 3. `FRIEND` displays in a separate window an explanatory text describing whether the field is required or not, what range of values is possible, and examples for typical cases. 4. The `FieldOfficer` may return to the form without closing the help window. For terminals with large enough screens, the `FieldOfficer` can display both the form and the help window side by side.
Exit condition	5. The use case ends when the `FieldOfficer` closes the help window.
Special requirements	None.

Figure 4-7 `HelpReportEmergency` use case.

Location	*Use case description*
Field Officer station	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal. 2. FRIEND responds by presenting a form to the officer. The form includes an emergency type menu (general emergency, fire, transportation), a location, incident description, resource request, and hazardous material fields. 3. The `FieldOfficer` fills the form by specifying minimally the emergency type and description fields. The `FieldOfficer` may also describe possible responses to the emergency situation and request specific resources. Once the form is completed, the `FieldOfficer` submits the form by pressing the "Send Report" button, at which point the `Dispatcher` is notified.
Dispatcher station	4. The `Dispatcher` is notified of a new incident report by a popup dialog. The `Dispatcher` reviews the submitted information and creates an `Incident` in the database by invoking the `OpenIncident` use case. All the information contained in the `FieldOfficer`'s form is automatically included in the `Incident`. The `Dispatcher` selects a response by allocating resources to the `Incident` (with the `AllocateResources` use case) and acknowledges the emergency report by sending a short message to the `FieldOfficer`.
Field Officer station	5. The `FieldOfficer` receives the acknowledgment and the selected response.

Figure 4-8 Refined description for the `ReportEmergency` use case.

4–6 Below are examples of nonfunctional requirements. Specify which of these requirements are verifiable and which are not.

- *"The system must be usable."* [not verifiable, no precise definition of "usable"]
- *"The system must provide visual feedback to the user within 1 second of issuing a command."*[verifiable]
- *"The availability of the system must be above 95%."* [verifiable, assuming sufficient resources]
- *"The user interface of the new system should be similar enough to the old system such that users familiar*

with the old system can be easily trained to use the new system." [not verifiable, no precise definition of "easily trained"]

4–7 The need for developing a complete specification may encourage an analyst to write detailed and lengthy documents. Which competing quality of specification (see Table 4-1) may encourage an analyst to keep the specification short?

Consistency. Long documents tend to have much redundancy and are difficult to maintain.

4–8 Maintaining traceability during requirements and subsequent activities is expensive, because of the addition information that must be captured and maintained. What are the benefits of traceability that outweigh this overhead?

- Ability to show that all requirements have been implemented
- Ability to show that all features in the system correspond to a requirement
- Ability to assess the impact of a change
- Ability to track the human source of a requirement

Which of those benefits are directly beneficial to the analyst?

- The last two benefits are beneficial to the analyst as it reduces the effort to change the specification without introducing new errors.

4–9 Explain why multiple-choice questionnaires, as a primary means of extracting information from the user, is not effective for eliciting requirements.

Multiple-choice questionnaires provide possible answers to the questions that are asked. This introduces two problems: first, the analyst must be familiar enough with the application domain to offer a good set of answers for each question. Second, the user is constrained to select these pre-designed answers. It is then preferable that the analyst use other elicitation methods, such as task analysis or unstructured interviews, to gather sufficient knowledge of the application domain and to discover implicit knowledge that the user assumes everybody has. Subsequently, the analyst can use multiple choice questionnaires to confirm a hypothesis or to prioritize certain functionality.

4–10 From your point of view, describe the strengths and weaknesses of users during the requirements elicitation activity. Describe also the strengths and weaknesses of developers during the requirements elicitation activity.

This is an open-ended question. The goal is to have students realize that no one person has the complete problem/solution under control and that it takes much work to gather all available sources of information.

Strengths of users:

- they usually have detailed knowledge of the problem that needs to be solved
- they have detailed knowledge of constraints imposed by the environment on the possible solutions
- they have detailed knowledge of the application domain.

Weaknesses of users:

- they usually have poor knowledge of the possible solutions.
- they usually have poor knowledge of the formal languages for describing the problem or the solutions.

Strengths of developers:

- they have detailed knowledge of different possible solutions
- they have detailed knowledge of formal languages to describe the problem or the possible solutions

Weaknesses of developers:

- they (initially) have poor knowledge of the problem to be solved
- they can make incorrect assumptions about the problem based on their prior knowledge of different problems.

4–11 Briefly define the term "menu." Write your answer on a piece of paper and put it upside down on the table together with the definitions of four other students. Compare all five definitions and discuss any substantial difference.

The goal of this exercise is to make students realize the ambiguity of natural language. The instructor can replace the term "menu" by any other computer-oriented terms whose original meaning is still in use.

4–12 Write the high-level use case `ManageAdvertisement` *initiated by the* `Advertiser`*, and write detailed use cases refining this high-level use case. Consider features that enable an* `Advertiser` *to upload advertisement banners, to associate keywords with each banner, to subscribe to notices about new tournaments in specific leagues or games, and to monitor the charges and payments made on the advertisement account. Make sure that your use cases are also consistent with the* `ARENA` *problem statement provided in Figure 4-17.*

Note: For all exercises involving writing use cases, the instructor could issue prior to the exercise writing guidelines that the students should comply with. This will reduce the variance in the length, detail, and style of use cases, making it easier to grade the exercise.

Figure 4-9 depicts a use case diagram illustrating a possible decomposition into a high-level use case and three detailed use cases for ManageAdvertisement.

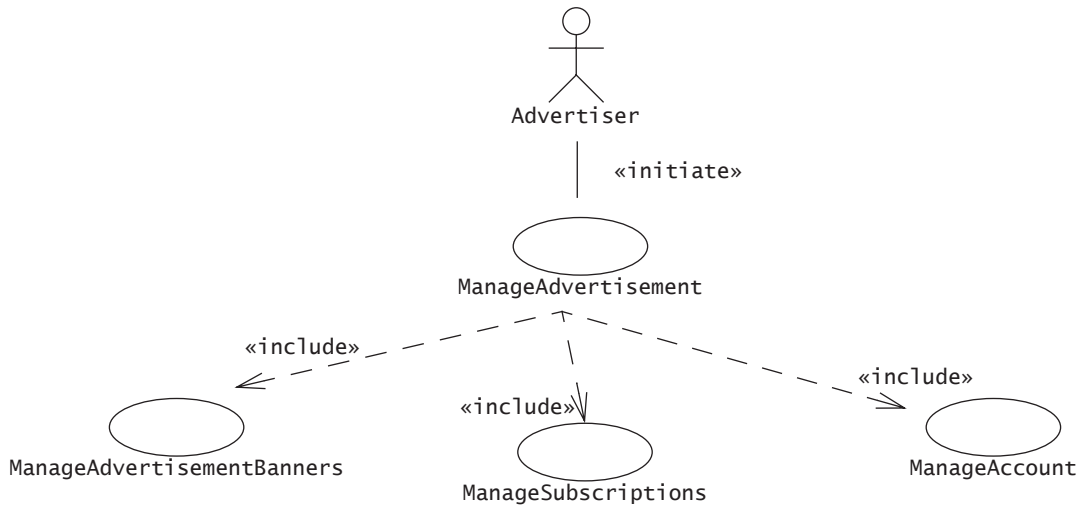


Figure 4-9 Detailed use cases refining the `ManageAdvertisement` high-level use case.

Use case name	`ManageAdvertisement`
Participating actors	Initiated by `Advertiser`
Flow of events	1. The `Advertiser` may upload any number of `AdvertisementBanners`, associate keywords with them, and remove previously uploaded `AdvertisementBanners`. (include use case `ManageAdvertisementBanners`). 2. `ARENA` displays the current state of the `Advertiser's` account.

Figure 4-10 An example of a high-level the `ManageAdvertisement` use case.

	3. The `Advertiser` may examine the details associated with his account and any charges or payments to the account. (include use case `ManageAccount`).	
		4. Based on the added advertisement banners and keywords, `ARENA` suggests leagues and games the `Advertiser` may subscribe to.
	5. The `Advertiser` may change his subscriptions to leagues and games (include use case `ManageSubscriptions`).	
Entry condition	• The `Advertiser` is logged into ARENA.	
Exit conditions	• New matches can use any newly uploaded banners that match the specified keywords. • The `Advertiser` is notified about any new tournaments in the specified leagues or games.	

Figure 4-10 An example of a high-level the `ManageAdvertisement` use case.

Figure 4-10 depicts a possible high-level use case for `ManageAdvertisement`. This use case should be relatively short and simply include the detailed use cases. Details such as the attributes of the forms involved should not appear at this level.

Figure 4-11 is an example of detailed use case for `ManageAdvertisementBanners`. There are many different other possibilities given the lack of details given in the exercise. Key features should include steps to add, remove, and modify banners, entry, and exit conditions. The use case should specify the attributes of advertisement banners that the `Advertiser` can change.

4–13 Considering the `AnnounceTournament` use case in Figure 4-24, write the event flow, entry conditions, and exit conditions for the use case `ApplyForTournament`, initiated by a `Player` interested in participating in the newly created tournament. Consider also the `ARENA` problem statement provided in Figure 4-17. Write a list of questions for the client when you encounter any alternative.

Figure 4-12 depicts a sample solution. There are, again, many different possibilities depending on the imagination of the students. The key features of a solution include system steps that check for a number of conditions which would make the application fail, and describe the corresponding steps. Given that this use case is relatively short, these actions should be described in the use case as opposed to using extends relationships and many one step use cases. The questions raised by the students should focus on when and how a Player is admitted into (and, correspondingly, thrown out of) a league. Students could also ask about alternative ways for handling too many applications, such as waiting lists, Players taking turns playing in different tournaments, and so on.

4–14 Write the event flows, entry conditions, and exit conditions for the exceptional use cases for the `AnnounceTournament` depicted in Figure 4-25. Use include relationships if necessary to remove redundancy.

This exercise should result in five or more short use cases describing a condition under which the exception occurs and a flow of events for handling it. Figure 4-13 is an example for one such exceptions. The student will discover that exceptional use cases should be used for complex cases, while trivial cases (e.g., name in use, maximum number of tournaments exceeded) could be either omitted or described within the invoking use case. This should to the realization that writing use cases is different than writing code, as the counterpart is the user and the client, not the computer or other developers.

Use case name	`ManageAdvertisementBanners`
Participating actors	Initiated by `Advertiser`
	1. The `Advertiser` requests the advertisement banner area.
	2. `ARENA` displays the list of all banners owned by this actor. The `Advertiser` has the option of uploading new banners (step 3.), editing existing banners (step 7.), or removing banners (step 11).
Flow of events	3. If the `Advertiser` requests the uploading of a new advertisement banner,
	4. `ARENA` checks if the `Advertiser` exceeded his quote. If not, `ARENA` presents the `Advertiser` with a form.
	5. The `Advertiser` specifies an image, a set of keywords, start and end dates, and a maximum frequency. The `Advertiser` can also specify that the banner can appear in any tournament.
	6. `ARENA` updates the list of advertisement banners that can be served for each tournament and returns to step 2.
	7. If the `Advertiser` selects an existing banner for editing,
	8. `ARENA` displays all the details associated with the banner (keywords, start and end dates, maximum frequency)
	9. The `Advertiser` specifies new parameters associated with the selected banners.
	10. `ARENA` updates the list of advertisement banners that can be served for each Tournament and returns to step 2.
	11. If the `Advertiser` selects to view the details associated with his account,
	12. `ARENA` displays the current total along with a list of recent charges and payments, including flat fees charged for exclusive sponsorships and charges for each individual banners, after which the `Advertiser` can select to go back to step 1.
Entry condition	• The `Advertiser` is logged into `ARENA`.
Exit conditions	• New advertisement banners can appear in new matches. • Old banners are retired immediately.

Figure 4-11 An example of a high-level the `ManageAdvertisement` use case.

Use case name	`ApplyForTournament`
Participating actors	Initiated by `Player`
Flow of events	1. The `Player` requests an application for an announced tournament 2. `ARENA` checks if the maximum number of applications for this tournament has been reached. If yes, the `Player` is notified that the tournament is full. 3. Otherwise, `ARENA` checks if the player is already registered with the League. If yes, the `Player` is informed that his application was successful, that he will be notified once the matches are scheduled, and the use case completes. 4. Otherwise, `ARENA` checks if the `Player` expertise rating falls within the bounds of the tournament. If so, the `Player` is automatically added to the league, and the `Player` is informed about the acceptance to the league and the tournament, and that he will be notified once the matches are scheduled. 5. Otherwise, the `Player` is informed why he cannot be admitted into the League.
Entry condition	• The `Player` is logged into ARENA.
Exit conditions	• The player is accepted in the tournament OR • The player is informed why the application failed.

Figure 4-12 `ApplyForTournament` use case.

Use case name	`NameInUse`
Participating actors	Initiated by `Player`
Flow of events	1. `ARENA` informs the `LeagueOwner` that the maximum number of open tournament for this league has been exceeded and presents a list of all the open tournaments, sorted by date of completion 2. The `AnnounceTournament` use case completes.
Entry condition	• The maximum number of open tournaments in the current league has been exceeded and the `LeagueOwner` attempts to create a new tournament. • Extends `AnnounceTournament`.
Exit conditions	

Figure 4-13 `NameInUse` use case.

5. Analysis: Solutions

5-1 Consider a file system with a graphical user interface, such as Macintosh's Finder, Microsoft's Windows Explorer, or Linux's KDE. The following objects were identified from a use case describing how to copy a file from a floppy disk to a hard disk: `File`, `Icon`, `TrashCan`, `Folder`, `Disk`, `Pointer`. *Specify which are entity objects, which are boundary objects, and which are control objects.*

Entity objects: `File`, `Folder`, `Disk`
Boundary objects: `Icon`, `Pointer`, `TrashCan`
Control objects: none in this example.

5–2 Assuming the same file system as before, consider a scenario consisting of selecting a file on a floppy, dragging it to Folder and releasing the mouse. Identify and define at least one control object associated with this scenario.

The purpose of a control object is to encapsulate the behavior associated with a user level transaction. In this example, we identify a `CopyFile` control object, which is responsible for remembering the path of the original file, the path of the destination folder, checking if the file can be copied (access control and disk space), and to initiate the file copying.

5–3 Arrange the objects listed in Exercises 5-1 and 5–2 horizontally on a sequence diagram, the boundary objects to the left, then the control object you identified, and finally, the entity objects. Draw the sequence of interactions resulting from dropping the file into a folder. For now, ignore the exceptional cases.

Figure 5-1 depicts a possible solution to this exercise. The names and parameters of the operations may vary. The diagram, however, should at least contain the following elements:

- Two boundary objects, one for the file being copied, and one of the destination folder.
- At least one control object remembering the source and destination of the copy, and possibly checking for access rights.
- Two entity objects, one for the file being copied, and one of the destination folder.

In this specific solution, we did not focus on the `Disk`, `Pointer`, and `TrashCan` objects. The Disk object would be added to the sequence when checking if there is available space. The TrashCan object is needed for scenarios in which Files or Folders are deleted.

Note that the interaction among boundary objects can be complex, depending on the user interface components that are used. This sequence diagram, however, only describes user level behavior and should not go into such details. As a result, the sequence diagram depicts a high level view of the interactions between these object, not the actual sequence of message sends that occurs in the delivered system.

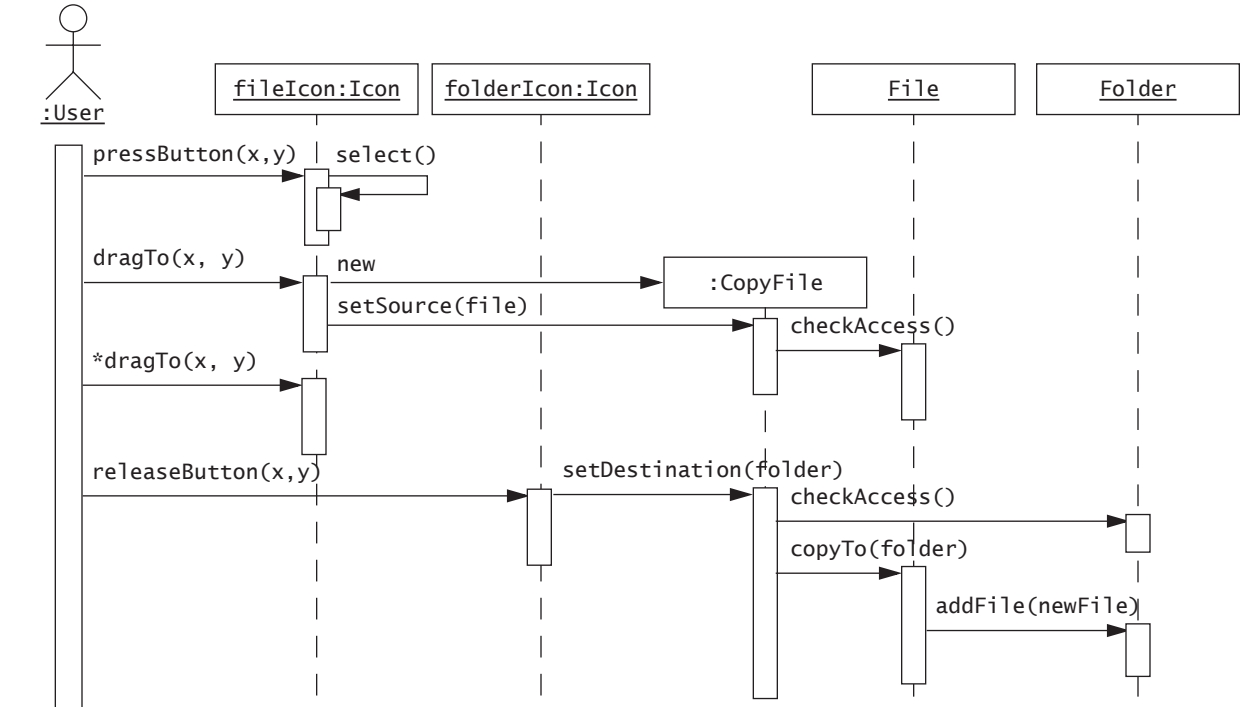


Figure 5-1 Sample solution for Exercise 5–3

5–4 Examining the sequence diagram you produced in Exercise 5–4, identify the associations between these objects.

Figure 5-2 is a possible solution for this exercises. We introduced an Item abstract class such that we can represent common associations to `Files` and `Folders`. A consequence of this decision would be to modify the sequence diagram of Figure 5-1 to use the Item class. The `CopyFile` control object has an association to the source of the `Copy`, which can be either a `File` or a `Folder`. The destination of a copy is always a `Folder`. The Icon boundary object has an association to the Item it represents and to the control object responsible for the copy.

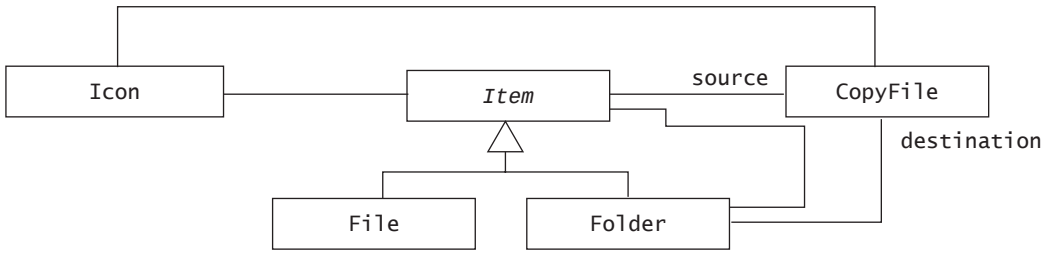


Figure 5-2 Sample solution for Exercise 5–4

5–5 Identify the attributes of each object that are relevant to this scenario (copying a file from a floppy disk to a hard disk). Also consider the exception cases "There is already a file with that name in the folder" and "There is no more space on disk."

The scenario implies that icons have positions (since they can be moved) and that they can be selected. Each `Item`, `Folder` or `File`, has a size attribute that is checked against the available space in the `Disk`. The exception "There is already a file with that name in the folder" implies that `Item` names are unique within a `Folder`. We represent this by adding a qualifier on the relationship between `Item` and `Folder`. Finally, when copying a file, we need to copy its contents, hence we add a contents attribute on the `File` object.

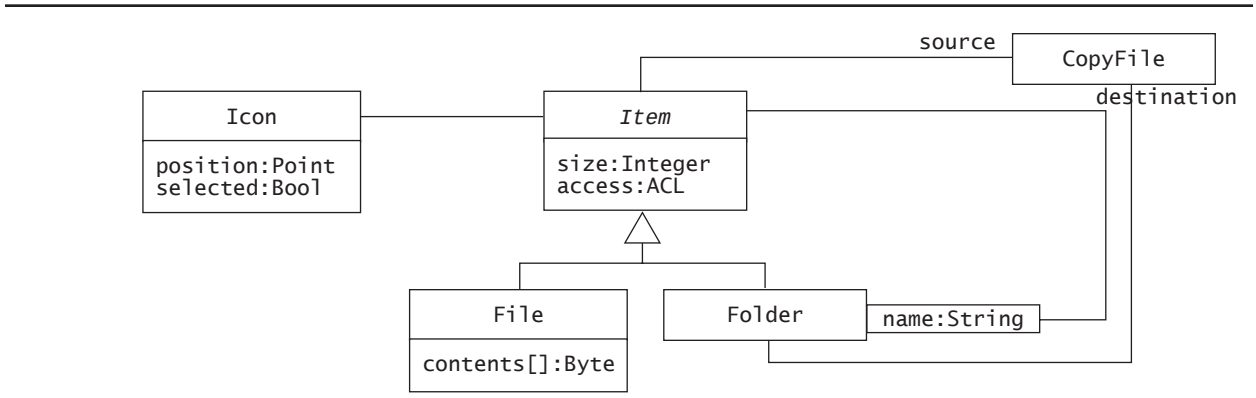


Figure 5-3 Sample solution for Exercise 5–5

5–6 Consider the object model in Figure 5-32 in the book (adapted from [Jackson, 1995]): Given your knowledge of the Gregorian calendar, list all the problems with this model. Modify it to correct each of them.

The problems with Figure 5-32 are related with the multiplicity of the associations. Weeks can straddle month boundaries. Moreover, the multiplicity on other associations can be tightened up: years are always composed of exactly twelve months, months do not straddle year boundaries, and weeks are always composed of seven days. Figure 5-4 depicts a possible revised model for this exercise.

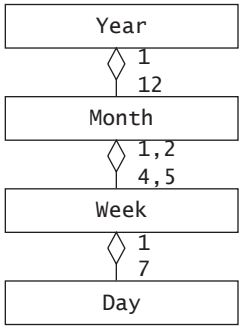


Figure 5-4 A naive model of the Gregorian calendar (UML class diagram).

5–7 Consider the object model of Figure 5-32 in the book. Using association multiplicity only, can you modify the model such that a developer unfamiliar with the Gregorian calendar could deduce the number of days in each month? Identify additional classes if necessary.

The purpose of this exercise is to show the limitation of association multiplicities. There is no complete solution to this problem. A partial solution indicating the number of days in each month is depicted in Figure 5-5. We created four abstract classes for each of the possible month lengths, 11 classes for each of the nonvariable months and two classes for the month of February. The part that cannot be resolved with association multiplicities is the definition of a leap year and that the number of days in February depends on whether the year is leap or not. In practice, this

problem can solved by using informal or OCL constraints, described in more detail in Chapter 9, Object Design: Specifying Interfaces.

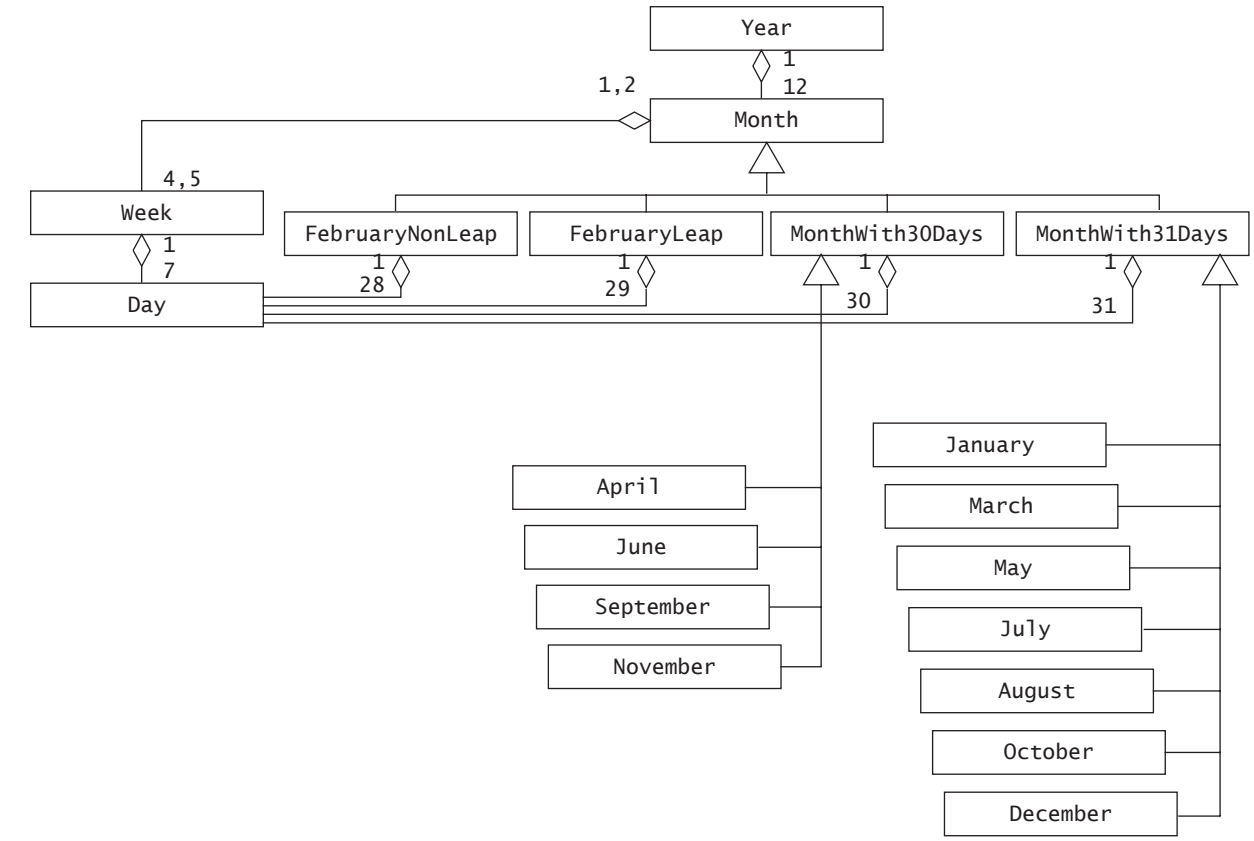


Figure 5-5 Revised class diagram indicating how many days each month includes.

5–8 Consider a traffic light system at a four-way crossroads (e.g., two roads intersecting at right angles). Assume the simplest algorithm for cycling through the lights (e.g., all traffic on one road is allowed to go through the crossroad while the other traffic is stopped). Identify the states of this system and draw a statechart describing them. Remember that each individual traffic light has three states (i.e. green, yellow, and red).

We model this system as two groups of lights, one for each road. Opposing lights have always the same value. Lights from at least one group must always be red for safety reasons. We assume there are no separate lights for left and right turns and that cycles are fixed. Figure 5-6 depicts the startchart diagram for this solution.

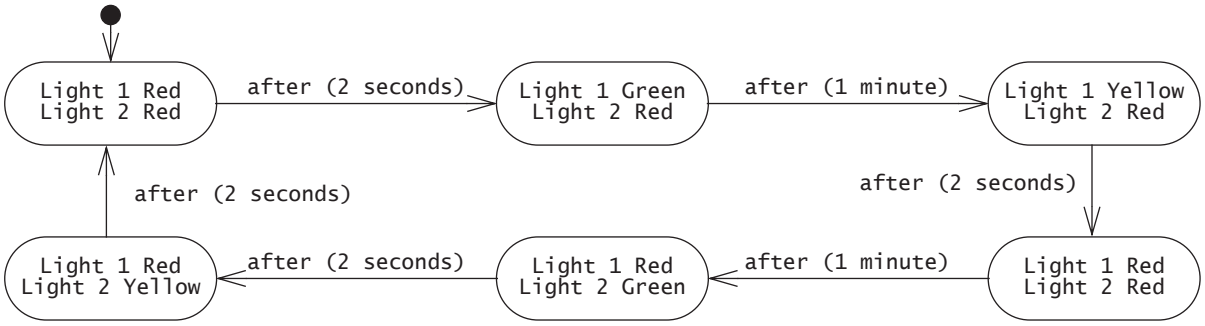


Figure 5-6 Statechart diagram for simplistic traffic light system.

5–9 From the sequence diagram Figure 2-34, draw the corresponding class diagram. Hint: Start with the participating objects in the sequence diagram.

Figure 5-7 depicts a sample solution. The student should figure out from the sequence diagram that there should be access paths among all three classes. One way to ensure that these paths exist is to introduce a top-level object `2Bwatch` which has aggregation associations with the other three classes.

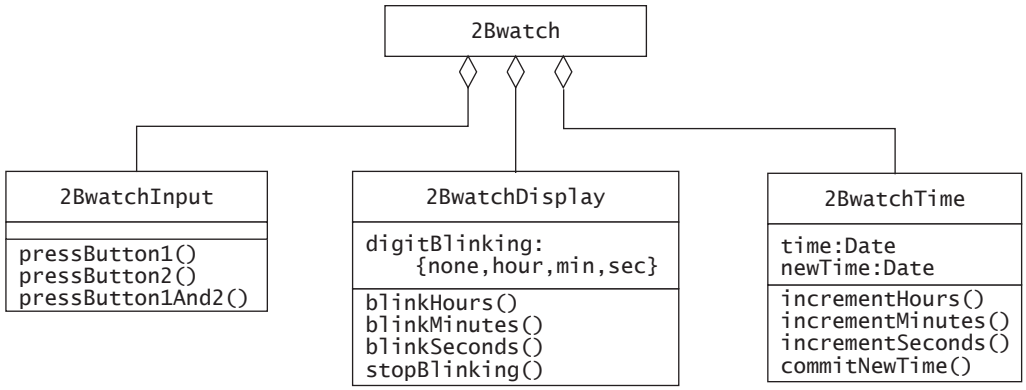


Figure 5-7 Sample solution for Exercise 5–5

5–10 Consider the addition of a nonfunctional requirement stipulating that the effort needed by `Advertisers` to obtain exclusive sponsorships should be minimized. Change the `AnnounceTournament` and the `ManageAdvertisement` use cases (solution of Exercise 4-12) so that the `Advertiser` can specify preferences in her profile so that exclusive sponsorships can be decided automatically by the system.

A new use case, ManageSponsorships, should be written and included in the ManageAdvertisement use case and define the preferences that the Advertiser can specify for automatically accepting an exclusive sponsorship. Figure 5-8 depicts a possible solution for this use case.

Use case name	ManageSponsorships
Participating actors	Initiated by Advertiser
Flow of events	1. The Advertiser requests the sponsorships area..
	2. ARENA displays the list of tournaments for which the Advertiser has exclusive sponsorships and indicates for each tournament whether the exclusive sponsorship was decided by the system or the Advertiser.
	3. The Advertiser may change the conditions under which a tournament becomes a candidate for her exclusive sponsorship, including specifying a set of leagues, a time frame, or a maximum fee that she is willing to pay for an exclusive sponsorship. The Advertiser may also select not to use the automatic sponsorship feature and be notified about such opportunities directly.
	4. ARENA records the changes of preferences for the automatic sponsorships.
Entry condition	• The Advertiser is logged into ARENA.
Exit conditions	• ARENA checks the advertisers automatic sponsorship profile for any new tournaments created after the completion of this use case.

Figure 5-8 ManageSponsorships use case.

In the AnnounceTournament use case, steps 6 & 7 are the modified as follows:

6. The system notifies the selected sponsors about the upcoming tournament and the flat fee for exclusive sponsorships. For advertisers who selected an automatic sponsorship preferences, the system automatically generates a positive answer if the fee falls within the bounds specified by the Advertiser.
7. The system communicates their answers to the LeagueOwner.

5–11 Identify and write definitions for any additional entity, boundary, and control objects participating in the AnnounceTournament use case that were introduced by realizing the change specified in Exercise 5-10.

As specified above, this exercise should result in the identification of two new entity objects:

- SponsorshipPreferences stores the time frame during which a tournament can be considered for an automatic sponsor ship and the maximum fee the Advertiser is willing to pay without being notified. This object also has associations to the leagues that should be considered during the automatic sponsorship decision.
- ExclusiveSponsorship represents an exclusive sponsorship that was awarded to an Advertiser. It is associated with the tournament and Advertiser objects involved in the sponsorship. It stores a boolean indicating whether or not the sponsorship was decided automatically.

However, the instructor may choose to extend the exercise so that the students also consider the objects participating in the ManageSponsorships use case. In this case, the student should identify a control object for the new use case and two boundary objects for displaying the exclusive sponsorships and the preferences, respectively.

5–12 Update the class diagrams of Figure 5-29 and Figure 5-31 to include the new objects you identified in Exercise 5-11.

Figure 5-9 depicts the objects discussed in Exercise 5-11 and their associations.

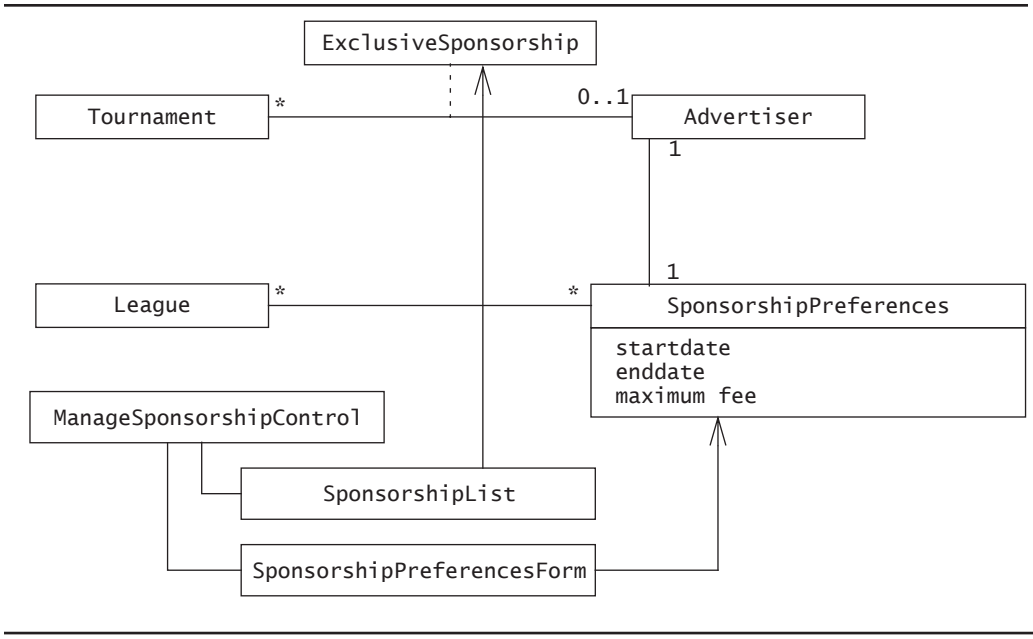


Figure 5-9 objects added after revising the `AnnounceTournament` and `ManageSponsorships` use case.

5–13 Draw a statechart describing the behavior of the `AnnounceTournamentControl` object based on the sequence diagrams of Figures 5-26 through 5-28. Treat the sending and receiving of each notice as an event that triggers a state change.

The `AnnounceTournament` is a simple linear workflow. Hence, this exercise is used to test the students knowledge of statechart diagram syntax and for choosing appropriate names for states and transitions. For modeling exercises, the instructor is advised to select more complex workflows that include decision points and concurrency, similar to the change process described in Figure 5-22.

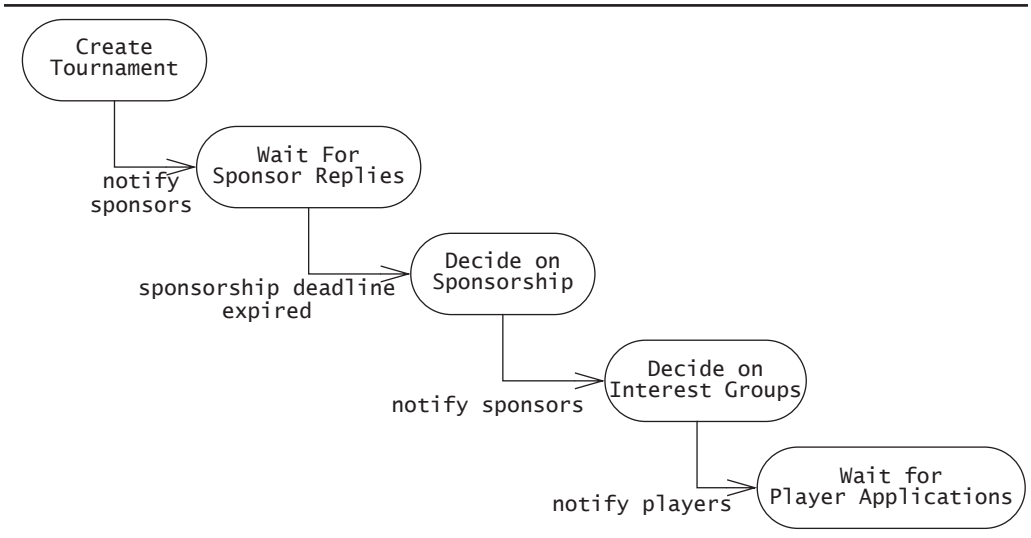


Figure 5-1 A UML startchart diagram for `AnnounceTournamentControl`.

6. System Design: Decomposing the System

6-1 Decomposing a system into subsystems reduces the complexity developers have to deal with by simplifying the parts and increasing their coherence. Decomposing a system into simpler parts usually results into increasing a different kind of complexity: Simpler parts also means a larger number of parts and interfaces. If coherence is the guiding principle driving developers to decompose a system into small parts, which competing principle drives them to keep the total number of parts small?

Decreasing coupling is the principle that competes with increasing coherence. A large number of parts will result in a large number of interfaces and many dependencies among the parts.

6–2 In Section 6.4.2 in the book, we classified design goals into five categories: performance, dependability, cost, maintenance, and end user. Assign one or more categories to each of the following goals:

- *Users must be given a feedback within 1 second after they issue any command.* [Performance]
- *The* `TicketDistributor` *must be able to issue train tickets, even in the event of a network failure.* [Dependability]
- *The housing of the* `TicketDistributor` *must allow for new buttons to be installed in the event the number of different fares increases.* [Maintenance]
- *The* `AutomatedTellerMachine` *must withstand dictionary attacks (i.e., users attempting to discover a identification number by systematic trial).* [Dependability]
- *The user interface of the system should prevent users from issuing commands in the wrong order.* [End user]

6–3 You are developing a system that stores its data on a Unix file system. You anticipate that you will port future versions of the system to other operating systems that provide different file systems. Propose a subsystem decomposition that anticipates this change.

A Bridge pattern should be used to define a generic interface to the storage subsystem. Consequently, different implementations of the storage subsystem can be provided to dealing with specific file systems. Note that the Bridge pattern does not address the problem of moving files from one file system to another. To address this problem, a separate conversion utility needs to be developed.

6–4 Older compilers were designed according to a pipe and filter architecture, in which each stage would transform its input into an intermediate representation passed to the next stage. Modern development environments, including compilers integrated into interactive development environments with syntactical text editors and source-level debuggers, use a repository architecture. Identify the design goals that may have triggered the shift from pipe and filter to repository architecture.

Older compilers trace their lineage to batch systems, in which a stack of cards were fed to the computer by system administrators and developers would pick up the resulting printout later in the day. The design goal of such system was high throughput, so that as many jobs as possible could be executed. A pipe and filter architecture allows parallelism between subsequent stages in the pipeline and achieves this design goal.

Modern interactive development environments are interactive, compilation occurs incrementally as the user types. The design goal of such environments is a short response time.

6–5 Consider the model/view/control example depicted in Figures 6-16 and 6-15.
a. Redraw the collaboration diagram of Figure 6-16 as a sequence diagram.

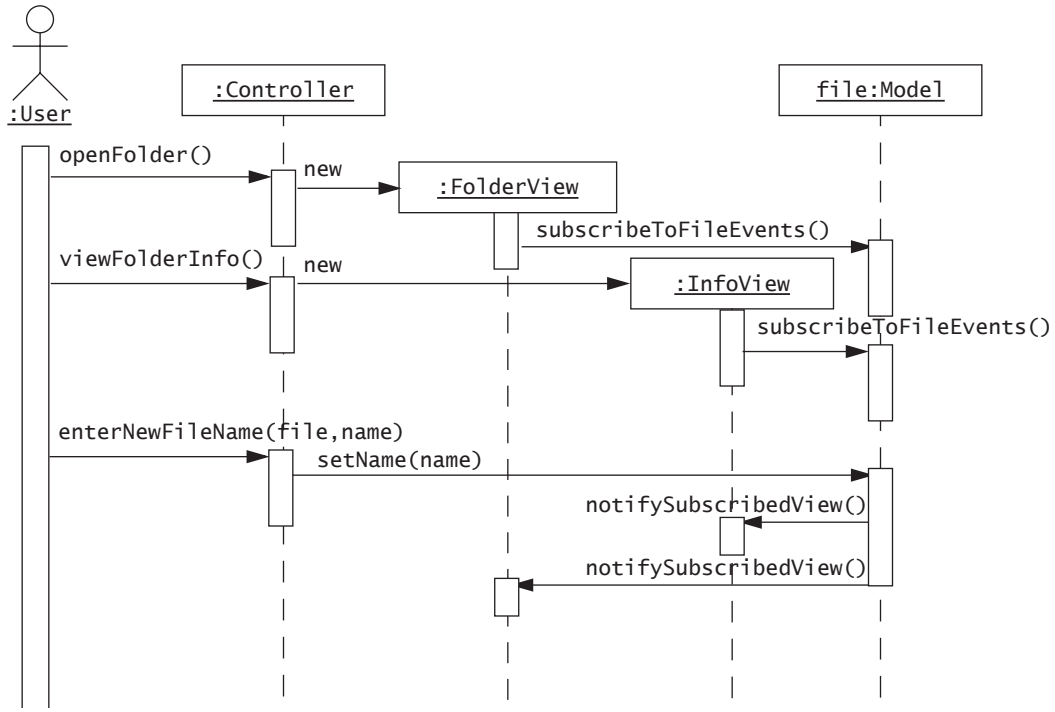


Figure 6-1 Sample solution for Exercise 6-5.

b. Discuss how the MVC architecture helps or hurts the following design goals.

- *Extensibility (e.g., the addition of new types of views).*

The MVC helps extensibility in terms of new types of views, since neither model objects nor existing view objects need to be modified to accommodate the new view.

- *Response time (e.g., the time between a user input and the time all views have been updated)*

A pure MVC hurts response time, because the view the user sees is updated only after the model has been updated. This can be an issue for commands involving a rapid series of interactions (e.g., dragging or resizing a geometrical shape).

- *Modifiability (e.g., the addition of new attributes in the model)*

The MVC helps this goal, since only the views that need to be aware of the new attribute need to be modified. All other views can be left unchanged.

- *Access control (i.e., the ability to ensure that only legitimate users can access specific parts of the model).*

The MVC helps this goal, as the model is accessed using a clear interface (the public methods of the model objects) which can be controlled, for example, using a proxy pattern for each model class.

6–6 List design goals that would be difficult to meet when using a closed architecture with many layers, such as the OSI example depicted in Figure 6-10.

A closed architecture requires method invocations across each layer boundary and often also copying data between each layer. This hurts both response time and memory foot print.

6–7 In many architectures, such as the three- and four-tier architectures (Figures 6-21 and 6-22), the storage of persistent objects is handled by a dedicated layer. In your opinion, which design goals have lead to this decision?

Portability: Separating the application from the details of storage allows the storage layer to be replaced by a different one (e.g., switching database management systems).

Robustness: It is easier to guarantee the integrity of the stored data if it is handled by a single layer.

7. System Design: Addressing Design Goals: Solutions

7-1 Consider a system that includes a Web server and two database servers. Both database servers are identical: The first acts as a main server, while the second acts as a redundant backup in case the first one fails. Users use Web browsers to access data through the Web server. They also have the option of using a proprietary client that accesses the databases directly. Draw a UML deployment diagram representing the hardware/software mapping of this system.

This exercise tests the student's knowledge of deployment diagrams. Several different solutions exist depending on the interpretation of the word "server." Given that the system uses a hot backup redundancy to increase availability, the student can infer that high availability is a design goal for the system. In this case, the system designers would also assign each server to a dedicated host, first, to increase the independence of failures, second, to limit damage in case of failure. Figure 7-1 depicts such a solution with three servers and two clients. An instructor may want to accept a solution where both database servers and the Web server run on the same host, given that this exercise tests knowledge about deployment diagrams, not about dependable systems.

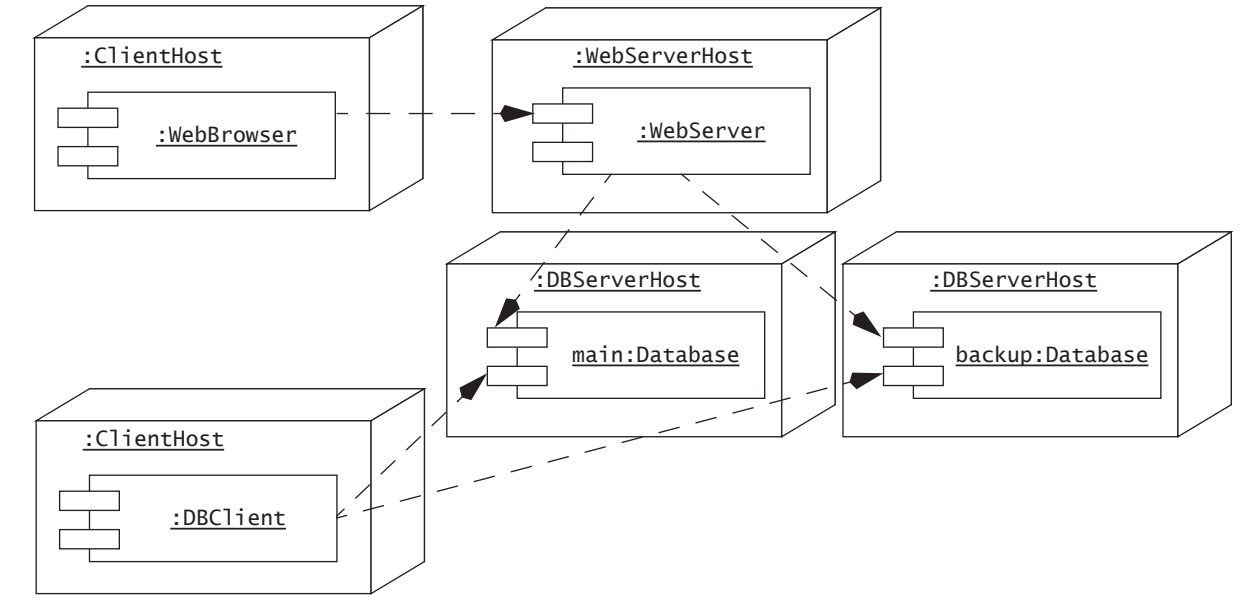


Figure 7-1 Sample solution for Exercise 7-1.

7–2 Consider a legacy, fax-based, problem-reporting system for an aircraft manufacturer. You are part of a reengineering project replacing the core of the system by a computer-based system, which includes a database and a notification system. The client requires the fax to remain an entry point for problem reports. You propose an E-mail entry point. Describe a subsystem decomposition, and possibly a design pattern, which would allow both interfaces.

There are two main approaches to this problem. The first approach is to design the system with an E-mail frontend and add a subsystem which converts fax problem reports to E-mail (possibly with the assistance of an actor). The E-mail front end then becomes a single point of entry into the problem reporting system. The second approach (depicted in Figure 7-2) is to provide a network level interface to the problem reporting system (e.g., using RPC, RMI, CORBA, or plain sockets) and to build two different clients to the system, one processing E-mail, the other processing fax reports. The first solution is cheaper to realize and would be appropriate if the fax front end is short lived or no other frontends are planned. The second solution is more general and allows the addition of other frontends in the future. Design goals defined during system design would be used to decide between these two

alternatives. The main point the student should realize is that there are alternative solutions to subsystem decomposition issues and that design goals can be used to sort them out.

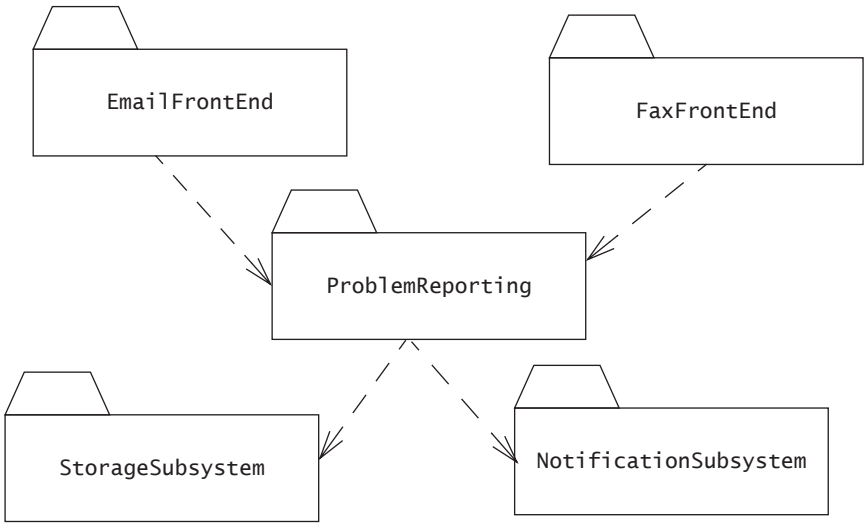


Figure 7-2 Sample solution for Exercise 7-2.

7–3 You are designing the access control policies for a Web-based retail store. Customers access the store via the Web, browse product information, input their address and payment information, and purchase products. Suppliers can add new products, update product information, and receive orders. The store owner sets the retail prices, makes tailored offers to customers based on their purchasing profiles, and provides marketing services. You have to deal with three actors: `StoreAdministrator`, `Supplier`, *and* `Customer`. *Design an access control policy for all three actors.* `Customers` *can be created via the Web, whereas* `Suppliers` *are created by the* `StoreAdministrator`.

An access control policy is represented with a matrix. The columns represent objects whose access is controlled, the rows represent the actors accessing the objects, the cells contain the operations that an actor is allowed to invoke for a specific object. In this exercise, there are four objects under access control: the product (including product information and price), the customer information, the supplier information, and the order. There are four actors which should be taken into account: the three actors mentioned in the exercise and the unregistered web user which can browse the product catalog and the create a new customer (as indicated in the last sentence of the exercise. Table 7-1 depicts a possible access matrix for the text above. The name of the operations may be different from one solution to another. The instructor may consider correct solutions which merge the unregistered user row with the customer row.

Table 7-1 Access control matrix for the web store of exercise 6.5.

	Product	**CustomerInfo**	**SupplierInfo**	**Order**
Unregistered web user	`getInfo()`	`create()`		
Customer	`getInfo()` `getPrice()`	`updateInfo()`		`create()`

Supplier	`create()`
	`getInfo()`
	`updateInfo()`
			`updateInfo()`	`process()`

StoreAdministrator	`updatePrice()`	`verifyInfo()` `examineProfile()`	`create()`	`examine()`

7–4 Select a control flow mechanism you find most appropriate for each of the following systems. Because multiple choices are possible in most cases, justify your choices.

- *a Web server designed to sustain high loads* [Threaded control flow]
- *a graphical user interface for a word processor* [Event-based control flow]
- *a real-time embedded system (e.g., a guidance system on a satellite launcher)* [Event-based control flow (e.g., preemptive, interrupt driven, scheduler)]

7–5 Why are use cases that describe boundary conditions described during system design (as opposed to during requirements elicitation or analysis)?

Use cases that describe boundary conditions depend on system design decisions. For example, software architecture decisions need to be made before developers can describe how the system is started or shutdown.

7–6 You are designing a caching subsystem that temporarily stores data retrieved over the network (e.g., web pages) into a faster access storage (e.g., the hard disk). Due to a change in requirements, you define an additional service in your subsystem for configuring cache parameters (e.g., the maximum amount of hard disk the cache can use). Which project participants do you notify?

You should notify at least:

- all teams using your subsystem
- the team responsible for testing your subsystem
- the team responsible for documenting the interface of your subsystem
- the architecture team

8. Object Design: Reusing Pattern Solutions

8-1 Consider the ARENA *object design model. For each of the following objects, indicate if it is an application object or a solution object:*

- `League` [application object]
- `LeagueStore` [solution object]
- `LeagueXMLStoreImplementor` [solution object]
- `Match` [application object]
- `MatchView` [solution object]
- `Move` [application object]
- `ChessMove`. [application object]

8–2 Indicate which occurrences of the following inheritance relationships are specification inheritance and which are implementation inheritance:

- *A* `Rectangle` *class inherits from a* `Polygon` *class*. [specification inheritance]
- *A* `Set` *class inherits from a* `BinaryTree` *class*. [implementation inheritance]
- *A* `Set` *class inherits from a* `Bag` *class (a* `Bag` *is defined as an unordered collection)*. [specification inheritance]
- *A* `Player` *class inherits from a* `User` *class*. [specification inheritance]
- *A* `Window` *class inherits from a* `Polygon` *class*. [implementation inheritance]

8–3 Consider an existing game of bridge written in Java. We are interested in integrating this bridge game into ARENA. *Which design pattern would you use? Draw a UML class diagram relating the* ARENA *objects with some of the classes you would expect to find in the bridge game.*

Integrating an existing game requires writing one `Adapter` per ARENA class in the `Game` interface. Figure 8-1 depicts an example for the `GameMove` class.

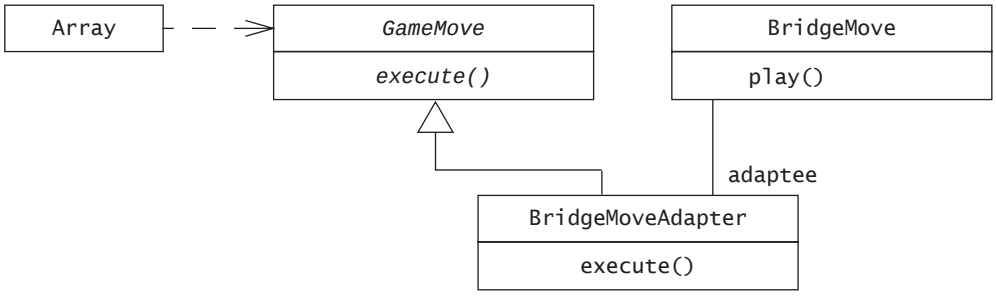


Figure 8-1 Sample solution for 8–3.

8–4 Consider a workflow system supporting software developers. The system enables managers to model the process the developers should follow in terms of processes and work products. The manager can assign specific processes to each developer and set deadlines for the delivery of each work product. The system supports several types of work products, including formatted text, picture, and URLs. The manager, while editing the workflow, can dynamically set the type of each work product at run time. Assuming one of your design goals is to design the system so that more work product types can be added in the future, which design pattern would you use to represent work products?

We use a bridge pattern. The interface class stores the deadline and type of work product and provides the interface to the rest of the application. The implementor classes store the content of the work product and can be substituted at run time. All implementor classes comply to the same abstract `WorkProductContent` interface.

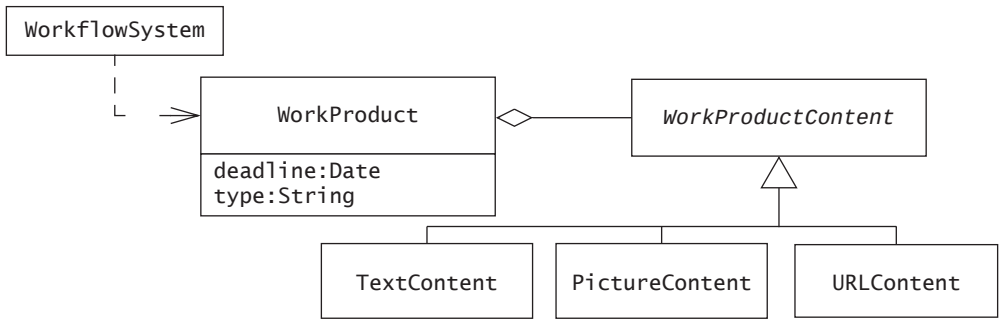


Figure 8-2 Sample solution for Exercise 8–4.

8–5 Consider a system that includes a database client and two redundant database servers. Both database servers are identical: the first acts as a main server, the second acts as a hot back-up in case the main server fails. The database client accesses the severs through a single component called a "gateway," hence hiding from the client which server is currently being used. A separate policy object called a "watchdog" monitors the requests and responses of the main server and, depending on the responses, tells the gateway whether to switch over to the back-up server. What do you call this design pattern? Draw a UML class diagram to justify your choice.

This can be interpreted as a Strategy in which two concrete strategies are identical, that is, they are instances of the same class (Figure 8-3). It can also be argued that this is a Proxy pattern which interacts with an external policy object.

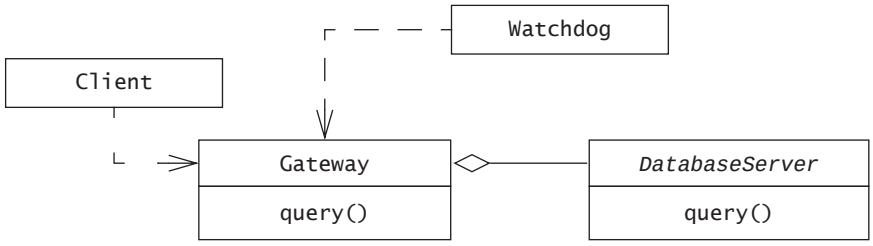


Figure 8-3 Sample solution for Exercise 8–5.

8–6 In Section 8.4.1, we used a Bridge pattern to decouple the implementation of the `ARENA` `LeagueStore` *subsystem from its interface, enabling us to provide different implementations for the purpose of testing. Ideally, we would apply the Bridge pattern to each subsystem in our system design to facilitate testing. Unfortunately, this is not always possible. Give an example of a subsystem where the Bridge pattern cannot be used.*

A Bridge pattern can be used in the case of a subsystem with a single Facade. Subsystems that offer a more complex interface, for example a whitebox framework used for realizing a user interface, are difficult to encapsulate.

8–7 Consider the following design goals. For each of them, indicate the candidate pattern(s) you would consider to satisfy each goal:

- *Given a legacy banking application, encapsulate the existing business logic component.* [Adapter]
- *Given a chess program, enable future developers to substitute the planning algorithm that decides on the next move with a better one.* [Bridge]
- *Given a chess program, enable a monitoring component to switch planning algorithms at runtime, based on the opposing player's style and response time.* [Strategy]
- *Given a simulation of a mouse solving a maze, enable the path evaluation component to evaluate different paths independently of the types of moves considered by the mouse.* [Command]

8–8 Consider an application that must select dynamically an encryption algorithm based on security requirements and computing time constraints. Which design pattern would you select? Draw a UML class diagram depicting the classes in the pattern and justify your choice.

A strategy pattern can be used to address this problem. The EncryptionAlgorithm interface defines the common interface that all encryption algorithms must comply with. A Policy class selects a concrete algorithm based on the computing time constraints and the security requirements of the application. The Client interacts only with the Context interface which stores the reference to the concrete algorithm.

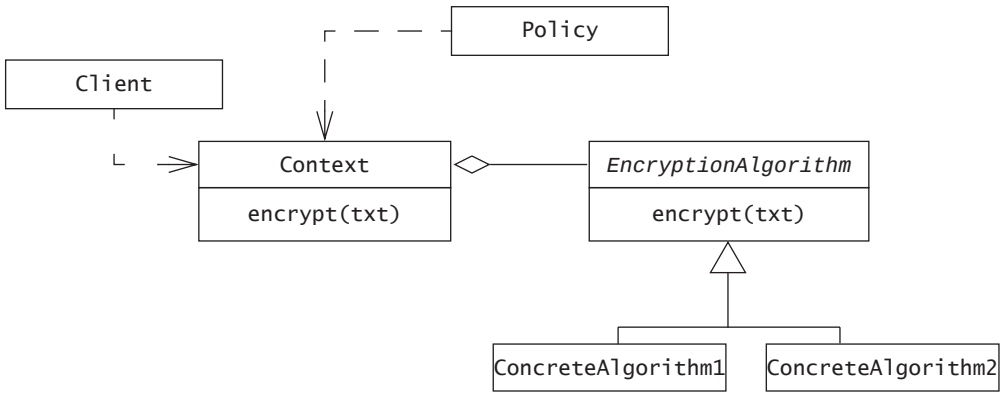


Figure 8-4 Sample solution for Exercise 8–8.

9. Object Design: Specifying Interfaces

9-1 Consider the `List` *interface in the* `java.util` *package for ordered collections of objects. Write preconditions and post conditions in OCL for the following operations:*

The constraints below assume that an ordered association between the list and its contained elements is called `elements`.

- `int size()` *returns the number of elements in the list.*

```
context List::size() post: result = elements->size
```

- `void add(Object e)` *adds an object at the end of the list.*

```
context List::add(e) post: contains(e) and size() = @pre.size() + 1
```

- `void remove(Object e)` *removes an object from the end of the list.*

```
context List::remove(e) post:
    (@pre.contains(e) implies size() = @pre.size() - 1) and
    (not @pre.contains(e) implies size() = @pre.size()) and
    result = @pre.contains(e)
```

- `boolean contains(Object e)` *returns true if the object is contained in the list.*

```
context List::contains(e) post: result = elements->includes(e)
```

- `Object get(int idx)` *returns the object located at index* `idx`*, 0 being the index of the first object in the list.*

```
context List::get(idx) pre: idx >= 0 and idx < size()
context List::get(idx) post: result = elements->at(idx+1)
```

9–2 Consider the `Set` *interface in the* `java.util` *package. Write preconditions and post conditions in OCL for the following operations:*

The constraints below assume that the association between the list and its contained elements is called `elements`.

- `int size()` *returns the number of elements in the set.*

```
context Set::size() post: result = elements->size()
```

- `void add(Object e)` *adds an object to the set. If the object is already in the set, does nothing.*

```
context Set::add(e) post:
    contains(e) and
    (@pre.contains(e) implies size() = @pre.size()) and
    (not @pre.contains(e) implies size() = @pre.size()+1)
```

- `void remove(Object e)` *removes an object from the set*

```
context Set::remove(e) post:
    not contains(e) and
    (@pre.contains(e) implies size() = @pre.size() - 1) and
    (not @pre.contains(e) implies size() = @pre.size())
```

- `boolean contains(Object e)` *returns true if the object is contained in the set.*

```
context Set::contains(e) post: elements->includes(e)
```

9–3 Consider the `Collection` *interface in the* `java.util` *package, which is the ancestor of both* `List` *and* `Set`*. Write preconditions and postconditions for the operations below and modify the constraints you wrote in exercises 9-1 and 9–2, knowing that contracts are inherited. Make sure you comply with the Liskov Substitution Principle.*

- `int size()` *returns the number of elements in the collection.*

```
context Collection::size() post: result = elements->size()
```

- `void add(Object e)` *adds an object to the collection.*


```
context Collection::add(e) post: contains(e)
```

- `void remove(Object e)` *removes an object from the collection.*

```
context Collection::remove(e) post:
    (@pre.contains(e) implies size() = @pre.size() - 1) and
    (not @pre.contains(e) implies size() = @pre.size())
```

- `boolean contains(Object e)` *returns true if the object is in the collection.*

```
context Collection::contains(e) post: elements->includes(e)
```

With contract inheritance, no constraints on `remove`, `contains`, and `size` of `List` and `Set` need to be added. The post condition on `add()` on `Set` and `List` become:

```
context List::add(e) post: size() = @pre.size() + 1
context Set::add(e) post:
    (@pre.contains(e) implies size() = @pre.size()) and
    (not @pre.contains(e) implies size() = @pre.size() + 1)
```

9–4 Consider a `Rectangle` *class and a* `Square` *class that inherits from the* `Rectangle` *class:*

- *Write post conditions for the* `Rectangle.setWidth(w:int)` *and the* `Rectangle.setHeight(h:int)` *operations in terms of the* `Rectangle.getWidth():int` *and the* `Rectangle.getHeight():int` *operations.*

```
context Rectangle::setWidth(w) post:
    w = getWidth() and getHeight() = @pre.getHeight()
context Rectangle::setHeight(h) post:
    h = getHeight() and getWidth() = @pre.getWidth()
```

- *Write an invariant for the* `Square` *class stating that the width and height of a* `Square` *should always be the same.*

```
context Square inv: getWidth() = getHeight()
```

- *Consider the rules for inheriting contracts described in Section 9.4.5 in the context of the* `Square.setWidth()` *and* `Square.setHeight()` *operations of the* `Square` *class. Are all rules met? Why not? What should change in the model?*

As written above, the `Square.setWidth()` and `Square.setHeight()` would violate the postconditions of the `Rectangle.setWidth()` and the `Rectangle.setHeight()`. `Square` should not inherit from `Rectangle` as it does not provide the same external behavior with respect to `setWidth()` and `setHeight()` (as defined by the LSP).

To make this exercise clearer, the instructor could write the postcondition for `Rectangle.getWidth()`, as the second part of the postcondition is often missed.

9–5 Consider a sorted list. Write an invariant in OCL denoting that the elements of the list are sorted.

The key to this exercise is to realize that the List class in Java provides an `indexOf(element)` method.

```
context SortedList inv:
    elements->forAll(e1, e2:Object|
        e1.lessThan(e2) implies indexOf(e1) < indexOf(e2))
```

9–6 Consider a sorted binary tree data structure for storing integers. Write invariants in OCL denoting that

- *All nodes in the left subtree of any node contain integers that are less than or equal to the current node, or the subtree is empty.*

```
context BinaryTree inv:
    left.getSubElements()->size = 0 or
        left.getSubElements()->forAll(e:Integer| e <= getElement())
```

- *All nodes in the right subtree of any node contain integers that are greater than the current tree, or the subtree is empty.*


```
context BinaryTree inv:
    right.getSubElements()->size = 0 or
        right.getSubElements()->forAll(e:Integer|e > getElement())
```

- *The tree is balanced.*

```
context BinaryTree inv:
    left.getSubElements()->size <= right.getSubElements()->size + 1 and
    right.getSubElements()->size <= left.getSubElements()->size + 1
```

9–7 Consider a simple intersection with two crossing roads and four traffic lights. Assume a simple algorithm for switching lights, so that the traffic on one road can proceed while the traffic on the other road is stopped. Model each traffic light as an instance of a `TrafficLight` class with a `state` attribute that can be either `red`, `yellow`, or `green`. Write invariants in OCL on the `state` attribute of the `TrafficLight` class that guarantee that the traffic cannot proceed on both roads simultaneously. Add associations to the model to navigate the system, if necessary. Note that OCL constraints are written on classes (as opposed to instances).

We add an association called `right` to denote the traffic light on the road to the right. Hence, the expression `right.right` denotes the opposing light.

```
/* Opposing lights have always the same state */
context TrafficLight inv:
    state = right.right.state
/* If the current light is green or yellow, the light on the right must be red */
context TrafficLight inv:
       (state = #green or state = #yellow) implies right.state = #red
```

9–8 After reading Sections 9.6.2 and 9.6.3, write constraints for a `RoundRobinStyle` class and a `RoundRobinRound` class, implementing the `TournamentStyle` and the `Round` interfaces, respectively. Assume that `RoundRobinStyle` plans a series of `Rounds` so that each `Player` is paired with the other `Players` exactly once in the `Tournament`. Note that the number of `Rounds` depends on whether the number of `Players` in the `Tournament` is odd or even, and that a given `Player` cannot play more than once in a given `Round`.

```
/* If the number of players is even, the number of rounds is one less than the
 * number of players. Otherwise, it is equal to the number of players */
context RoundRobinStyle::planRounds(t:Tournament) post:
    if (t.players.size.mod(2) = 0) then
        result->size = t.players.size - 1
    else
        result->size = t.players.size
    endif
/* Each player plays each other in a RoundRobinStyle tournament.
context RoundRobinStyle::planRounds(t:Tournament) post:
       t.players->forAll(p:Player|
           p.getMatches(t).players->includesAll(t.players))
```


10. Mapping Models to Code: Solutions

10-1 In Web pages, tables consist of rows, which in turn consist of cells. The actual width and height of each cell is computed based in part on its content (e.g., the amount of text in the cell, the size of an image in the cell), and the height of a row is the maximum of the heights of all cells in the row. Consequently, the final layout of a table in a Web page can only be computed once the content of each cell has been retrieved from the Internet. Using the proxy pattern described in Figure 10-7, describe an object model and an algorithm that would enable a Web browser to start displaying a table before the size of all cells is known, possibly redrawing the table as the content of each cell is downloaded.

The proxy design pattern is applied to the table cell. When created, `TableCellProxy` estimates its dimensions as best as it can and updates them as the content is retrieved. Every time `TableCellProxy` provides a new estimate of its dimensions, it signals `Table` (e.g., via a semaphore or a listener). The `Table` then recomputes the dimensions and positions of all of its rows and redraws itself.

In general, the challenge of this type of approach is not the algorithm but to provide a visual behavior that does not confuse the user, especially when only part of the table is visible.

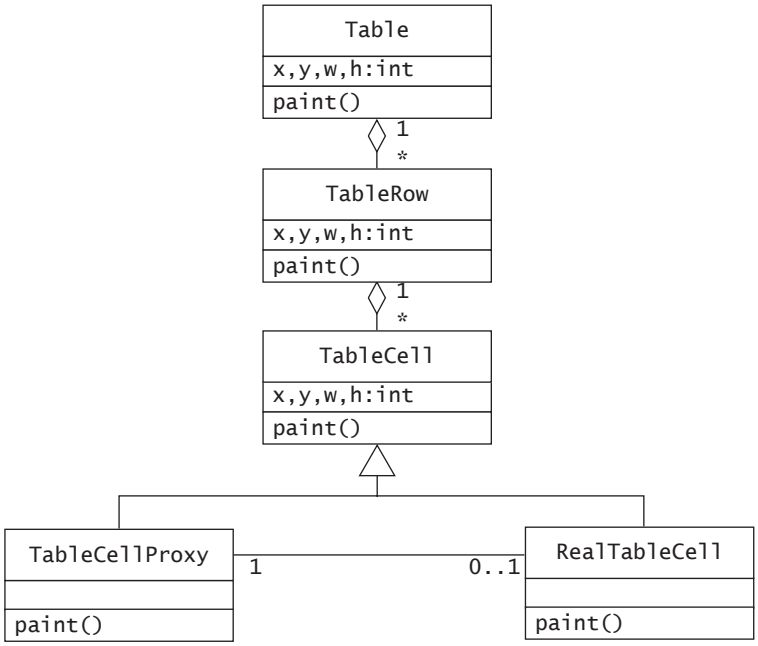


Figure 10-1 Sample class diagram for Exercise 10-1.

10–2 Apply the appropriate transformations described in Section 10.4.2 to the associations below. Assume that all associations are bidirectional and that they can change during the lifetime of each object. Write the source code needed to manage the associations, including class, field, and method declarations, method bodies, and visibility.

Figures 10-2 through 10-5 depict sample classes for this exercise. Import statements and constructors were omitted. Since no ordering was specified on the associations, a `Set` was used to represent the many side. If the student selects a `List`, additional checking code must be added to ensure that multiple associations to the same object are not created.


```
public class Mailbox {
    private Set folders = new HashSet();
    public Folder [] getFolders() {
        return (Folder[])folders.toArray();
    }
    public void addFolder(Folder f) {
        folders.add(f);
        f.setMailbox(this);
    }
    public void removeFolder(Folder f) {
        folders.remove(f);
        f.setMailbox(null);
    }
}
```

Figure 10-2 Class `Mailbox`.

```
public class Folder {
    private Mailbox mailbox = null;
    private Set messages = new HashSet();
    public Mailbox getMailbox() {
        return mailbox;
    }
    public void setMailbox(Mailbox newMailbox) {
        if (newMailbox != mailbox) {
            Mailbox oldMailbox = mailbox;
            mailbox = newMailbox;
            if (newMailbox != null) {
                newMailbox.addFolder(this);
            }
            if (oldMailbox != null) {
                oldMailbox.removeFolder(this);
            }
        }
    }
    public Message [] getMessages() {
        return (Message[])messages.toArray();
    }
    public void addMessage(Message m) {
        messages.add(m);
        m.setFolder(this);
    }
    public void removeMessage(Message m) {
        messages.remove(m);
        m.setFolder(null);
    }
}
```

Figure 10-3 Class `Folder`


```
public class Message {
    private Folder folder = null;
    private Set views = new HashSet();
    public Folder getFolder() {
        return folder;
    }
    public void setFolder(Folder newFolder) {
        if (newFolder != folder) {
            Folder oldFolder = folder;
            folder = newFolder;
            if (newFolder != null) {
                newFolder.addMessage(this);
            }
            if (oldFolder != null) {
                oldFolder.removeMessage(this);
            }
        }
    }
    public View [] getViews() {
        return (View[])views.toArray();
    }
    public void addView(View v) {
        if (!views.contains(v)) {
            views.add(v);
            v.addMessage(this);
        }
    }
    public void removeView(View v) {
        if (views.contains(v)) {
            views.remove(v);
            v.removeMessage(this);
        }
    }
}
```

Figure 10-4 Class `Message`.

```
public class View {
    private Set messages = new HashSet();
    public Message [] getMessages() {
        return messages.toArray();
    }
    public void addMessage(Message m) {
        if (!messages.contains(m)) {
            messages.add(m);
            m.addView(this);
        }
    }
    public void removeMessage(Message m) {
        if (messages.contains(m)) {
            messages.remove(m);
            m.removeView(this);
        }
    }
}
```

Figure 10-5 Class `View`.

10–3 Apply the appropriate transformations described in Section 10.4.2 to the associations below. Assume that all associations are bidirectional, but that the aggregation associations do not change after each object has been created. In other words, the creator of each class must be modified so that aggregations are initialized during the creation of each object. Write the source code needed to manage the associations, including class, field, and method declarations, method bodies, and visibility.

Figures 10-6 through 10-9 depict a sample solution. The constructor of `Tournament` and `Round` set the one side of the aggregation association, removing the need for a `setLeague` and `setTournament` method.

```
public class League {
    private Set tournaments;
    public League() {
        tournaments = new HashSet();
    }
    public Tournament [] getTournaments() {
        return (Tournament[])tournaments.toArray();
    }
    public void addTournament(Tournament t) {
        tournaments.add(t);
    }
    public void removeTournament(Tournament t) {
        tournaments.remove(t);
    }
}
```

Figure 10-6 Class `League`.

```
public class Tournament {
    private League league;
    private Set rounds;
    private Set players;

    public Tournament(League l) {
        rounds = new HashSet();
        players = new HashSet();
        league = l;
        l.addTournament(this);
    }

    public League getLeague() {
        return league;
    }

    public Round [] getRounds() {
        return (Round[])rounds.toArray();
    }

    public void addRound(Round r) {
        rounds.add(r);
    }

    public void removeRound(Round r) {
        rounds.remove(r);
    }

    public Player [] getPlayers() {
        return (Player[])players.toArray();
    }

    public void addPlayer(Player p) {
        if (!players.contains(p)) {
            players.add(p);
            p.addTournament(this);
        }
    }

    public void removePlayer(Player p) {
        if (players.contains(p)) {
            players.remove(p);
            p.removeTournament(this);
        }
    }
}
```

Figure 10-7 Sample class `Tournament`.

```
public class Round {
    private Tournament tournament;

    public Round(Tournament t) {
        tournament = t;
        t.addRound(this);
    }

    public Tournament getTournament() {
        return tournament;
    }
}
```

Figure 10-8 Class `Round`.


```
public class Player {
    private Set tournaments;

    public Player() {
        tournaments = new HashSet();
    }

    public Tournament [] getTournaments() {
        return (Tournament[])tournaments.toArray();
    }

    public void addTournament(Tournament t) {
        if (!tournaments.contains(t)) {
            tournaments.add(t);
            t.addPlayer(this);
        }
    }

    public void removeTournament(Tournament t) {
        if (tournaments.contains(t)) {
            tournaments.remove(t);
            t.removePlayer(this);
        }
    }
}
```

Figure 10-9 Class `Player`.

10–4 Figure 10-15 depicts the checking code for the `addPlayer()` *method of* `Tournament`. *Write the checking code for the other constraints associated with* `Tournament` *depicted in Figure 9-16.*

Figure 10-10 depicts the checking code for `Tournament.removePlayer()`. The key points are that

- preconditions are checked first,
- some values may have to be saved for checking postconditions before the real work is done,
- postconditions and invariants are checked after the real work is done, and
- the conditions checked (i.e., the conditions under which an exception should be raised) are the opposite of the conditions stated in the contracts (i.e., the conditions that should always be true).


```
public class Tournament {
//...
    private List players;

    public void removePlayer(Player p) throws UnknownPlayer, KnownPlayer,
            IllegalNumPlayers, IllegalLeague, IllegalName
    {
        // check precondition isPlayerAccepted(p)
        if (!isPlayerAccepted(p)) {
            throw new UnknownPlayer(p);
        }
        // check precondition getNumPlayers() < maxNumPlayers
        if (getNumPlayers() == getMaxNumPlayers()) {
            throw new TooManyPlayers(getNumPlayers());
        }
        // save values for postconditions
        int pre_getNumPlayers = getNumPlayers();
        // accomplish the real work
        players.add(p);
        p.addTournament(this);
        // check post condition !isPlayerAccepted(p)
        if (isPlayerAccepted(p)) {
            throw new KnownPlayer(p);
        }
        // check post condition getNumPlayers() = @pre.getNumPlayers() - 1
        if (getNumPlayers() != pre_getNumPlayers - 1) {
            throw new IllegalNumPlayers(getNumPlayers());
        }
        // check invariant maxNumPlayers > 0
        if (getMaxNumPlayers() <= 0) {
            throw new IllegalMaxNumPlayers(getMaxNumPlayers());
        }
        // check invariant getNumPlayers() <= getMaxNumPlayers()
        if (getNumPlayers() > getMaxNumPlayers()) {
            throw new IllegalNumPlayers(getNumPlayers());
        }
        // check invariant getLeague() != null
        if (getLeague() == null) {
            throw new IllegalLeague();
        }
        // check invariant getName() != null
        if (getName() == null) {
            throw new IllegalName();
        }
    }
    //...
}
```

Figure 10-10 Checking code for `removePlayer()`

10–5 Write checking code for the contracts of `TournamentStyle` and `Round` described in Section 9.6.2. Write checking code for preconditions, postconditions, and invariants

Figures 10-11 and 10-12 depict the checking code associated with the constraints in Section 9.6.2.

.

```
public class TournamentStyle {
    // ...
    public List planRounds(Tournament t) throws IllegalTournament,
        TournamentAlreadyPlanned,PlayerNotAssigned, OverlappingRounds
    {
        // check precondition t <> nil
        if (t == null) {
            throw new IllegalTournament();
        }
        // check precondition t.rounds = nil
        if (t.getRounds() != null) {
            throw new TournamentAlreadyPlanned();
        }
        // check precondition legalNumPlayers(t.players->size))
        if (!legalNumPlayers(t.getNumPlayers())) {
            throw new IllegalNumPlayers();
        }
        // ... do the real work ...
        List rounds = new ArrayList();
        // ....
        // check postcondition
        //  t.getPlayers()->forAll(p|p.getMatches(t)->notEmpty)
        for (Iterator i = t.getPlayers().iterator(); i.hasNext();) {
            Player p = (Player)i.next();
            if (p.getMatches() == null || p.getMatches().size() == 0) {
                throw new PlayerNotAssigned(p);
            }
        }
        // check postcondition result->forAll(r1, r2| r1<>r2 implies
        //      r1.getEndDate().before(r2.getStartDate()) or
        //          r1.getStartDate().after(r2.getEndDate())
        for (Iterator i = rounds.iterator(); i.hasNext();) {
            Round r1 = (Round)i.next();
            for (Iterator j = rounds.iterator(); j.hasNext();) {
                Round r2 = (Round)j.next();
                if (r1 != r2) {
                    if (!r1.getEndDate().before(r2.getStartDate()) and
                        !r1.getStartDate().after(r2.getEndDate())) {
                        throw new OverlappingRounds(r1, r2);
                    }
                }
            }
        }
        return rounds;
    }
}
```

Figure 10-11 Checking code for `TournamentStyle.planRounds()`.


```
public class Round {
    List matches = new ArrayList();
    boolean planned = false;
    boolean completed = false;

    public void plan() throws RoundNotPlanned, PlayerInMoreThanOneMatchInRound {
        previousRoundCompleted = getPreviousRound().isCompleted();
        // ... plan the round;
        // check postcondition
        if (previousRoundCompleted) {
            if (!isPlanned()) {
                throw RoundNotPlanned();
            }
        }
        // check invariant
        for (Iterator i = matches.iterator(); i.hasNext();) {
            Match m1 = (Match)i.next();
            for (Iterator j = m1.getPlayers().iterator(); j.hasNext();) {
                Player p = (Player)j.next();
                for (Iterator k = p.getMatches(); k.hasNext();) {
                    Match m2 = (Match)k.next();
                    if (m1 != m2) {
                        if (m1.getRound() == m2.getRound()) {
                            throw PlayerInMoreThanOneMatchInRound(m1, m2);
                        }
                    }
                }
            }
        }
    }
    public boolean isPlanned() throws IllegalNumPlayersInMatch {
        // check postcondition
        if (planned) {
            for (Iterator i = matches.iterator(); i.hasNext();) {
                Match m = (Match)i.next();
                if (m.getPlayers()->size() !=
                    getTournament().getLeague.getGame().getNumPlayersPerMatch) {
                    throw IllegalNumPlayersInMatch(m);
                }
            }
        }
        return planned;
    }
    public boolean isCompleted() throws NoWinnerInCompletedMatch {
        // check postcondition
        if (completed) {
            for (Iterator i = matches.iterator(); i.hasNext();) {
                Match m = (Match)i.next();
                if (m.getWinner() == null) {
                    throw NoWinnerInCompletedMatch();
                }
            }
        }
        return completed;
    }
}
```

Figure 10-12 Checking code for `Round`.

10–6 Design a relational database schema for the object model of Figure 10-30. Assume `Leagues`, `Tournaments`, `Players`, *and* `Rounds` *have a* `name` *attribute and a unique identifier. Additionally,* `Tournaments` *and* `Rounds` *have* `start` *and* `end` *date attributes. When different transformations are available, explain the trade-off involved.*

Figure 10-13 depicts a sample solution for this exercise. This exercise checks only a basic understanding of the class model to relational schema since most of the tables in this schema already appear in various figures in Chapter 10. The next exercise focuses more on modeling.

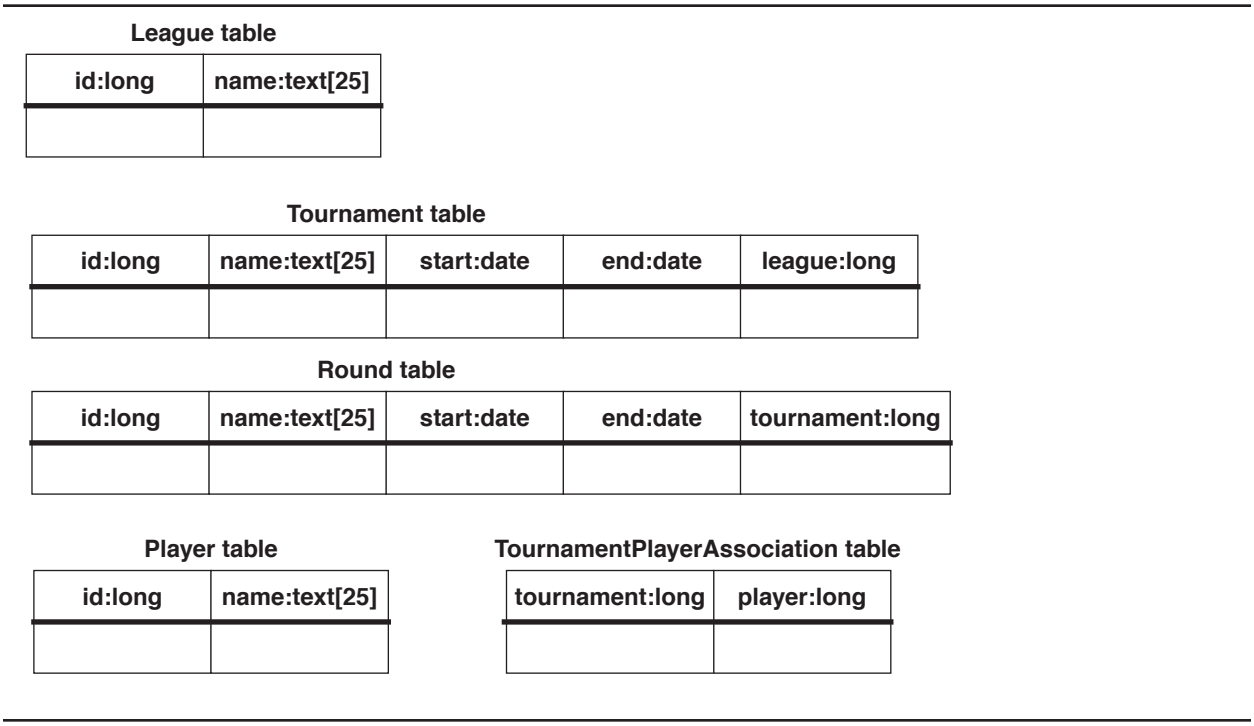


Figure 10-13 Sample solution for Exercise 10–6.

10–7 Draw a class diagram representing the application domain facts below, and map it to a relational schema.

- *A project involves a number of participants.*
- *Participants can take part in a project either as project manager, team leader, or developer.*
- *Within a project, each developer and team leader is part of at least one team.*
- *A participant can take part in many projects, possibly in different roles. For example, a participant can be a developer in project A, a team leader in project B, and a project manager in project C. However, the role of a participant within a project does not change.*

Figure 10-14 depicts a class diagram for the above domain facts and Figure 10-15 depicts the corresponding schema. Worth of note:

- The key to the exercise is to create a `Role` class between the participant and the project.
- The association between `Manager` and `Project` is not explicit in the above facts but should be known to the student from the definition of project manager in the book. As a one-to-one association, it can be represented as a buried association in either the `Manager` or the `Project` table.
- Also, implicit in the above description is that each participant takes part in a project in exactly one role, which would have to be represented with an OCL constraint and checking code. This sample solution does not take into account this constraint in the sense that it would allow a single person to take part in the same project in multiple roles.

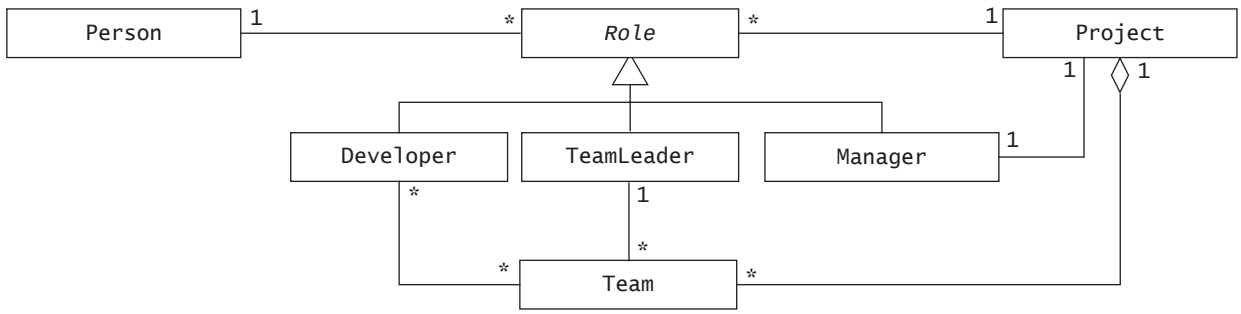


Figure 10-14 Class diagram for Exercise 10–7.

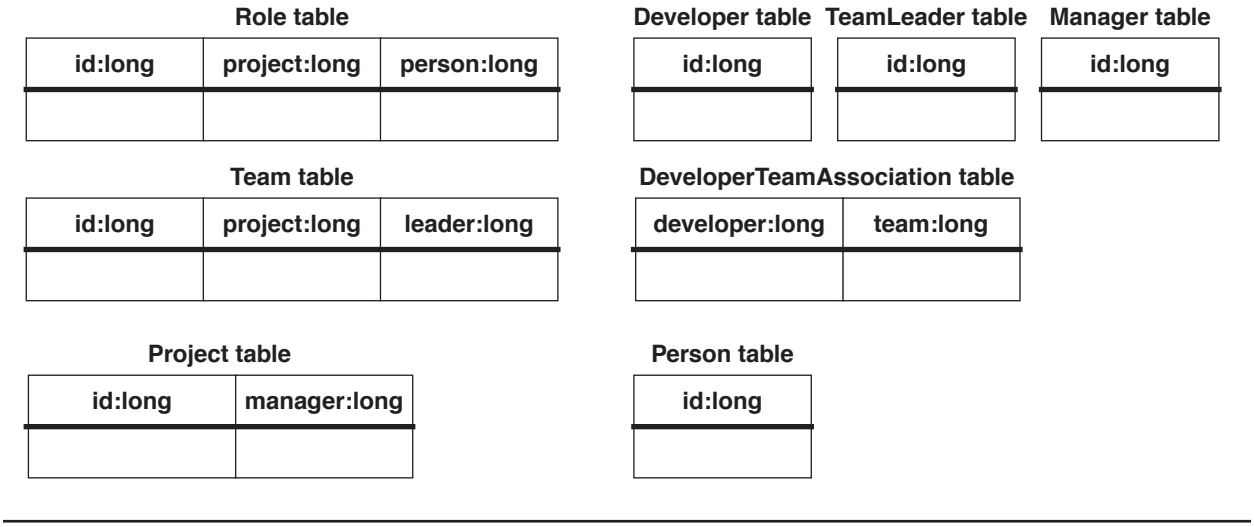


Figure 10-15 Relational schema for Exercise 10–7.

10–8 There are two general approaches for mapping an association to a set of collections. In Section 10.6.2, we map the N-ary association `Statistics` *to two classes, a simple* `Statistics` *class to store the attributes of the association, and a* `StatisticsVault` *class to store the state of the links among the association links. In Section 10.4.2, we described an alternative approach where the association links are stored in one or both classes at the ends of the association. In the event associations were stored in both classes, we added mutually recursive methods to ensure that both data structures remained consistent. Use this second approach to map the N-ary* `Statistics` *association to* `Collections`*. Discuss the trade-offs you encounter and the relative advantages of each approach.*

Mapping the N-ary Statistics association to a set of binary associations which are in turn mapped to a set of collections results in a loss of information and a lot of redundancy. In Figure 10-25, there are only three methods for retrieving statistics objects based on three possible pairs of objects. When mapping binary associations to collections, there will be at least two different methods for each case provided by two different classes. This distribution of code makes it more difficult to maintain it and to extend it when new classes are added to the N-ary association.

The only advantage of the binary association approach is that it is mechanical and could be supported by a code generation tool. While this would take care of ensure that redundant code is realized correctly, it does not address the problem of extensibility.

11. Testing: Solutions

11-1 Correct the faults in the `isLeapYear()` and `getNumDaysInMonth()` methods of Figure 11-12 and generate test cases using the path testing methods. Are the test cases you found different than those of Table 11-14 and Figure 11-13? Why? Would the test cases you found uncover the faults you corrected?

```
public class MonthOutOfBounds extends Exception {…};
public class YearOutOfBounds extends Exception {…};

public class MyGregorianCalendar {
    public static boolean isLeapYear(int year) {
        boolean leap;
        if ((year%400) == 0) {
            leap = true;
        } else if ((year%100) == 0) {
            leap = false;
        } else if ((year%4) == 0) {
            leap = true;
        } else {
            leap = false;
        }
        return leap;
    }
    public static int getNumDaysInMonth(int month, int year)
            throws MonthOutOfBounds, YearOutOfBounds {
        int numDays;
        if (year < 1) {
            throw new YearOutOfBounds(year);
        }
        if (month == 1 || month == 3 || month == 5 || month == 7 ||
                month == 10 || month == 12) {
            numDays = 31;
        } else if (month == 4 || month == 6 || month == 9 || month == 11) {
            numDays = 30;
        } else if (month == 2) {
            if (isLeapYear(year)) {
                numDays = 29;
            } else {
                numDays = 28;
            }
        } else {
            throw new MonthOutOfBounds(month);
        }
        return numDays;
    }
...
}
```

Figure 11-1 Corrected implementation of `getNumDaysInMonth()` and `isLeapYear()` methods (Java).

By applying the path testing method, we find the test cases in Table 11-1. The test cases for `getNumDaysInMonth()` are the same as before since we did not change its flow of control. The test cases for `isLeapYear()`, however, are

different, since we added two more decision points. Note that the additional test cases applied to the incorrect version of `isLeapYear()` would have exercised the bugs.

Table 11-1 Test cases and their corresponding path for the corrected implementation of `getNumDaysInMonth()` an `isLeapYear()` methods.

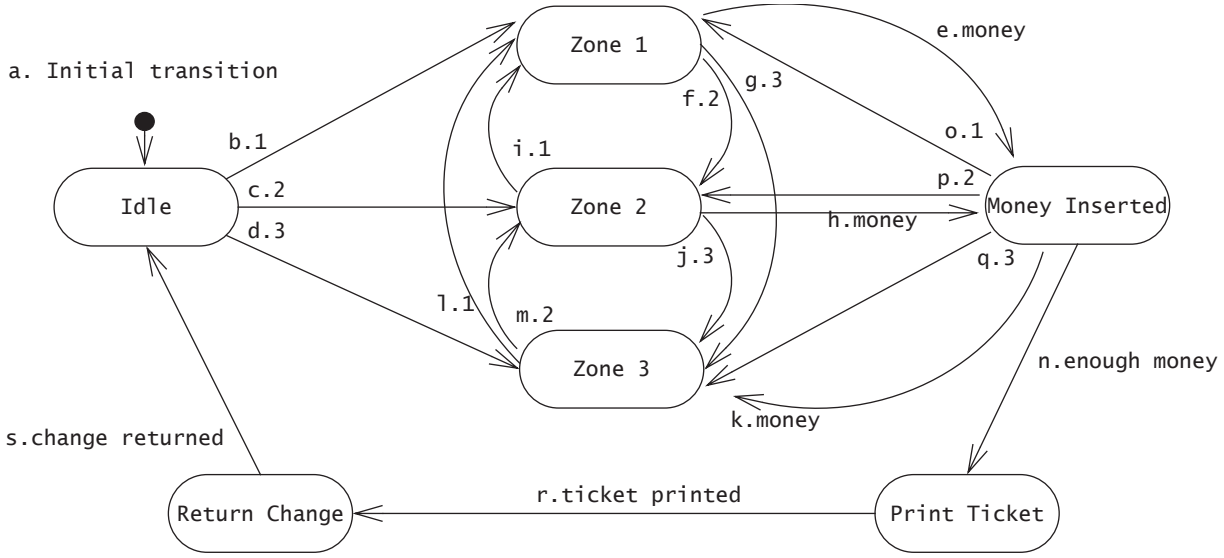
Test case	Path
`For getNumDaysInMonth()`	
`(year = 0, month = 1)`	`{throw1}`
`(year = 1901, month = 1)`	`{n=31 return}`
`(year = 1901, month = 2)`	`{n=28 return}`
`(year = 1904, month = 2)`	`{n=29 return}`
`(year = 1901, month = 4)`	`{n=30 return}`
`(year = 1901, month = 0)`	`{throw2}`
`For isLeapYear()`	
`(year = 2000, month = 1)`	`{leap = true return}`
`(year = 1900, month = 1)`	`{leap = false return}`
`(year = 1904, month = 1)`	`{leap = true return}`
`(year = 1901, month = 1)`	*`{leap = false return}`*

11–2 Generate equivalent Java code for the statechart diagram for the `SetTime` use case of `2Bwatch` (Figure 11-14). Use equivalence testing, boundary testing, and path testing to generate test cases for the code you have just generated. How do these test cases compare with those generated using state-based testing?

State-based testing is similar to applying equivalence testing for every state, hence, equivalence testing applied to a state-driven system results in the same set of test cases. Path testing and boundary testing do not generate any new tests.

11–3 Build the statechart diagram corresponding to the `PurchaseTicket` use case of Figure 11-24. Generate test cases based on the statechart diagram using the state-based testing technique. Discuss the number of test cases and differences with the test case of Figure 11-25.

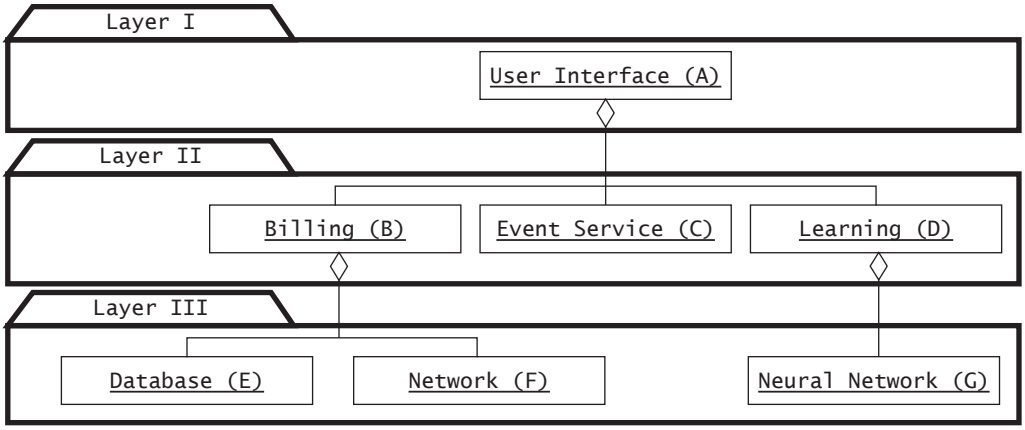
Figure 11-2 depicts a possible statechart diagram and a series of derived test cases for this exercise. The main difference between these test cases and those in the book is that they have been more systematically generated, and thus, cover more possible transitions. It should also be noted, however, that this set of test cases depends on the level of detail statechart diagrams. For example, in this sample solution, we did not differentiate between the case when the user inserts the exact amount and the case when the user inserts more money than necessary. Doing so would have added an additional state and transitions, and consequently, additional test cases. Moreover, this technique fails to detect omissions error as in the case of path testing the `isLeapYear()` method. If behavior is missing from the system or the statechart, it is unlikely that a test case would uncover such a defect.



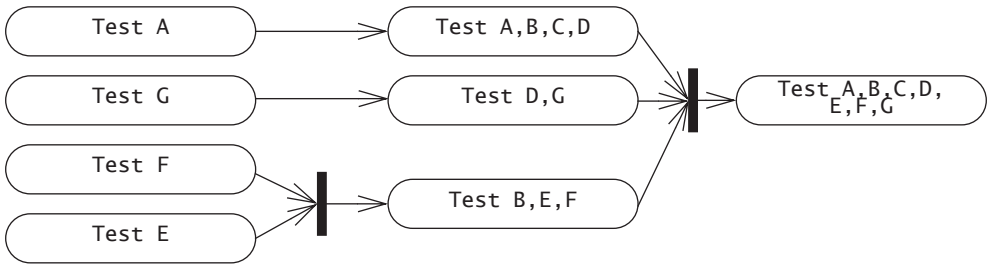
Stimuli	Transition tested	Predicted resulting state
Empty set	a. *Initial transition*	`Idle`
button 1	b.	`Zone 1`
button 2	f.	`Zone 2`
button 1	i.	`Zone 1`
button 3	g.	`Zone 3`
button 1	l.	`Zone 1`
button 2	f. *Put the system into the Zone 2 state to test the next transition.*	`Zone 2`
button 3	j.	`Zone 3`
button 2	m.	`Zone 2`
insert money < zone 2 amount	h.	`Money Inserted`
button 2	p.	`Zone 2`
button 1	i. *Put the system into the Zone 1 state to test the next transition.*	`Zone 1`
insert money < zone 1 amount	e.	`Money Inserted`
button 1	o.	`Zone 1`
button 3	g. *Put the system into the Zone 3 state to test the next transition.*	`Zone 3`
insert money < zone 3 amount	k.	`Money Inserted`
button 3	q.	`Zone 3`
insert money > zone 3 amount	k., n, r, s.	`Idle`
button 2	b.	`Zone 1`
insert money > zone 2 amount	k, n, r, s.	`Idle`
button 3	c.	`Zone 3`

Figure 11-2 UML statechart diagram and resulting tests for the `PurchaseTicket` use case.

11–4 Given the following subsystem decomposition:



comment on the testing plan used by the project manager:



What decisions were made? Why? What are the advantages and disadvantages of this particular test plan?

This test plan was generated according to a sandwich testing plan, without unit testing subsystems from the target layer and without double tests. The advantage is that this test plan consists of fewer tasks. The disadvantage is that failures discovered during the Test A,B,C,D activity will be difficult to locate. A better strategy is to unit test the subsystems in the target layer.

11–5 You are responsible for the integration testing of a system that encrypts network traffic. This system includes a key generator subsystem that uses random numbers. During integration testing, you use a stub implementation of the key generator that produces a predictable result. However, for the release version of the system, you want to substitute the stub implementation with the random implementation, so that the generated keys are not predictable to an outsider. Implement a test infrastructure using one of the design patterns described in Chapter 8, Object Design: Reusing Pattern Solutions to enable the exchange of these two key generator implementations at run time. Justify your choice.

We use a bridge pattern to substitute different implementations of the random number generator at run time.

11–6 Use path testing to generate test cases for all the methods of the `NetworkConnection` *class depicted in Figure 11-15 and in Figure 8-11. Expand first the source code to remove any polymorphism. How many test cases did you generate using path testing? How many test cases would you generate when the source code is not expanded?*

The `receive()` and `setNetworkInterface()` methods would yield only one test case without expanding the source code, since they do not have decision points. However, `receive()` has one polymorphic message send that could be bound to three different classes, resulting in three test cases. `setNetworkInterface()` has two that could be each independently bound to three different classes, resulting in six test cases. See Table 11-2.

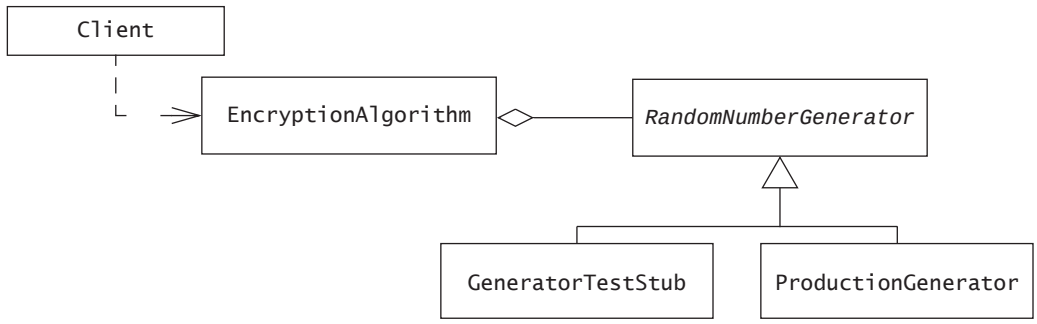


Figure 11-3 Sample solution for Exercise 11–5.

Table 11-2 Number of test cases produced with path testing, with and without source code expansion in Exercise 11–6.

Method	Number of test cases without expanding source code	Number of test cases after expanding the source code
`send()`	2	6
`receive()`	1	3
`setNetworkInterface`	1	6

11–7 Apply the software engineering and testing terminology from this chapter to the following terms used in Feynman's article mentioned in the introduction:

- *What is a "crack"?* [error]
- *What is "crack initiation"?* [error introduction]
- *What is "high engine reliability"?* [high component reliability]
- *What is a "design aim"?* [design goal or a nonfunctional requirement]
- *What is a "mission equivalent"?* [the operation of the engine for 500 continuous seconds, either during a mission or during a test; this term has no equivalent in the testing chapter]
- *What does "10 percent of the original specification" mean?* [the achieved lifetime of the engine is a tenth of the original goal, that is, instead of having a lifetime of 55 mission equivalents, it has a life time of 5.5 mission equivalents]
- *How is Feynman using the term "verification," when he says that "As deficiencies and design errors are noted they are corrected and verified with further testing"?* [In this context, verification means the accumulation of sufficient experimental evidence to convince the designers that an error has been corrected; in software engineering, verification is the development of a proof using formal methods to achieve the same goal]

12. Rationale Management: Solutions

12-1 In Section 12.3, we examined an issue related to access control and notification in the CTC system. Select a similar issue that could occur in the development and CTC and populate it with relevant proposals, criteria, arguments, and justify a resolution. Examples of such issues include:

This exercise is best done in groups of three or more. The goal is not to test the student's knowledge of the CTC systems and its related issues, but rather, to test their ability to structure the debate resulting in this knowledge using rationale techniques.

- *How can consistency between* `mainServer` *and* `hotBackup` *be maintained?*

I[1]: Maintaining consistency between mainServer and hotBackup.

P[1]: The mainServer receives all the requests and forwards them to the hotBackup server after they have been processed. Replies from hotBackup are simply ignored.
A[1] against P[1]: If the mainServer crashes after successfully precessing a request but before forwarding it to the hotBackup, the states of both servers will be inconsistent.

P[2]: Requests to the mainServer are concurrently forwarded to the hotBackup (e.g., by a proxy server standing between the clients and the servers). The replies from both servers are received by the proxy but those from hotBackup are simply ignored.
A[2] for P[2]: This solution can ensure consistency if requests are processed sequentially by both servers and if no processing occurs between requests.
A[3] against P[2]: This adds a component in the architecture (the proxy) which represents a single point of failure.

P[3]: Requests and replies from the main server are first logged and then forwarded to the mainServer or the requesting client, respectively. When switching servers, the hotBackup examines the log of requests to determine the state of the server at the crash time.
A[4] against P[3]: This solution introduces a delay during switching. Given a reliable log, restarting the mainServer may be simpler.

R[1]: P[2] since P[1] does not solve the problem and P[3] is inefficient.

- *How should failure of the* `mainServer` *be detected and the subsequent switch to the* `hotBackup` *be implemented?*

I[2]: How to implement hot switching of servers?

P[2]: Requests to the mainServer are concurrently forwarded to the hotBackup (e.g., by a proxy server standing between the clients and the servers). The replies from both servers are received by the proxy but those from hotBackup are simply ignored.(same as P[2] above). If the main server fails to respond within a time out, it is restarted and the hotBackup becomes the mainServer.
A[5]: Simplicity.

P[4]: A similar proxy server forwards requests to both servers and forwards the first reply that comes back to the client. If the second reply does not occur within a time out, the slower server is restarted.
A[6] for P[4]: Better response time.
A[7] against P[4]: To take advantage of the shorter response time, the proxy has to be multithreaded, adding complexity to a single point of failure.

R[2]: P[4] for response time.

12–2 You are developing a CASE tool using UML as its primary notation. You are considering the integration of rationale into the tool. Describe how a developer could attach issues to different model elements. Draw a class diagram of the issue model and its association to model elements.

This is an open ended question. The main point that should be present in all correct answers is that, ideally, developers should be able to attach any type of rationale node to any type of modeling element. For example, developers could attach issues on individual methods, develop proposals which refer to proposed additional methods

and attributes, and discuss arguments which refer to existing model elements. The creation of two abstract classes, `ModelElement` and `RationaleNode`, would make this type of functionality easier to implement (see Figure 12-1).

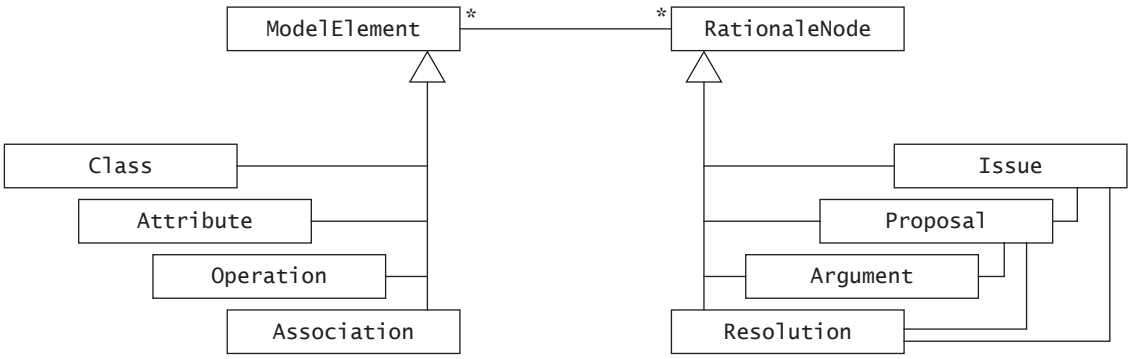


Figure 12-1 Example class diagram for exercise 12–3 (some classes omitted for clarity)

12–3 Below is an excerpt from a system design document for an accident management system. It is a natural language description of the rationale for a relational database for permanent storage. Model this rationale with issues, proposals, arguments, criteria, and resolutions, as defined in Section 12.3.

> One fundamental issue in database design was database engine realization. The initial nonfunctional requirements on the database subsystem insisted on the use of an object-oriented database for the underlying engine. Other possible options included using a relational database, a file system, or a combination of the other options. An object-oriented database has the advantages of being able to handle complex data relationships and is fully buzzword compliant. On the other hand, OO databases may be too sluggish for large volumes of data or high-frequency accesses. Furthermore, existing products do not integrate well with CORBA, because that protocol does not support specific programming language features such as Java associations. Using a relational database offers a more robust engine with higher performance characteristics and a large pool of experience and tools to draw on. Furthermore, the relational data model integrates nicely with CORBA. On the downside, this model does not easily support complex data relationships. The third option was proposed to handle specific types of data that are written once and read infrequently. This type of data (including sensor readings and control outputs) has few relationships with little complexity and must be archived for extended periods of time. Files offer an easy archival solution and can handle large amounts of data. Conversely, any code would need to be written from scratch, including serialization of access. We decided to use only a relational database, based on the requirement to use CORBA and in light of the relative simplicity of the relationships between the system's persistent data.

This exercise tests the student's knowledge of issue models. The student can either draw a graph or use idented text to represent the resulting issue model. The text below shows a simple representation using idented text and tags. Once the issue model is completed, the reader may notice that some of the proposals were not thoroughly discussed and argued and that one of the pseudo requirements (modeled as a criterion) was not met.

Key: I[1] = issue 1, P[1] = proposal 1, A[1] = argument 1, C[1] = criterion 1, R[1] = resolution to I[1].
I[1]: Which database engine realization?
    P[1]: OO DBMS.
        A[1] for P[1]: Can handle complex data relationships.
        A[2] for P[1]: Buzzword compliant. (see C[1])
        A[3] against P[1]: Sluggish for large volumes of data or high-frequency accesses.
        A[4] against P[1]: Existing products do not integrate well with CORBA. (see C[2])
    P[2]: Relational DBMS.

A[5] for P[2]: More robust engine than OO DBMS.
A[6] for P[2]: Higher performance characteristics.
A[7] for P[2]: Large pool of experience.
A[8] for P[2]: Some relational DBMS integrate well with CORBA.(see C[2])
P[3]: File system
A[9] for P[3]: Good for serial data with simple relationships, including sensor data.
A[10] against P[3]: Need to rewrite many programs to provide similar functionality than a DBMS, including serialization of access.
P[4]: Combination of P[1,2,3].
C[1]: Database subsystem should use an OO database.(Pseudorequirement)
C[2]: Needs to co-exist with CORBA. (Prior design decision)
R[1]: P[2], because of C[2] and simplicity of relationships of system's persistent data.

12–4 Consider the NFR Framework described in Section 12.3.7. Draw a QOC model that is equivalent to the goal graph depicted in Figure 12-12. Discuss the respective advantage and disadvantages of QOC and the NFR Framework for representing rationale during requirements.

Figure 12-2 depicts a QOC model equivalent to the goal graph. The advantages of the NFR Framework over QOC are that

- goals (questions) are represented as an AND-OR refinement a hierarchy
- the satisfied goals for a given option are more easily visible

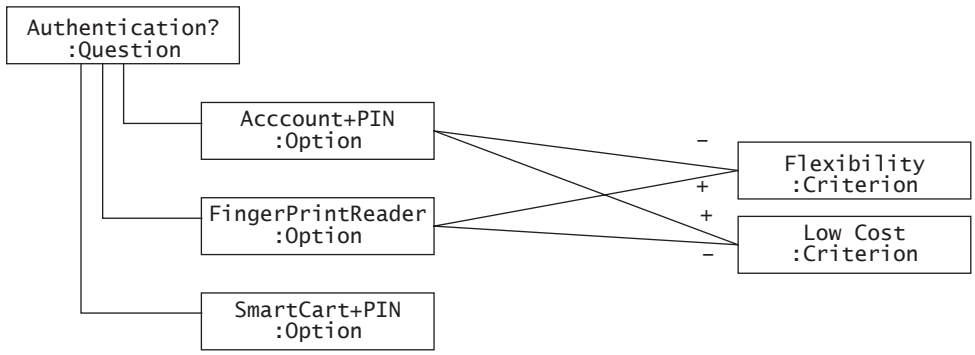


Figure 12-2 QOC model for Exercise 12–4.

12–5 You are integrating a bug reporting system with a configuration management tool to track bug reports, bug fixes, feature requests, and enhancements. You are considering an issue model for integrating these tools. Draw a class diagram of the issue model, the corresponding discussion, configuration management, and bug-reporting elements.

This exercise can be answered only after students have read Chapter 13, *Configuration Management*. This exercise is with Chapter 12, however, since it is relevant to rationale management.

This exercise is also an open-ended question, as exercise 12–2 The solutions of this exercise, however, should different significantly from those of exercise 12–2, as the integration problem is only superficially related.

Problem reporting and configuration management tools are usually based on the following workflow: a change is requested, the change is discussed, assessed and planned, and the change is realized, test, and incorporated into a future release. In this exercise, change requests can be feature requests or bug reports. During the assessment of these proposed changes, developers raise issues, propose alternative solutions, and eventually come to a consensus in the form of a resolution. This resolution is then realized as a change.

In the class diagram representing these concepts (Figure 12-2), the main associations between both models are between Change Request and Issue, and between Resolution and Change. Note that both associations are many to many.

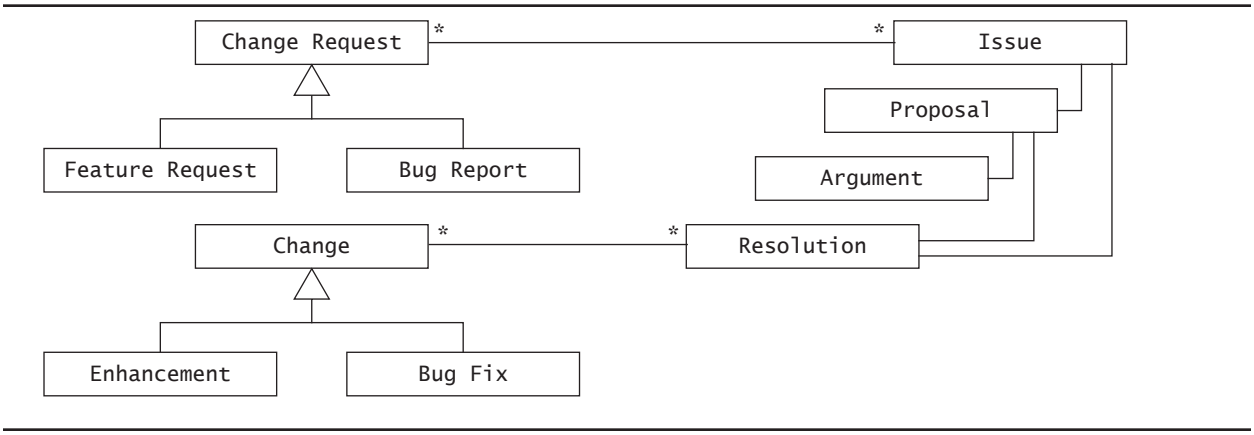


Figure 12-3 Example class diagram for exercise 12–3 (some classes omitted for clarity)

13. Configuration Management: Solutions

13-1 RCS adopts a reverse delta approach for storing multiple versions of a file. For example, assume a file has three revisions, 1.1, 1.2, and 1.3, RCS stores the file as of version 1.3, then, the differences between 1.2 and 1.3, and the differences between 1.1 and 1.2. When a new version is created, say 1.4, the difference between 1.3 and 1.4 is computed and stored, and the 1.3 version is deleted and replaced by 1.4.
Explain why RCS does not simply store the initial version (in this case 1.1) and the differences between each successive version.

The trade-off between reverse deltas and forward deltas is a response time trade-off. In the case of reverse deltas (as RCS does), the time to retrieve the latest version is the shortest, while time to retrieve the oldest versions is the longest. In the case of forward deltas, the time to retrieve the initial (oldest) version is the shortest. In most projects, retrieving the latest version is the most common case. Note that storing reverse deltas increases the time for creating a new version.

13–2 CVS uses a simple text-based rule to identify overlaps during a merge: There is an overlap if the same line was changed in both versions that are being merged. If no such line exists, then CVS decides there is no conflict and the versions are merged automatically. For example, assume a file contains a class with three methods, `a()`*,* `b()`*, and* `c()`*. Two developers work independently on the file. If they both modify the same lines of code, say the first line of method* `a()`*, then CVS decides there is a conflict. Explain why this approach will fail to detect certain types of conflict. Provide an example in your answer.*

This approach fails when developers introduce conflicts when modifying different lines of a file. For example, if two developers introduce a new field with the same name in the same class in different locations in the file. Note that developers can also introduce semantic conflicts by modifying two different files (e.g., by making different assumptions about the same class).

13–3 Configuration management systems such as RCS, CVS, and Perforce use file names and their paths to identify configuration items. Explain why this feature prevents the configuration management of CM aggregates, even in the presence of labels.

Note to the instructor: this is a difficult question for students who have not used configuration management systems, in particular branch management.

The main issue with using path names for identifying configuration items is that path names are attributes of configuration items and thus, can also change. Assume for example that directories represent subsystems (e.g., storage, notification, user interface, train tracking). In the case of a change of the subsystem decomposition, the directory structure may change (e.g., the storage directory may be split into a database directory and a JDBC directory). For tools which use path names as identifiers, this translates into configuration items being deleted and added under a different path name. Since different versions of the same configuration items have different identifiers, operations such as merging branches are consequently not possible.

13–4 Explain how configuration management can be beneficial to developers, even in the absence of a change control or auditing process. List two scenarios illustrating your explanation.

Scenario 1: Assume developers work on new features sequentially and create new versions of the systems as a feature becomes stable. Assume the current version is *n* and that versions *m* through *n* contain partial implementation of the latest feature *F*. The implementation of *F* unfortunately, introduces many issues and developers decide to rethink their approach. They find a better way to implement *F*, however, it requires them to undo all changes related to *F*. When using a version control tool and the sequential configuration management policy described before, this can be simply done by checking out version *m-1* and discarding versions *m* through *n*.

Scenario 2: Branches can be used to manage the development of concurrent features. If the implementation of all features succeed, the branches need to be merged. If the development of one feature fails as in scenario 1, the corresponding branch can be ignored while the rest of the development proceeds.

13–5 In Chapter 12, Rationale Management, we described how rationale information can be represented using an issue model. Draw a UML class diagram for a problem tracking system that uses an issue model for the description and discussion of changes and their relationship with versions. Focus only on the domain objects of the system.

Clarification: "domain objects" here means "entity objects".

This exercise has many different possible solutions since it is a design exercise. Figure 13-1 depicts a possible solution. The key element is that the release of a new version triggers the discovery of new problems, which are then discussed (cause identified, solution proposed and evaluated, and decision made). The decision then results in a new version aimed at correcting the new problem.

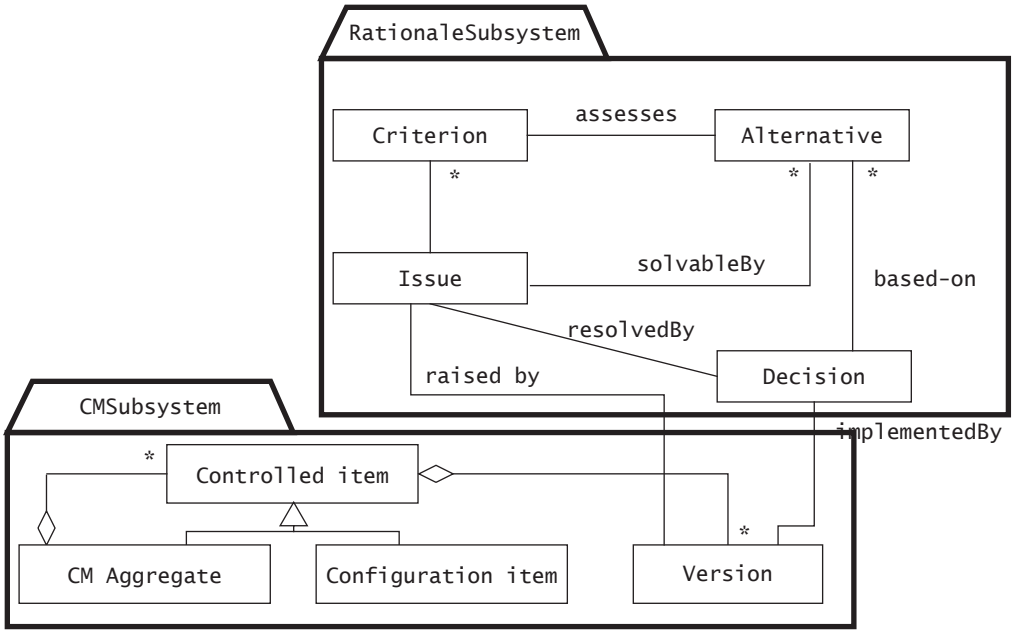


Figure 13-1 Example solution for exercise 10.5.

13–6 In Chapter 11, Testing, we described how the quality control team find faults in promotions created by subsystem teams. Draw a UML activity diagram including the change process activities and testing activities of a multiteam project.

Figure 13-10 in the book is a sample solution to this problem. The key elements the instructor should look in student solutions are:

- Testing is performed by the quality control team.
- The quality control team sends change requests or problem reports to the subsystem teams.
- Changes are performed by the subsystem teams.
- Subsystem teams send promotions to the quality control team.
- The subsystem teams decide when to create promotions.
- The quality control team decides when to create a release.

14. Project Management: Solutions

14-1 In Figure 14-1, we modeled the phases of a project with a state chart. Use the state pattern by making each of these phases a different class. Use the project management activities introduced in this chapter to make them public operations on these classes. Assume a team-based project organization.

Figure 14-1 depicts a sample solution to this exercise. The *ProjectPhase* class defines operation for each project management activity class. Each concrete state class (one per phase) defines operations for those operations that it can handle. All other operations are implemented so that they trigger a change of state. The context is stored in the `Project` class.

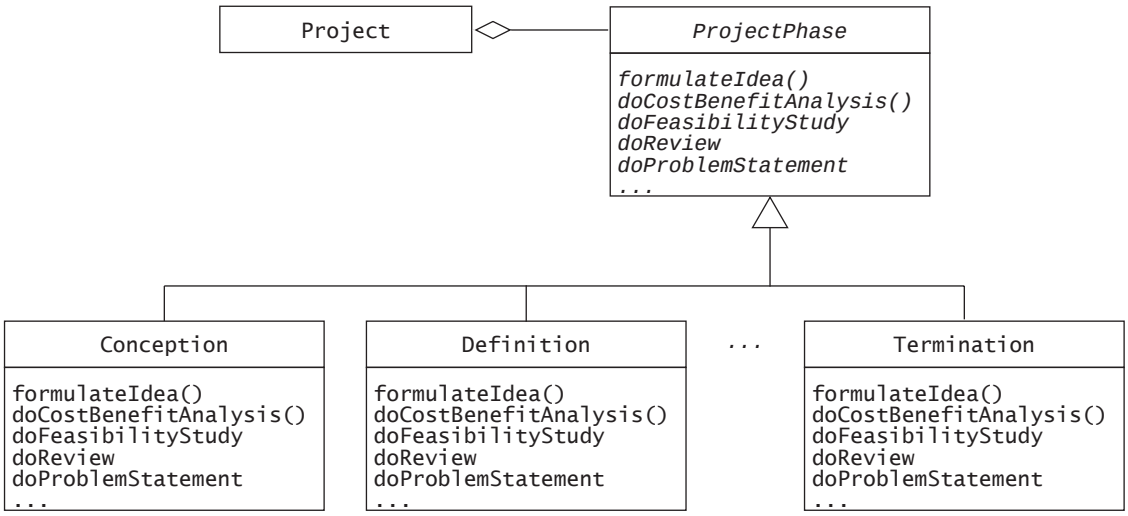


Figure 14-1 Sample solution for Exercise 14-1.

14–2 What are the relative advantages of flat staffing versus gradual staffing?

The advantage of gradual staffing is that saves resources in the early part of the project. The advantage of flat staffing is that all the ramp up effort (e.g., training, team formation, definition of procedures, etc.) is moved up front.

14–3 What is the difference between status meetings and decision meetings? Should they always be kept separate in a meeting?

The objective of a status meeting is for team members to report status to the team leaders, in an honest, verifiable, and unbiased manner. This enables the team leader to track progress and plan corrective decisions if necessary, such as adding resources or creating new communication paths.

The objective of a decision meeting is either, for a team leader to communicate a decision to the team and ensure it is implemented through action items, or for a team to discuss a set of issues and achieve a consensus.

It is better to keep status and decision separate, such that team members do not feel punished for reporting problems. A team leader needs the team members to report problems as soon as possible and can encourage this if the team leader gives the opportunity for the team to resolve the problem first by themselves. If the problem worsen, then the team leader may decide to intervene or to call a decision meeting in which the team cooperatively designs a solution. In any case, there should be a time interval between the reporting of a problem and its corresponding intervention.

14–4 Why should the role of architect and team leader be assigned to distinct persons?

The role of architect requires good technical abilities. The role of team leader requires good management abilities. Both kinds of abilities are rarely present in the same person. Both roles are time consuming can rarely be accomplished well by the same person at once.

14–5 Draw a UML model of the team organization of the OWL project for each the three main phases (i.e., initiation, steady state, termination).

An example solution to this exercise is depicted in Figure 14-1. During project initiation, the management team is responsible for the planning and other management activities, while the requirements team defines the system with the users and the client and the architecture team design a preliminary architecture. During steady state, a team for each subsystem is created, along with a documentation team and a quality control team. During project termination, the subsystem teams are replaced with a single installation team which is responsible for deploying the system and fixing any last minute problems discovered by the quality control team.

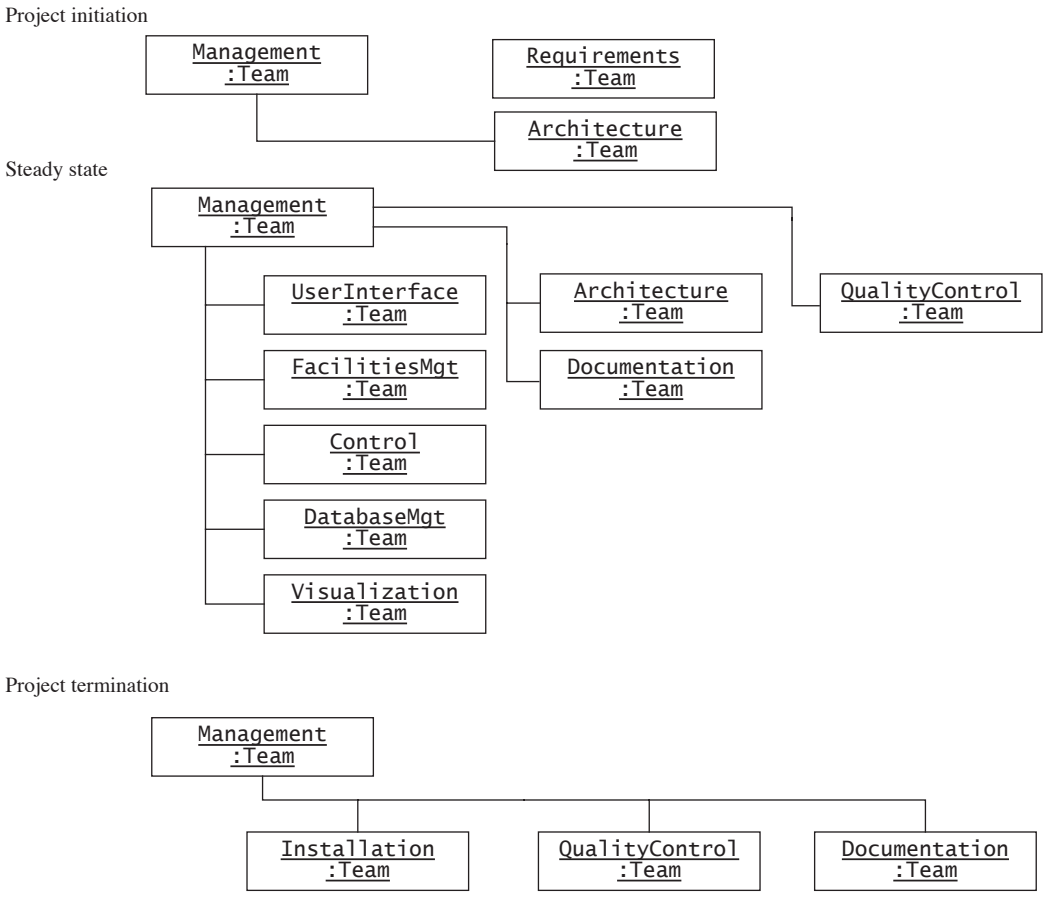


Figure 14-2 Example solution for exercise 11.4 (UML class diagram).

14–6 Draw a task model of the system design of the MyTrip system presented in Chapter 6, System Design: Decomposing the System.

A task plan can be represented as an activity diagram. The MyTrip system was used as an example to illustrate each system design activity. Hence, the activity diagram of Figure 6-28 can be used as a starting point for the solution to this exercise. We remove the "Implement Subsystem" activity (it is not part of system design) and add a "Refine subsystems" activity to plan for a second iteration (see Figure 14-2). A conservative manager may even want to serialize all tasks instead of conducting the activities of system design in parallel.

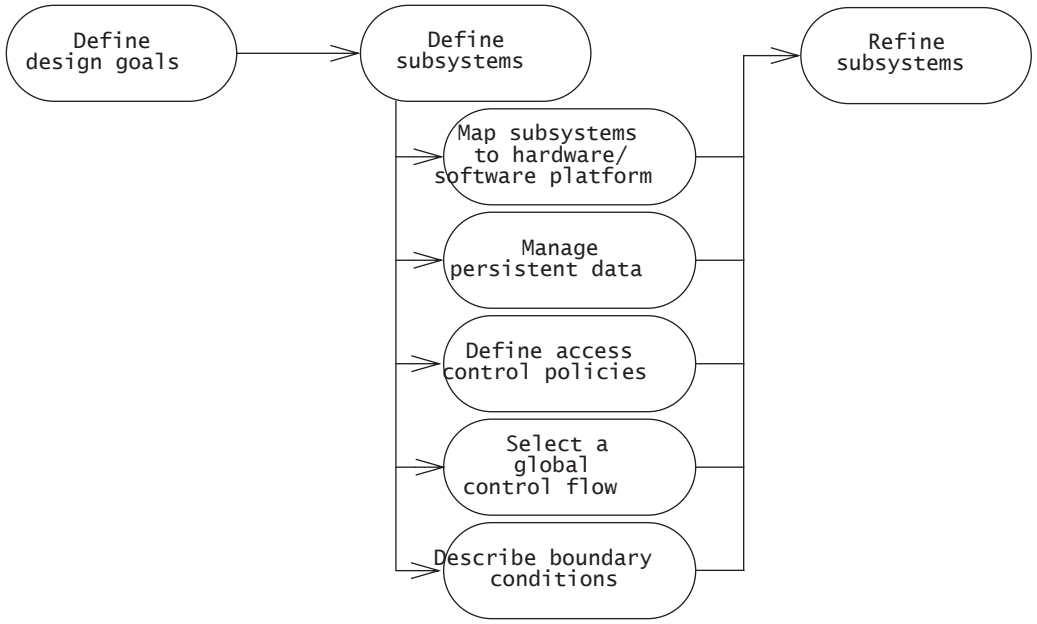


Figure 14-3 Example solution for exercise 11.5 (UML activity diagram).

14–7 Estimate the time to complete each task of the task model produced in Exercise 14–6 and determine the critical path.

Note to the instructor: the estimates provided by the students need not (and will not) be realistic. The goal of the exercise is to check the understanding of the concepts of critical path and task dependency. The example solution in Figure 14-3 depicts the task model as a PERT chart and indicates the critical path in bold.

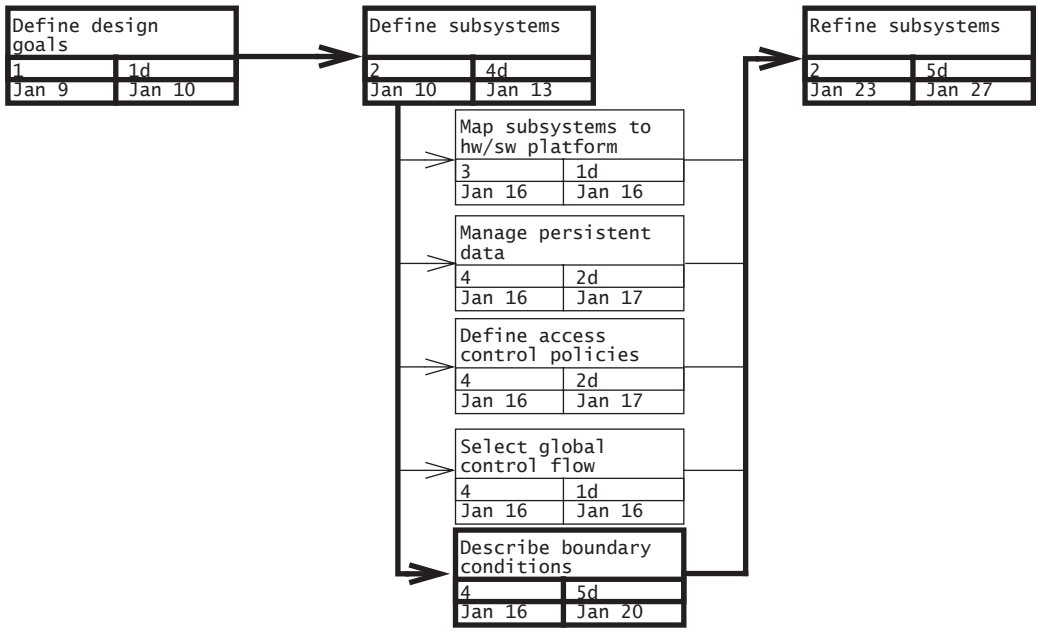


Figure 14-4 Example solution for exercise 11.6 (PERT chart).

14–8 Identify, prioritize, and plan for the top five risks related to the system design of MyTrip presented in Chapter 6, System Design: Decomposing the System.

Note to the instructor: Again, the risks provided by the students need not be realistic. They do need, however, to be risks. The goal of the exercise is to make the students think about what could go wrong and to check their understanding of the risk concept. Table 14-1 is an example solution of risks and corresponding mitigation strategies. The risks are ordered by priority (most important first).

Table 14-1 MyTrip risks

Risk	Mitigation strategy
MyTrip presents a safety hazards for drivers.	• Design a user interface that only allows passengers to consult trips. • Describe the risks of using MyTrip while driving on the first page of the user manual.
MyTrip is not secure.	• Plan for security tests by the quality control team. • As part of maintenance, plan for a security team to continuously test and address security vulnerabilities.
MyTrip is not reliable.	• Underestimate reliability of components, in particular, the hardware components over which the project has no control: phone network, home computer, etc. • Plan for the release of patches and upgrades to address serious bugs.
A competitor develops conflicting standards for storing and planning trips	• Design for modifiability.
The number of users and trips were underestimated. MyTrip does not scale beyond requirements	• Design for more users and trips than required.

14–9 Identify, prioritize, and plan for the top five risks related to the user interface subsystem of CTC presented in Chapter 12, Rationale Management.

Same exercise as before with a different application domain. Table 14-1 depicts an example solution.

Table 14-1 CTC user interface risks

Risk	Mitigation strategy
The user interface does not meet the availability criteria.	• Plan for extensive field tests before deployment. • Plan for an emergency maintenance team to address any serious problems. • Plan to roll back to the older system if the availability criteria is not met.
The user interface consumes too many system resources.	• Plan for early performance testing. • Select hardware that can accommodate more load than initially required.
The user interface does not meet the usability criteria	• Plan for usability tests with a representative set of users.

Table 14-1 CTC user interface risks

Risk	Mitigation strategy
The user interface represents a paradigm shift (i.e., the users might be confused by the new interface)	• Ensure that users have sufficient training before they dispatch trains with the new system.
The user interface does not scale to a larger number of trains and dispatchers	• Design for a larger number of trains and dispatchers than initially required.

14–10Linux, developed using the bazaar model, is more reliable and more responsive than many operating systems running on Intel PCs. Discuss in some detail why the bazaar model should, however, not be used for the Space Shuttle control software.

One of the key of the bazaar model is that many users are actually developers willing to diagnose problems and propose solutions. The system, once released, goes through a large scale and parallel test which will find most of the important problems. The Space Shuttle does not have many users. Moreover, the price of discovering a bug in the field is much more serious.

14–11Organize the project participants into teams of four people. Each team has the following resources available: 2 eggs, 1 role of TESA film, 1 role of toilet paper, a cup with water, a bucket with two liters of sand, 20 foam balls (each about 1 cm diameter), 1 table whose surface is about 1 meter above the ground. Each team has 25 minutes time to build and test an artifact that allows an egg to be released 75 cm above the table such that the egg falls on the table without cracking. Each team has another 5 minutes to demonstrate the artifact to project management.

This is a team building exercise that forces groups of individuals who do not know each other to work together, identify each individuals strengths, and recognize that some problems are better solved in a collaborative environment. Such exercises are done at the beginning of a project to form teams.

14–12Organize the project participants into teams of four people. Each team has the following resources available: 2 buckets of DUPLO pieces, 2 tables separated by a distance of 1.50 meters. Each team has 25 minutes time to build and test a bridge solely out of the available DUPLO pieces that hangs free for at least one minute between the two table surfaces. Each team has another 5 minutes to demonstrate the artifact to project management.

This is another team building exercise like Exercise 14–11.

14–13Specify all management models for the icebreaker project described in Exercise 14-11.

After the completion of Exercise 14–11, the instructor should select a team that displayed a minimum of structure during the problem solving (as opposed to a single participant coming up with the solution) and ask participants reverse engineer a schedule, roles, teams, and a task model. This should illustrate both the usefulness of such models to reason about the team dynamic and their limitations.

14–14Write an SPMP for the icebreaker project described in Exercise 14-11.

This exercise has the same purpose as Exercise 14–13.

15. Software Life Cycle: Solutions

15-1 Model as classes the activities of Figure 15-2 and the work products of Figure 15-4 and draw a UML class diagrams depicting the relationships between activities and work product.

Figure 15-5 in the book is a possible solution.

15–2 Assume that the waterfall model shown in Figure 15-8 has been derived from the IEEE standard model in Figure 15-7 during the activity Life Cycle Modeling. What processes and activities have been omitted in the waterfall model?

The goal of this exercise is to have students think about processes which are needed all the time, e.g., configuration management. The following processes were omitted from the waterfall model:

- Project management processes: Project Monitoring & Control, Software Quality Management,
- Integral processes: Configuration Management, Documentation Development, Training,
- Post development processes: Maintenance, Retirement.

The project management and integral processes that were omitted run for the complete duration of the project, and thus, do not fit in the sequential model of the waterfall. The Maintenance and Retirement processes could simply added at the bottom of the water fall. They were probably omitted because those activities are not considered part of development.

15–3 Redraw Boehm's spiral model in Figure 15-10 as a UML activity diagram. Compare the readability of the original figure with the activity diagram.

Figure 15-1 depicts an example solution for this exercise. Without the spiral metaphor, it is difficult to understand that the life cycle is incremental and iterative.

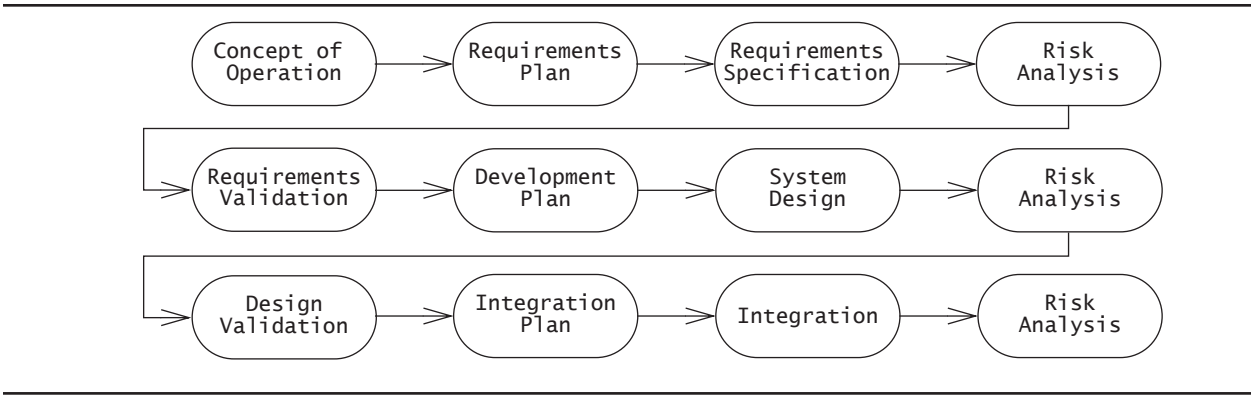


Figure 15-1 Activity diagram for exercise 12.5 (only first three cycles shown)

15–4 Draw a UML activity diagram describing the dependency between activities for a life cycle in which requirements, design, implementation, test, and maintenance occur concurrently. (This is called an evolutionary life cycle.)

Figure 15-2 depicts an example solution for this exercise.

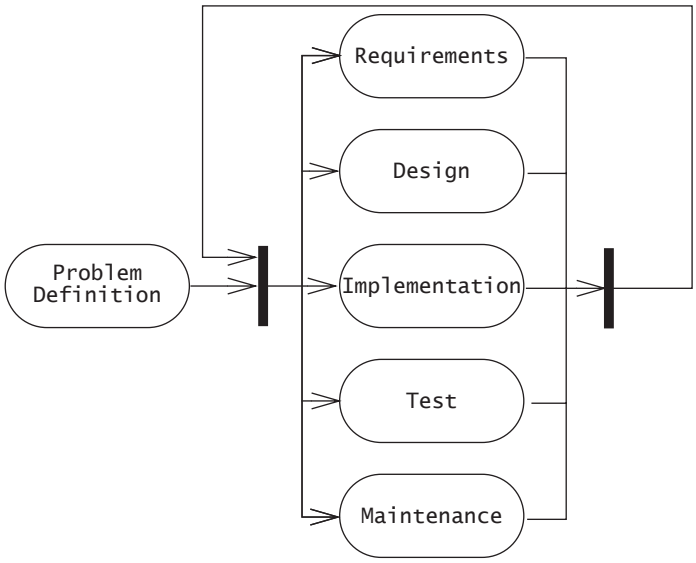


Figure 15-2 Activity diagram for exercise 12.6

15–5 Describe how testing activities can be initiated well before implementation activities. Explain why this is desirable.

Testing activities can be initiated as soon as requirements activities are completed. System tests are generated from the requirements. Integration tests are generated from the system design. Unit tests are generated from the object design. Initiating testing activities early has two main advantages:

- Some types of faults can be found merely by developing test cases, before executing them (e.g., interface issues).
- The test cases will not be biased by the implementation, and thus focus on the requirements of the system.

15–6 In project management, a relationship between two tasks is usually interpreted as a precedence relationship; that is, one task must complete before the next one is initiated. In a software life cycle, a relationship between two activities is a dependency; that is, one activity uses a work product from another activity. Discuss this difference. Provide an example in the context of the V-model.

A precedence relationship indicates that a task must end before the succeeding task can start. A dependency relationship, however, does not always imply that the dependent activities must proceed in sequence. In the V-model, for example, the component integration and test activity depends on both the preliminary design activity (for the subsystem decomposition) and the unit test activity (for the unit tested system). The component integration and test activity can start with the planning of the integration tests as soon as the preliminary design is complete, which can occur even before the unit test activity is initiated.

15–7 Assume you are part of the IEEE committee that will revise the IEEE 1074 standard. You have been assigned the task of modeling communication as an explicit integral process. Make a case for the activities that would belong to this process.

Since communication would be an integral process, all activities in this process group would be process independent and phrased in similar terms as the configuration management or documentation activities. Examples would include:

- Plan communication infrastructure
- Deploy communication infrastructure
- Plan reviews
- Define change request procedures
- Plan issue tracking
- etc.

16. Methodologies: Putting It All Together

16-1 What is the difference between a software life cycle and a methodology?

A software life cycle is a model of the activities and work products necessary to develop a system. A methodology is a collection of methods and tools for developing a system and specifies when methods or tools should be used and what to do if unexpected events occur. Methodologies and life cycles are complementary: a software life cycle can be realized using a variety of software engineering methodologies. Although specific methodologies may imply a specific life cycle, in general, a methodology can be applied in a different life cycles.

Examples of life cycles: Boehm's spiral model, the Unified Process, the Waterfall model.

Examples of methodologies: Royce's unified framework, Extreme Programming.

16–2 You are improving a legacy system. You need to customize a software life cycle for your project. Which activities would you need? In which order?

Any answer to this question should include a life cycle with an inventory analysis activity (to assess what work products exist), one or more reverse engineering activities (to recreate missing work products), and change activities to change or improve the aspect of the legacy system under consideration.

16–3 Royce uses a management metric that counts the number of participants who leave the project or join the project. You are managing a multi-team project and notice that the staff turnover in one team is high. Hypothesize what causes could lead to such a symptom. For each cause, propose a remedy.

High-staff turnover is a negative indicator. It points to a fundamental problem that can only worsen if not addressed. Some of the cause and remedies are depicted in Table 16-1

Table 16-1

Cause	**Remedy**
Unclear or unrealistic goals	Provide better tools to meet the goals or redefine goals.
Unrealistic schedule	Revisit schedule or increase resources.
Conflict with team leader	Replace team leader.
Skills needed for the team are better rewarded in other organizations.	Revise reward structure or determine if required skills are needed.

16–4 The heuristics we outlined in Section 16.5.4 indicate that the need for models is higher in distributed organizations. Open-source projects are highly distributed projects that follow an entity-based life cycle and typically do not have requirements or system design documents. Provide examples of how modeling knowledge is made explicit and transferred among participants in such cases.

Open source projects typically feature models that can be easily refined or extended incrementally. Examples include information stored in change tracking logs, configuration management logs, frequently asked questions, and how-tos.

16–5 Royce's methodology considers six project factors (scale, stakeholder cohesion, process flexibility, process maturity, architectural risk, and domain experience) when tailoring a process for a specific project (Section 16.4.1). Use these factors to describe which types of projects could use XP as an appropriate methodology. Justify your choice for each factor.

Table 16-2 discusses the types of projects for each factor.

Table 16-2

Factor	XP value
Scale	Low to medium. The maximum number of participants that XP can accommodate is limited to the number of participants who can meet at the same time during the daily stand-up meeting.
Stakeholder cohesion	High. As many requirements change during the project, it is critical that the client and developers be able to agree informally on important decisions.
Process flexibility	High. XP has been designed for projects that feature a client who is willing to be collocated with the project and to refine requirements and estimates at the beginning each iteration.
Process maturity	Medium. XP projects need a critical mass of experienced participants to be effective. This knowledge is then transmitted to the novice participants during pair programming and peer reviews. An insufficient number of experienced developers knowledgeable in XP will result in difficulty to manage the expectations of the client.
Architectural risk	High. XP can accommodate architectural changes until late in the project, as the architecture is incrementally refined with the rest of the code.
Domain experience	Low. A collocated client enables developers to learn domain knowledge rapidly.

16–6 While working with Staphylococcus bacteria in 1928, Sir Alexander Fleming accidently dropped some bread crumbs on one of the dishes. After a week the bacteria on this dish did not grow as expected. Fleming noticed a bacteria-free circle around a mold that was contaminating the staphylococcus culture. Instead of throwing the dish away because the planned experiment had failed, he isolated the mold, grew it in a fluid medium, and discovered a substance that prevented growth of the bacteria even when it was diluted 800 times. Discuss the discovery of penicillin using the terminology and issues introduced in this chapter.

This is an example of redefining project goals. The initial experiment of Fleming failed, as he contaminated the petri dish. However, Fleming recognized that he discovered a substance that was much more significant than the original experiment.

16–7 Based on the assumption that the earth was round, Columbus's goal was to find a shorter way to India by going west instead of east. He encountered America instead of India. Discuss the discovery of America by Columbus using the terminology and issues introduced in this chapter.

Columbus and his contemporaries thought he succeeded in discovering a new route to India. From his perspective, he succeeded. Later, after maritime routes around Africa were discovered and when Magellan's crew circumvented the globe, it became clear that Columbus had found a new continent. In the terminology introduced in Chapter 1, this is also an example of how a theory is devised, validated, and falsified, as knowledge is accumulated and tested against the theory.

16–8 Select a project you have been involved with. Categorizing into methodological issues as defined in this chapter, describe the methodological trade-offs that occurred in the project.

The discussion should include a description of the project environment followed by a discussion of each methodological issue, including the options that were considered and those that were selected.

