David Kung

Object-Oriented Software Engineering

An Agile Unified Methodology

Solutions Manual

Message to Instructors

July 10, 2013

The solutions provided in this manual may not be complete, or 100% correct, due to my limitation and the nature of some software engineering problems. Although I have tried to check and correct grammar errors and typos, I am sure the manual may still have many. I will continue to improve it and apologize for any inconvenience that may cause to you.

Dave

Contents

Part I

INTRODUCTION AND SYSTEM ENGINEERING

Chapter 1

Introduction

1.1 Search the literature and find four other definitions of software engineering in addition to the one given in this chapter. Discuss the similarities and differences between these definitions.

Solution. Below are five definitions of software engineering including the one in the textbook, listed chronologically. The similarities and differences are shown in Figure 1.1. A better solution should also provide a convincing explanation of the differences, and significant implications of the differences. For example, software engineering education, and significant improvement of software PQCT are important considerations of software engineering.

1. **IEEE.** The IEEE Computer Society defines software engineering as: "(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1)." ("IEEE Standard Glossary of Software Engineering Terminology," IEEE std 610.12-1990, 1990.)

2. **Ghezzi.** "Software engineering is the field of computer science that deals with the building of software systems that are so large or so complex that they are built by a

	IEEE	Ghezzi	Brugge	Sommerville	Kung
(1) Application of engineering to software	√	√		√	√
(2) Study of approaches as in (1)	√				√
(3) Modeling, problem-solving, knowledge acquisition, and rationale driven activity	√	√	√	√	√
(4) Education of engineering processes and methodologies					√
(5) Significantly improve software PQCT (that is, engineering and business aspects)					√

Figure 1.1: Similarities and differences between SE definitions

team or teams of engineers." It is "the application of engineering to software." (Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli, "Fundamentals of Software Engineering," 2nd Edition, Prentice Hall, 2003.)

3. **Brugge and Dutoit.** Software engineering is a *modeling, problem-solving, knowledge acquisition*, and *rationale-driven* activity. (Bernd Brugge and Allen H. Dutoit, "Object-Oriented Software Engineering Using UML, Patterns, and Java," 3rd Edition, Prentice Hall, 2010.)

4. **Sommerville.** "Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use." (Ian Sommerville, "Software Engineering," 9th Edition, Addison-Wesley, 2011.)

5. **Kung.** "Software engineering as a discipline is focused on the research, education, and application of engineering processes and methods to significantly increase software productivity and software quality while reducing software costs and time to market." (David Kung, "Object-Oriented Software Engineering: An Agile Unified Methodology," McGraw-Hill Higher Education, 2013.)

1.2 Describe in a brief article the functions of software development process, software quality

assurance, software project management, and software configuration management. Discuss how these work together during the software development life cycle. Discuss how they improve software PQCT.

Solution. This sample solution includes the main points. A student's solution may expand on issues discussed here.

"A software development process transforms an initial system concept into an operational system running in the target environment. Its functions include identification of business needs, conducting feasibility study, formulating capabilities that the system must deliver as well as design, implementation, testing and deployment of the system to the target environment. The functions of software quality assurance (SQA) include definition of quality assurance standards and procedures, and verification, validation and testing activities to ensure that the development process is carried out properly, and the software artifacts produced by the development activities meet the software requirements and desired quality standards. Software project management oversees the control and administration of the development and SQA activities. Its functions include effort estimation, project planning and scheduling, risk management, and project administration. These activities ensure that the software system is delivered on time and within budget. During the development process, numerous software artifacts are produced including software requirements specification (SRS), software design, code, test cases, user's manual, etc. These are the software, or part of it, under different stages of the development process. These documents depend on each other. This implies that changes to one document may affect other documents, which may need changes as well. Software configuration management (SCM) is a mechanism to coordinate changes to ensure that changes are made consistently and cost-effectively.

All of software development process, SQA, project management and SCM contribute to

PQCT. In particular, good software development practices would apply well-established software development methodologies, software design principles, software design patterns, coding standards, test-driven development. These could lead to improvement of software productivity and software quality while at the same time reduce software costs and time to market. SQA ensures that the software meets the requirements and quality standards. It contributes to improvement of software quality. This in turn reduces rework and field-detected bugs; and hence, it also improves software productivity, reduces costs associated with rework and fixing field-detected bugs. Software project management ensures proper planning and administration of the software project. In particular, it should request the needed resources to develop the software system, properly schedule the development activities and SQA activities, manage budget and risks. These indirectly contribute to improvement of software productivity and software quality. Proper planning and administration of development and SQA activities directly contribute to reducing software development costs and time to market. This is because these activities could be performed smoothly, e.g., the needed components and resources are in place. SCM supports project management, SQA and software development process. It ensures that components of the software system are constructed and modified consistently and cost-effectively. Consistent modification implies productivity and quality, and cost-effectiveness implies reduction in cost and time to market.

A student's answer to this question may also include a discussion of the balance between PQCT. See Exercise 1.5."

1.3 Should optimization be a focus of software engineering? Briefly explain, and justify your answer with a practical example.

Solution. The answer to this question may depend on the interpretation of "optimization."

If it is about "optimization of software PQCT," then it is the focus of software engineering. If it is about performance optimization, then it should not be a focus, although SE also considers performance issues such as testing for performance. The database access example discussed in Section 1.5 is a practical example.

Optimization could be a focus for a given project. For example, the construction of a compiler for multicore computers. In this case, it depends on whether the project is classified as a software engineering, or a computer science project. It might be an SE project. For example, it is constructed for a certain application. (See solution to Exercise 1.6 for more on optimization and SE.)

1.4 Identify three computer science courses of your choice. Show the usefulness of these courses in the software life-cycle activities.

Solution. An Algorithms and Data Structures course is useful in the implementation phase for the design and implementation of algorithms and data structures to implement business processes. In particular, the course lets the student know the available algorithms and related data structures. The computational complexity lets the student select appropriate algorithms and data structures according to the nature of the computation.

A Database Systems course is useful in the analysis, design, and implementation phases. In the analysis phase, it helps the student understand database related requirements such as the need to support multiple databases for some applications. In the design phase, the course enables the student to design the database to fulfill the requirements and constraints. In the implementation phase, the course provides the student abilities to store and retrieve information with a database.

A Discrete Mathematical Structures course is useful in many phases of the life cycle. In

particular, mathematical logic lets the student design and implement logically consistent and logically complete algorithms, and check for such properties during design review and code review. Graph theory helps the student understand design diagrams such as UML diagrams because UML diagrams are directed graphs. Thus, concepts and algorithms of graph theory can be applied. Examples include fan-in and fan-out of a class, transitive closure for computing the change impact of a class, traversal algorithms for calculating reachability in a state diagram.

Courses on programming languages are useful for implementation, testing, and SQA activities. Network courses are helpful in SE project that must communicate with a remote computer, such as accessing a remote database. Artificial intelligence courses are useful for SE projects that involve heuristic, and/or learning algorithms.

1.5 There are interdependencies between software productivity, quality, cost, and time to market. For example, more time and effort spent in coding could increase productivity. This may result in less time and effort in software quality assurance because the total time and effort are constants. Poor quality could cost productivity due to rework. Identify three pairs of such interdependencies of your choice. Discuss their short-term and long-term impacts on the software development organization. How should software engineering solve the "dilemmas" induced by the interdependencies?

Solution. Barry Boehm in his papers on software engineering economics pointed out that the cost to fix a requirements error increases exponentially with time. That is, removing errors as early as possible is a cost-saving effort. This also coincides with the philosophy advocated by agile methods — that is, test early and often. Thus, if SQA is carried out as a life-cycle activity and follows a good SQA process, then it should detect requirements, design and

implementation errors early. This reduces bug fixing costs exponentially. Moreover, since SQA is a cooperate-wide practice, developers are conscious of developing quality software. This should significantly reduce the error rate; and hence, it reduces the cost that would otherwise have to be spent to fix the bugs.

In the short term, implementing and executing an SQA framework may reduce productivity, increase costs and time to market. However, in the long term, quality software brings many benefits to the organization. These include significant reduction in error rate and error correction costs, customer satisfaction, and increase in software capability maturity level. These should positively impact productivity, cost and time to market.

One should be aware that quality is not the more the better. To a certain point, there is the so-called "diminishing returns." Thus, how much SQA is appropriate remains a research problem in general and for a software organization in particular.

1.6 What are the differences and relationships between OO software engineering and conventional software engineering? Discuss whether OO software engineering will replace conventional software engineering.

Solution. The main difference between conventional software engineering and OO software engineering is paradigm shift — that is, how they view the world and systems. Because of this, they differ in the basic concepts, basic building blocks, and starting point for the conceptualization, design and implementation of software systems. These in turn affect SQA, project management (for example, effort estimation and planning), and SCM.

Will OO replace the conventional paradigm? The answer should be no[1]. The reasons are:

[1]A student's solution may indicate "yes," and provide convincing arguments. Such a solution should also be considered a good solution.

1. There are numerous systems that were developed using one or more of the conventional paradigms. It is very costly and risky to replace these systems. Therefore, the other paradigms will continue to exist because bug fixing, performance improvements, and functional enhancements to these systems are required.

2. There are hundreds of thousand organizations and millions of software developers using only the conventional paradigms. It is practically impossible and unjustifiable to require them to convert to the OO paradigm.

3. A conventional paradigm may be more suitable for some projects. For example, scientific computing typically involves series of transformations of input into output. Therefore, the function-oriented paradigm is more suitable for such applications. Moreover, scientific computing emphasizes computing speed, the ability to solve complex computation problems, and the accuracy of the result. OO programming languages may not satisfy such requirements. These and the facts that scientific computing is there to stay and expand into computational sciences imply that the function-oriented paradigm will continue to exist.

In addition to the above, one should know that different parts of a system may be developed using different paradigms. For example, a subsystem that performs scientific computing may be developed using the function-oriented paradigm. A database subsystem may be developed using the data-oriented paradigm. In practice, there are systems that are modeled and designed using the OO paradigm but implemented in a non-OO language. Similarly, there are projects that are modeled and designed using a conventional paradigm but implemented in SmallTalk or C++.

Chapter 2

Software Process and Methodology

2.1 What are the similarities and differences between the conventional waterfall model and the Unified Process model? Identify and explain three advantages and three disadvantages of each of these two models.

Solution. The waterfall model and the Unified Process (UP) model are similar in the sense that they are process models, they define phases, the activities and products of each of the phases. The waterfall process is a sequential process although backtracking is possible. The UP, on the other hand, is an iterative, incremental process.

Waterfall process advantages are: (1) it facilitates project management, scheduling and status tracking, (2) its can be used for function-oriented team organzation, and (3) it is more appropriate for some types of software project. Its disadvantages are: (1) it is difficult to respond to requirements change, (2) the long development duration is unacceptable, and (3) users cannot experiment with the system until late in the development life cycle.

UP advantages are: (1) its iterative process can better accommodate requirements change because changes can be made to remaining iterations, (2) it is use-case driven, allowing the development team to focus on customer value — that is, development and deployment of

high-priority use cases as early as possible, (3) it is incremental, this reduces the risk of requirements misconception. It disadvantages are: (1) an iterative process is more difficult to manage and schedule, (2) the early versions of the UP emphasize too much on documentation and much of it is not used, (3) the UP is a process, not a methodology, therefore, it is useful only for experienced software developers.

2.2 Write an essay about how a good process and a good methodology help tackling the project and product challenges. Limit the length of the essay to five pages, or according to the instructions of the instructor.

Solution. There could be many different answers to this exercise. It is difficult to come up with a standard solution and use it to grade the submissions. However, the answer should show how a good process and methodology address each of the challenges. Figure 2.1 of this manual highlights the main points and provides pointers to related chapters.

Grading of this exercise could be done by reading the solutions submitted by the students, according to the writing, the grader classifies the solutions into 3-5 categories such as very good, good, fair, below, and poor. Each of the categories is then reviewed and a score is assigned to each of the solution.

2.3 Write a brief essay on the differences between a software process and a software methodology.

Solution. Figure 2.11 of the textbook shows the differences between a process and a methodology. Therefore, the student needs only to explain the differences in the essay. Section 2.6.1 of the textbook presents the differences. A student's solution may reuse the materials in the section, and/or augment with practical examples, or experience.

2.4 Write an essay that discusses the following two issues:

	Description	Process or Methodology Solutions
Project Challenge 1	How do we plan, schedule and manage a project without sufficient knowledge about what will happen in the next several years?	• Effort estimation, and project planning and scheduling (Chapter 23) • Agile planning (Chapter 23) • Agile manifesto, principles, practices, and values (Chapter 2, and throughout the book) • Agile development, i.e., design for change, frequent delivery of small increments in short intervals (various chapters) • Risk management (Chapter 23)
Project Challenge 2	How do we divide the work among different departments and teams, and smoothly integrate the resulting components?	• System engineering (Chapter 2) • Software architectural design, behavioral design, and derivation of design class diagram (Chapters 6-16) • Peer review, inspection and walkthrough (Chapter 18) • Integration testing (Chapter 19)
Project Challenge 3	How do we ensure proper communication and coordination among the departments and teams?	• Modeling, analysis, and design using a unified modeling language such as UML (various chapters) • Applying design patterns during the design process (Chapters 11 and 16) • Software configuration management (Chapter 22)
Product Challenge 1	How do we develop the system to ensure that these requirements and constraints are met?	• Deriving use cases from requirements (Chapter 5) • Use case driven (various chapters) • Peer review, inspection and walkthrough (Chapter 18) • Acceptance testing and system testing (Chapter 19)
Product Challenge 2	How do we cope with changes?	• Design for change (various design chapters) • Applying design patterns to provide the needed flexibility (Chapters 11 and 16)
Product Challenge 3	How do we design the system so that changes can be made relatively easily and without much impact to the rest of the system?	Same as product challenge 2
Product Challenge 4	How do we design the system to hide the hardware, platform and implementation so that changes to these will not affect the rest of the system?	Same as product challenge 2

Figure 2.1: Dealing with project and product challenges

a. The pros and cons of plan-driven development and agile development processes, respectively.

b. Whether and why agile development will, or will not, replace plan-driven approaches.

Solution. The solution to 2.4a is similar to the solution about the differences between the waterfall and UP process models. The answer to 2.4b can be "yes" or "no," and the answer is not that important. The importance is the understanding of the differences between the two approaches, and the student's reasoning to justify the conclusion. This exercise should be graded using the method described in the solution for Exercise 2.2.

2.5 Write a short article that answers the following questions:

a. What are the similarities and differences between the spiral process, the Unified Process, and an agile process.

b. What are the pros and cons of each of these processes.

c. Which types of projects should apply which of these processes?

Solution. The similarities are that they are iterative processes, and meant to be an improvement over the existing processes. However, the iterations in the spiral process is situation dependent — that is, what to perform next depends on the outcome of the current iteration. Moreover, risk management is a unique feature of the spiral process. Unlike the spiral process, the UP repeats the same four phases in each iteration. It does not require the spiral process like decision making. It also does not indicate risk management. Agile processes are different from the spiral and UP in the agile manifesto, agile practices and values, and agile principles. In addition, agile development tend to adopt short iterations and frequent delivery of small increments. There are other differences but a solution should focus on these.

Among the three choices, projects that are research-oriented or exploratory may use the

spiral process. Projects that require adequate documentation should use the UP. Projects that need to respond quickly to changing business environments, and hence software requirements, should use an agile process. There are subtle differences between "research-oriented" and "changing requirements." Both need to tackle changing requirements. Research-oriented requirements need to be discovered with research tasks and experiments, which require considerable time and effort, and the costs are high.

2.6 Explain in an essay why the waterfall process is a process for solving tame problems.

Solution. The waterfall process requires that the requirements of the system must be identified, clearly and completely defined before the design and implementation of the system. This is at least true in theory, although many real-world projects do not happen like this. The first two properties of wicked problems are: (1) a wicked problem does not have a definite formulation, and (2) the specification of the problem and the solution cannot be separated. Clearly, the waterfall process cannot solve wicked problems because the problem-solving process does not address these two wicked-problem properties. A student's solution may address other properties as well. (See also solution to Exercise 2.7, especially Figure 2.2 of this manual. From the discussion and the Figure 2.2, one may infer more on the inadequacy of waterfall in solving software development as a wicked problem.)

2.7 Explain in an essay how agile development tackles application software development as a wicked problem.

Solution. Software development as a wicked problem implies that the requirements for a software system cannot be completely and definitely formulated, and the specification and the solution cannot be separated — the specification is the solution, and vice versa. Agile development recognizes these and advocates responding to requirements change. The 20/80

Properties of a Wicked Problem	Agile Development Solution
1) A wicked problem does not have a definite formulation.	• Value working software over comprehensive documentation. • Value responding to change over following a plan. • Capture requirements at a high level, lightweight, and visual. • User involvement is imperative. • Good enough is enough.
2) For a wicked problem, the specification is the solution and vice versa.	
3) There is no stopping rule for a wicked problem --- you can always do it better.	• Requirements evolve but the timescale is fixed. • Don't work over 40 hours a week.
4) Solutions to wicked problems can only be evaluated in terms of good or bad, and the judgment is subjective.	• User involvement is imperative. • The team is empowered to make decisions. • Users perform testing.
5) Each step of the problem-solving process has an infinite number of choices ---everything goes as a matter of principle.	• Value individual and interaction over processes and tools. • The team is empowered to make decisions.
6) Cause-effect reason is premise-based, leading to varying actions, but hard to tell which one is the best.	• User involvement, value individual and interaction, team decision making. • A collaborative and cooperative approach between all stakeholders is essential. • Value customer collaboration over contract negotiation.
7) The solution cannot be tested immediately and is subject to life-long testing.	
8) Every wicked problem is unique.	
9) The solution process is a political process.	
10) The problem-solver has no right to be wrong because the consequence is disastrous.	

Figure 2.2: Wicked problems and agile development as a solution

rule indicates that it is good enough to identify 80% of the requirements that are of high customer value. In additiion, it advocates capturing requirements at a high level, lightweight, and visual. That is, low-level requirements are to be captured during the implementation phase. This is because the specification and the solution cannot be separated. Agile development also emphasizes on user involvement because the "correctness" of a software system cannot be determined objectively and scientifically. Figure 2.2 of this manual shows how agile manifesto and principles solve wicked problems.

Chapter 3

System Engineering

3.1 Provide a brief description of the functions of a vending machine. Identify and formulate all functional and performance requirements.

Solution. A vending machine (VM) allows an operator to program the bill acceptor, the coin acceptor, and vending prices. A customer can vend items with the VM, receive the vend item and the change. The VM shall keep track of the all-time total amounts and since-reset total amounts of bills and coins received, respectively. An operator can reset only the since-reset total amount. The operator can also view the all-time total amount and the since-reset total amount.

The functional requirements of the VM include the following:

Software Functional Requirements

R1 The VM shall allow an operator to perform the following VM operator functions:

R1.1 An operator shall be able to program the bill acceptor for bills accepted ($1, $5, $10, $20).

R1.2 An operator shall be able to program the coin acceptor for coins accepted (nickel, dime,

quarter, and dollar coin).

R1.3 An operator shall be able to view the all-time total amount of bills and coins received.

R1.4 An operator shall be able to view the since-reset total amount of bills and coins received.

R1.5 An operator shall be able to reset either or both of the since-reset total amount of bills and since-reset total amount of coins received, respectively.

R1.6 An operator shall not be allowed to reset the all-time total amount of bills and coins received.

R1.7 An operator shall be able to program the vending prices for the vending items using the alphanumeric keypad mounted behind the front panel.

R2 The VM shall allow a customer to vend items.

R2.1 A customer shall be able to vend items with bills.

R2.2 A customer shall be able to vend items with coins.

R2.3 A customer shall be able to vend items with a combination of bills and coins.

R2.4 The VM shall display the total amount of bills and coins inserted by the customer for the current transaction.

R2.5 A customer shall be able to press a letter key and a digit key on the keypad mounted on the front panel to vend an item when the total amount of bills and coins inserted is sufficient.

R2.6 The VM shall dispense the change and the item selected by the customer if the inserted amount is sufficient and the selected item is available, else the VM shall display an error message.

R2.7 A customer shall be able to press the return key to cancel an outstanding transaction. The VM shall return the funds inserted by the customer.

R2.8 ...

Hardware Functional Requirements

R3 The VM shall include all hardware components to support the VM functions:

R3.1 The VM shall include a programmable bill accepter, and a programmable coin acceptor.

R3.2 The VM shall include an operator keypad for the operator functions. The keypad shall be mounted on the back of the front panel and accessible only when the front panel is unlocked.

R3.3 The VM shall include a customer vending keypad, mounted on the front of the front panel.

R4 ... (Other hardware functional requirements can be formulated similarly and are omitted.)

Nonfunctional Requirements

Nonfunctional requirements for the VM include performance, response time, user interface (UI), and other nonfunctional requirements. For example, the VM shall display the total inserted amount, and dispense items within X and Y seconds, respectively.

3.2 Identify two embedded systems not mentioned in this chapter. For each of these systems, perform the following:

a. Briefly describe the functions of the system. The description should be about a half of a page to one page including diagrams, if any.

b. Identify and formulate five functional requirements and two nonfunctional requirements. One of the nonfunctional requirements must be a performance requirement.

c. Decompose the system and allocate the system requirements to the subsystems.

Solution. The solution to this exercise depends on the systems chosen by the student. Grading could be done by ranking the solutions as A-C or A-E and then score the solutions in each rank. For example, A=90-100, B=80-89, C=70-79, D=60-69, and E=0-59. The best solution in the A ranking could score 100 or close to 100 and the worst solution in the A ranking could score 90 or close to 90. The other categories can be treated similarly.

3.3 A coin-operated car wash system is a self-service car wash system. A customer inserts the required number of quarters to buy a preset period of wash time. The customer can turn a dial to select soap, foam, rinse, and wax any time during the wash period. The system beeps when one minute is remaining. The customer can insert more quarters to buy more wash time. Perform the following for the car wash system:

a. Identify and specify the system requirements including functional and nonfunctional requirements.

b. Decompose the system into functional subsystems. Decompose the system requirements if necessary.

c. Allocate the system requirements to the subsystems.

d. Construct a system architectural design diagram using a diagramming technique of your choice or as designated by the instructor.

Solution. (Omitted.)

3.4 A railroad crossing system employs sensors, flashing lights, sounding bells, and gates to control the traffic at a railroad intersection. When a train approaches the intersection from either direction, a sensor device senses the train and communicates with the software. The software turns on the flashing yellow warning light and the sounding bell for a given period. It then changes the light to flashing red and closes the gates. After the train has left the intersection,

another sensor device detects this and communicates the event to the software. The software turns off the lights and the sounding bell and opens the gates. Perform the following for the railroad crossing system:

a. Identify the hardware and software subsystems and specify the functionality of each of the subsystems.

b. Describe how the subsystems relate to and interact with each other. That is, specify the subsystem interfaces and interaction behavior.

c. Construct a system architectural design diagram to show the relationships between the subsystems.

d. Identify and formulate safety requirements and allocate them to the subsystems. Describe how the subsystems will satisfy the safety requirements.

Solution. Figure 3.1 of this manual shows the major subsystems and their relationships. The arrival and departure detectors sense the arrival and departure of a train and send the sensory information to the central control, which is a software subsystem. The central control issues instructions to control the bell, the light and the gate. The interaction behavior is as described in the exercise. The backup power subsystem is a battery-based, rechargeable power backup device to supply power to the other subsystems during a temporary power outage. It notifies the central control when its backup battery power is low. In this case, the central control instructs the gate to close the intersection for safety purpose.

The safety requirements for the railroad crossing system may include the following, although other safety requirements may be formulated. Figure 3.2 of this manual shows how the subsystems satisfy these safety requirements.

R1 The system shall equip with a backup power system to allow the entire system to operate

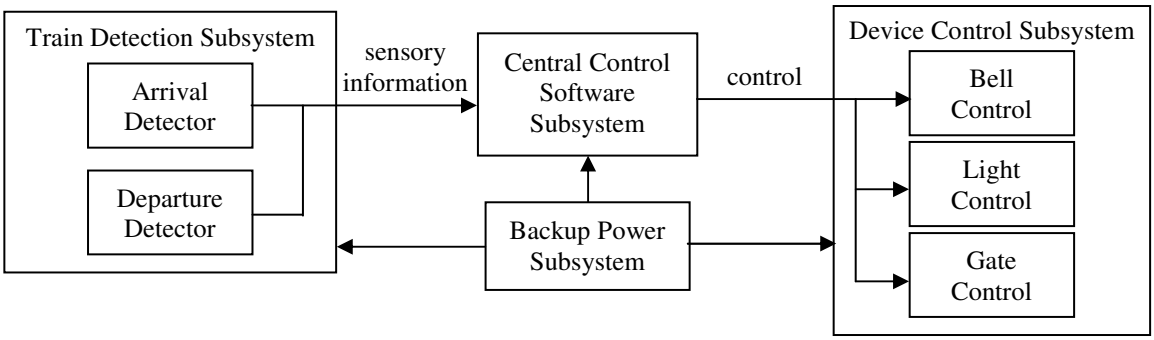


Figure 3.1: System architecture for railroad crossing

	Train Detection	Central Control	Backup Power	Device Control
R1 - equip with a backup power system			√	
R2 - notify system operator when power outage		√	√	
R3 - close the gates before backup power outage		√	√	√
R4 - self-diagnosing and notifying of defects	√	√	√	√

Figure 3.2: Allocating safety requirements to subsystems

as usual during temporary power outages.

R2 The system shall immediately notify the system operator through phone call, text messaging, and email whenever a power outage occurs.

R3 The system shall close the gates before the backup power system runs out of power.

R4 The system shall be self-diagnosing and notify immediately the system operator through phone call, text messaging, and email in case any sensing or external control device is out of order.

3.5 Managers of a department store want to expand into online retailing. This means that the company needs to develop an online system that can take orders online, and ship the ordered items through a designated national shipment carrier. To reduce labor costs, the company wants a fully automated system. Perform the following for this system:

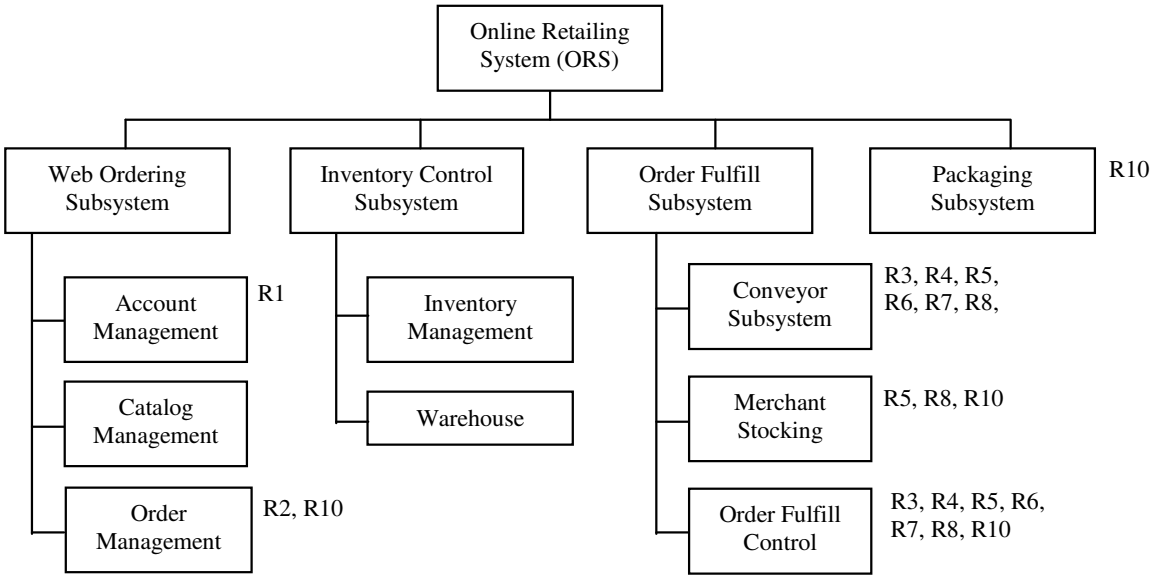


Figure 3.3: Online ordering system decomposition and allocation

a. Identify and formulate five functional requirements and two performance requirements for the system.

b. Decompose the system into a hierarchy of subsystems and allocate the system requirements to the subsystems.

c. Produce a system architectural design diagram.

Solution. The functional and performance requirements for the online retailing system (ORS) include the following:

R1 ORS shall allow a potential customer to create an online account.

R2 ORS shall allow customers to order online.

R2.1 A customer shall be able to order multiple items in one online transaction.

R2.2 A customer shall be able to make payment using one of several payment methods including credit card, PayPal, personal check, and bank account.

R2.3 A customer shall be allowed to specify a shipping address different from the billing

address.

R2.4 ORS shall print the shipping label and place it into the designated conveyor cart.

R3 ORS shall dispense the required packaging material such as a carbon box into the designated conveyor cart.

R3.1 ORS shall calculate the volume of the items ordered.

R3.2 ORS shall determine the appropriate packaging material required to ship the ordered items.

R4 ORS shall direct the conveyor cart to traverse the conveyor network to load the ordered items.

R5 ORS shall dispense the ordered items when the conveyor cart traverses by the ordered items.

R6 ORS shall direct the conveyor cart to the packaging area when all of the ordered items are loaded.

R7 ORS shall ensure efficient utilization of the conveyor subsystem with different sizes of conveyor carts to accommodate different volumes of orders. (A nonfunctional requirement.)

R8 ORS shall ensure that the dispensers have enough time to correctly dispense the ordered items into the designated conveyor carts. (A nonfunctional requirement.)

R9 The conveyor belt shall traverse at a speed of X feet per second. (A performance requirement.)

R10 ORS shall be able to process Y transactions per minute. (A performance requirement.)

A decomposition of the ORS into subsystems and the allocation of the requirements to the subsystems are shown in Figure 3.3 of this manual. Note that some subsystems do not

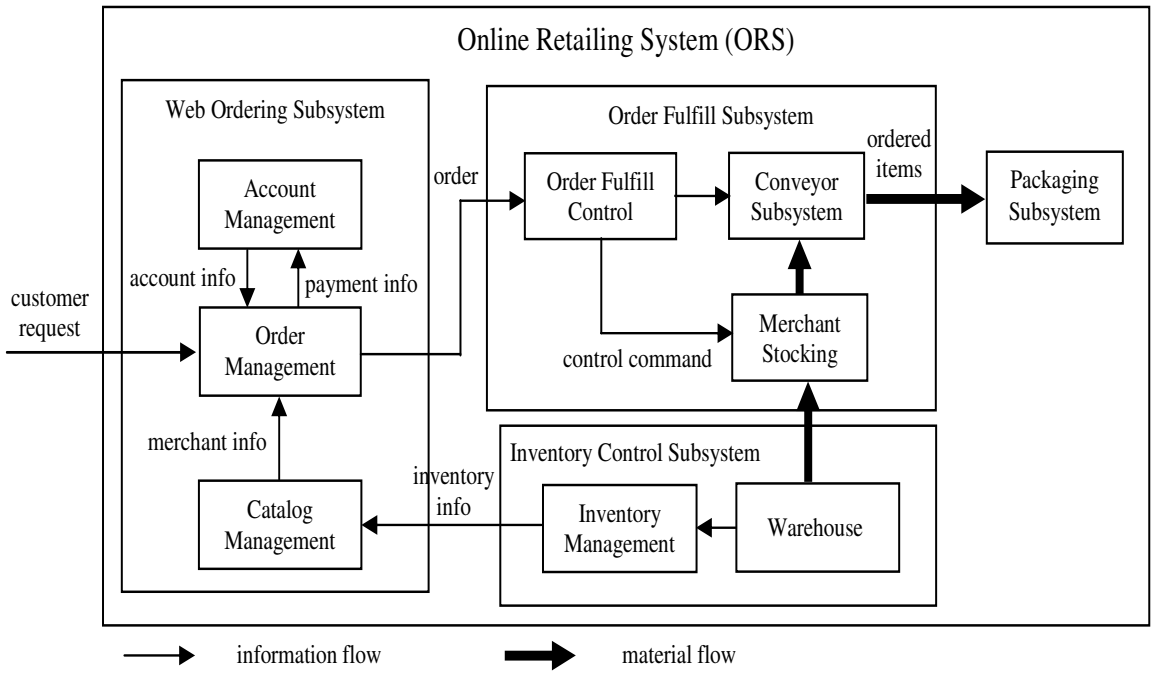


Figure 3.4:

have allocated requirements. This is because the list of requirements is incomplete. An architectural design is given in Figure 3.4 of this manual.

3.6 If you have learned UML class diagramming, then produce an object-oriented context diagram for the software subsystem of the online retailing system you produced in exercise 3.5. Discuss the usefulness of the context diagram for the design, implementation, testing, and maintenance of the software subsystem.

Solution. (Omitted.)

Part II

ANALYSIS AND ARCHITECTURAL DESIGN

Chapter 4

Software Requirements Elicitation

4.1 Produce a software requirements specification (SRS) for a library information system that is similar to the system in use in your school. At the minimum, the system should provide functions to allow the patron to search, check out, and return documents, respectively.

Solution. This exercise is quite common. There are quite a few solutions published on the web. Moreover, the library information systems are different for different universities. Therefore, a solution is not provided.

4.2 Formulate the functional and nonfunctional requirements for each of the following systems. For each system, limit the requirements specification to no more than three pages.

a. A desktop virtual calculator that allows the user to use the mouse as well as the keyboard to enter the input

b. A telephone answering machine

c. A web-based email system

Solution. These devices are common. The student's solution should include the commonly seen functions of these devices. A desktop computer usually comes with a calculator. The

help topics of the calculator provide descriptions of functions of the calculator. Similarly, the help pages of a web-based email system provide descriptions of the functions the email system.

4.3 Identify and specify the functional and nonfunctional requirements for a web-based course management system. The system should allow students to search, register, and drop courses, respectively. It also allows the staff members of the study administration of the university to schedule classes.

Solution. The functional and nonfunctional requirements of the online course management system (OCMS) include the following:

R1 OCMS shall provide web-based access to OCMS users including staff members of the study administration, system administrators (which are staff members of the study administration), students, advisors of academic colleges and departments, as well as faculty members of the university.

R1.1 OCMS shall allow users to login and logout OCMS within and outside of the campus.

R1.2 OCMS shall provide role-based access control (RBAC) so different categories of users can access to different resources including OCMS functions and information.

R1.2.1 OCMS shall allow a system administrator to define roles and resources, and which roles can have which access privilleges to which resources.

R1.2.2 OCMS shall implement the RBAC as described in R1.2.1.

R2 OCMS shall allow staff users and academic advisors to schedule, modify, and delete courses according to RBAC defined in R1.2.1:

R2.1 Staff users and academic advisors shall be able to schedule courses by specifying the course number, course title, lecture dates and times, room, instructor, and capacity

(enrollment limit).

R2.2 OCMS shall allow staff members and academic advisors to modify scheduled courses.

R2.3 OCMS shall allow staff members to delete scheduled courses.

R3 OCMS shall allow students to enroll in courses if the student has passed the prerequisite courses and the course to enroll has available seats.

R4 OCMS shall update the number of enrollments and the available seats whenever a student successfully enrolls in a course.

R5 OCMS shall allow all users to search for courses using a variety of search criteria.

R6 OCMS shall allow all users to browse the scheduled courses.

R7 OCMS shall allow users to access OCMS functions using popular web browsers including Google Chrome, Mozilla FireFox, and Microsoft Internet Explorer.

R8 OCMS shall allow users to access OCMS functions using popular mobile devices such as cell phones, and Windows Mobile devices.

4.4 Formulate the functional requirements for the calendar management system described below. Limit the length of the SRS to no more than two pages. This calendar management software allows the user to schedule personal activities such as meetings and tasks to be performed. An activity can take place on a future date during a certain period of time. An activity can take place for several consecutive days. Each activity has a brief mnemonic description. An activity can be a recursive activity, which takes place repeatedly every hour, every day, every week, or every month. A user can schedule an activity using a month-by-month calendar to select the date or dates, and then zooms in to select the begin time and end time on a date. The calendar system shall notify the user by email, text message, or phone call the day before and on the activity day. The user can review past activities and modify the schedule

including updating and deleting activities.

Solution. Online calendar management systems are common. The student's solution should reflect the commonly used functions of such systems.

4.5 Produce an SRS for the room reservation system described below. Limit the length to no more than two single-space pages. This web-based single-room reservation system is a resource management system that allows registered users of an organization to reserve a room for a given period of time on a given date. Moreover, the system allows the user to choose whether a reminder message will be sent automatically to a group of users and on which date(s) to send the reminder message. Since only one room is available, therefore, no search capabilities are necessary. The system allows a user to navigate to the desired day using a visual monthby-month calendar. As the monthly calendars are displayed, the time periods that are already reserved are colored red and the available time periods are colored differently, or shown with no color. The configuration of the system is used to specify a number of properties or constraints:

a. Enable or disable a display of which user reserves which time period.

b. Enable or disable the ability for a user to reserve the room recursively for a given period of time. For example, 10:00 a.m.12:00 p.m. every Wednesday for the whole semester.

c. Specify a constraint on the maximal period of time for recursive reservations.

d. Specify a constraint on the number of reservations that a user can make.

e. Enable or disable automatic system reminder message.

Solution. This exercise has described most of the functionality of the room reservation system. The student's solution should formulate the functional requirements accordingly. The student's solution may include other useful functions.

4.6 In Appendix D.2, a National Trade Show Services (NTSS) business is described. Suppose that currently the NTSS business is conducted manually. The company wants to develop a web-based online system to automate the business. This exercise may be an individual assignment or part of a team project. It requires the student or team to do the following:

a. From the business description, identify and list the most important business activities of the current business. These are the functions that the future system must provide. Hint: The number of such activities may differ from student to student. However, the combined functionality identified should cover at least 90% of the current business activities, which includes the most important business functions.

b. Identify other functions for the online system so the business can run online. For example, the system needs to authenticate users.

c. Formulate functional requirements.

Solution. a. The most important business activities of the NTSS are:

1. Create an online account
2. Submit an event proposal
3. Cancel an event
4. Edit a pending event proposal
5. Evaluate an event proposal
6. Submit a booth lease request
7. Evaluate booth lease requests
8. Print payment summary
9. Withdraw a booth lease request
10. Register for an event

11. Cancel a registration

b. Other functions include:

12. Logon

13. Logoff

c. The software requirements for the NTSS include:

R1 The system shall allow a potential user to create an online account identified by a valid email address.

R1.1 The system shall require the user to specify the account type (event organizer, exhibitor, speaker, or observer). Different types of account shall have different privileges.

R1.2 The system shall send a temporary password to the email address. The temporary password must comply to commonly adopted password security rules.

R1.3 The system shall require the user to change the temporary password at the first login. The new password must comply to commonly adopted password security rules.

R2 The system shall allow an event organizer to submit an event proposal using an event proposal form with fields that describe the event.

R2.1 Upon submission, the system shall generate a payment slip to be sent by the event organizer along with the payment to NTSS.

R2.2 Upon reception of the payment for an event proposal,

R2.2.1 the system shall change the status of the event proposal to "Pending Review" from "Submitted,"

R2.2.2 the system shall email notify the event organizer that the payment has been received and the proposal is pending review.

R3 The system shall allow an event organizer to cancel an event with a 15% cancellation charge.

R4 The system shall allow an event organizer to edit a "Submitted" or "Pending Review" proposal without charge.

R5 The system shall allow an event organizer or NTSS staff to view an event proposal.

R6 The system shall allow an NTSS staff to evaluate an event proposal and enter feedback and an acceptance decision of either "Accepted" or "Rejected." If "Rejected," then the staff must also provide feedback explaining why the proposal is rejected.

R7 The system shall allow an exhibitor to submit a booth lease request.

R7.1 The system shall display a floor plan with available booths, their sizes and leasing prices. The booth sizes are large, medium, and small, each of which is specified by width and length.

R7.2 The system shall allow the exhibitor to select one or more available booths to be leased.

R7.3 The system shall generate a summary and total payment slip so that the exhibitor can print and send along with the payment to NTSS.

R7.4 When the payment is received, the system shall change the status of the request to pending review and notify the exhibitor of the status change.

R8 The system shall allow an exhibitor to withdraw a booth lease request with a 10% cancellation charge.

R9 The system shall allow an exhibitor or NTSS staff to view a booth lease request.

R10 The system shall allow an NTSS staff to evaluate a booth lease request.

R10.1 The system shall allow an NTSS staff to enter "Accepted" or "Rejected" decision.

R10.2 If the decision is Rejected, then the system shall require the NTSS staff to enter feedback explaining why the lease request was rejected.

R10.3 The system shall change the status of the request accordingly and notify the exhibitor of the change in request status.

R10.4 The system shall change the status of a booth to "unavailable" once it is leased out.

R11 The system shall allow a potential participant to register for an event.

R11.1 The system shall allow the participant to select from a list of event of her/his interest.

R11.2 The system shall allow a participant to register for one or more events.

R11.3 The system shall generate a payment slip showing a summary and the total payment.

R11.4 The system shall allow the participant to print the payment slip to be sent together with the payment.

R11.5 Once the payment is received, the system shall change the participants' registrations to "Paid" and notify the registering participant of this change.

R12 The system shall allow a registering participant to cancel a registration without charge one week before the event and with a 10% cancellation charge afterwards.

R13 The system shall allow an NTSS staff to specify a floor plan with predefined sizes.

R14 The system shall allow payment by credit card, cashier check, personal check, money order, or PayPal.

4.7 Describe in concrete terms how you would conduct each of the three types of review for the NTSS software requirements specification you produce in Exercise 4.6. Limit your answer to no more than one single-spaced page and no less than 11 point font size.

Solution. The solution to this exercise is a straightforward application of the three types of review to the NTSS requirements. The three types of review are: internal technical review,

domain expert review, and customer review. The focus of each type of review has been described in Chapter 4 and can be applied directly to the requirements specification presented above.

4.8 Identify and formulate requirements and constraints for the online car rental system described in Appendix D.1. Also provide priority weights for the requirements according to the nature of the car rental business.

Solution. Some requirements are:

R1 The car rental CRS (CRS) shall provide a secure means for customers to create an online account with the car rental business.

R2 The CRS shall provide a secure means for customers and employees to search rental vehicles.

R3 The CRS shall provide a secure means for customers to reserve vehicles online.

R3.1 The CRS shall allow customers to select vehicle make, model, and available options.

R3.2 The CRS shall suggest similar vehicles if the selected vehicle is not available.

R3.3 The CRS shall allow the user to select a rental plan including the Daily Unlimited Miles Plan and Weekend 10% Discount Plan.

R3.4 The CRS shall display the rental prices for vehicles with selected options.

R3.5 The CRS shall allow customers to make block reservations of more than one vehicle.

R4 The CRS shall allow customers to cancel reservations online.

R5 The CRS shall allow an employ to make reservations for a customer.

R6 The CRS shall void a reservation if the customer does not sign the rental contract for a reservation within a predefined amount of time.

R7 The CRS shall allow The CRS shall allow a customer to view reservations made by/for the customer.

R8 The CRS shall allow an employee to search reservations using a number of search criteria.

R9 The CRS shall allow an employee to cancel a reservation for a customer.

R10 The CRS shall open an invoice when a vehicle is checked out to a customer.

R11 The CRS shall generate invoices for rental contracts when vehicles are returned.

R12 The CRS shall maintain purchase, maintenance, repair, and disposal logs for each vehicle.

R12.1 The CRS shall allow an employee to update the logs.

R12.2 The CRS shall allow employees to view the logs.

R13 ...

Constraints for the CRS include:

C1 The CRS must provide a means to ensure that every rental car that is checked in to the system meets federal and state safety and environmental regulations.

C2 The CRS must not allow vehicles that do not meet federal or state safety regulations to be displayed in search results or checked out.

C3 The CRS must provide a means to validate that the customer has a valid driver license.

C4 The CRS must provide a means to validate that the customer possess minimal liability insurance (which the customer may already have with her/his existing cars or can be purchased at time of check out).

C5 The CRS must secure customers privacy and financial information when transmitting such information over the Internet.

4.9 Write an article to describe how requirements elicitation would be performed using the waterfall process and an agile process, respectively. List the differences between the two approaches, and provide a brief explanation of the differences.

Solution. The main points are discussed here. The student's solution may be more thorough and provide more detail.

Requirements elicitation using the waterfall process is aimed at identifying as complete and correct requirements. This tend to result in heavy documentation. Agile processes advocate capturing requirements at a high level, lightweight, and visual. Furthermore, agile processes suggest "good enough is enough." That is, requirements elicitation aims at capturing 80% of the requirements, which are of high customer value. Waterfall and agile also differ in the way toward requirements change. Requirements change is difficult for a waterfall project. On the other hand, agile processes respond to change.

Chapter 5

Domain Modeling

5.1 A directed graph or digraph G = (V, E, L) consists of a set of vertexes V = v1, v2, . . . , vn, a set of edge labels L, and a set of directed edges E $\subseteq$ V x V x L. Suppose you are assigned to design and implement a graph editor, that is, an editor for drawing and editing a digraph. You need to construct a domain model. This exercise requires you to perform the following:

a. Produce a textual description for a graph. The description must describe the elements of a graph such as vertexes and edges and relationships between the elements.

b. Apply the brainstorming rules to the textual description as described in Section 5.4.2 and produce the brainstorming result.

c. Classify the brainstorming result as described in Section 5.4.3 and produce the classification result.

d. Convert the classification result into a UML class diagram.

Solution. Two textual descriptions for a directed graph with domain concepts identified are shown in Figure 5.1 of this manual. The classification results and domain model class diagrams are shown in Figure 5.2 and Figure 5.3 of this manual, respectively. Note: edge

Description 1	Description 2
A directed graph[1] or digraph consists of[7] a[5] set of[2] zero or more[5] vertexes, a[5] set of[2] zero or more[5] directed edges, and a[5] set of[2] zero or more[5] edge labels. Each vertex[1] has[6] a name[1], which is a string[1]. Each directed edge[1] connects[3] one[5] vertex, called the source[1], to one[5] other vertex, called the destination[1]. Each directed edge has[6] a label[1]. The source vertex and the destination vertex may be the same.	A directed graph[1] or digraph consists of[7] zero or more[5] vertexes, directed edges, and edge labels. Each vertex[1] has[6] a name[1], which is a string[1]. Each directed edge[1] connects[3] one[5] vertex, called the source[1], to one[5] other vertex, called the destination[1]. Each directed edge has[6] a label[1].

Figure 5.1: Description and brainstorming for a digraph

Brainstorming Result	Classified to Modeling Concepts
directed graph[1] consists of[7] a[5] set of[2] zero or more[5] vertexes, a[5] set of[2] zero or more[5] directed edges, and a[5] set of[2] zero or more[5] edge labels.	(C) Directed Graph (AG) Part-of (Vertex Set, Edge Set, Label Set, Directed Graph) (C) Vertex Set (C) Edge Set (C) Label Set (AS) contain (Vertex Set, Vertex) (1, *) (AS) contain (Edge Set,Directed Edge) (1, *) (AS) contain (Label Set, Label) (1, *)
Each vertex[1] has[6] a name[1], which is a string[1].	(C) Vertex (A) name: String
Each directed edge[1] has[6] a label[1].	(C) Directed Edge (AS) has (Directed Edge, Label) (C) Label
Each directed edge connects[3] one[5] vertex called the source[1], to one[5] other vertex called the destination[1]	(AS) connect-from (Directed Edge, Vertex) (NONE, source) (AS) connect-to (Directed Edge, Vertex) (NONE, destination)

Classification solution 1

Brainstorming Result	Classified to Modeling Concepts
directed graph[1] consists of[7] zero or more[5] vertexes, zero or more[5] directed edges, and zero or more[5] edge labels.	(C) Directed Graph (AG) Part-of (Vertex, Directed Edge, Label, Directed Graph) (*, *, *, 1)
Each vertex[1] has[6] a name[1], which is a string[1]	(C) Vertex (A) name: String
Each directed edge[1] has[6] a label[1].	(C) Directed Edge (AS) has (Directed Edge, Label) (C) Label
Each directed edge connects[3] one[5] vertex called the source[1] to one[5] other vertex called the destination[1]	(AS) connect-from (Directed Edge, Vertex) (NONE, source) (AS) connect-to (Directed Edge, Vertex) (NONE, destination)

Classification solution 2

Figure 5.2: Classifying brainstorming result for a digraph

labels could be classified as attributes of labels. If this is adopted, then a digraph has only a vertex set and an edge set, and a label is an attribute of an edge.

5.2 Produce a one-page description of the business operation for a hotel reservation system. State practical and reasonable assumptions.

Solution. The description of a hotel reservation system is the following:

"The hotel reservation system (HRS) is intended to be an online reservation system. Authorized hotel employees shall be able to enter and update hotel and room information online,

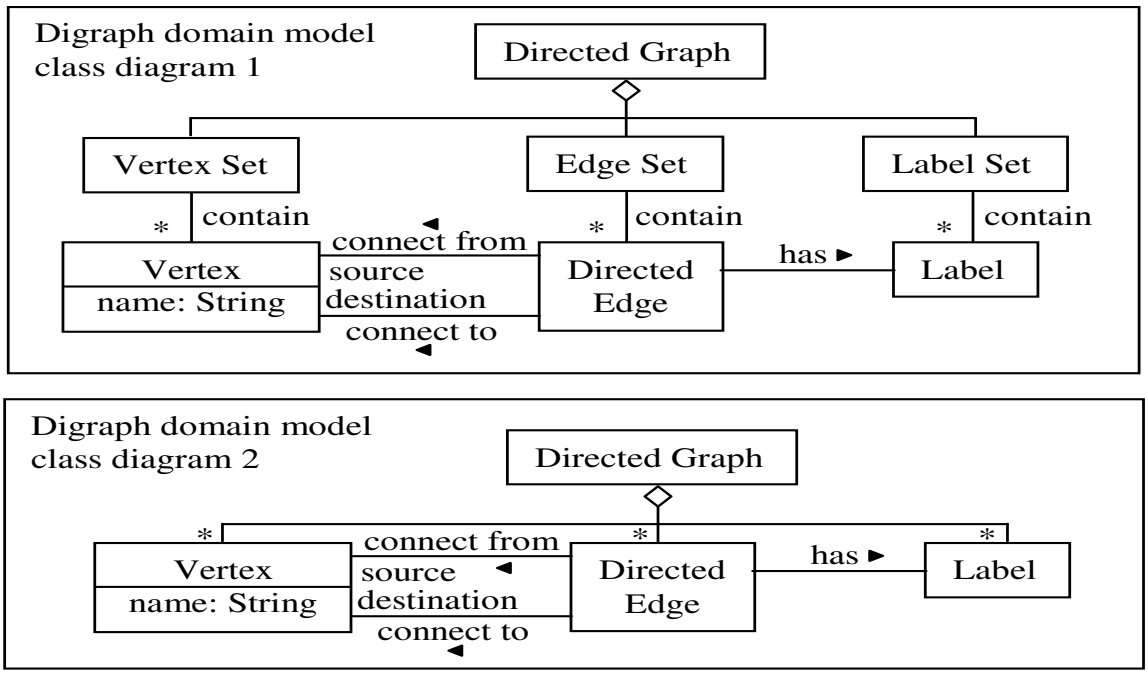


Figure 5.3: Digraph domain model class diagram

including special offers. Potential guests shall be able to search rooms using a number of search criteria. The HRS shall respond with a list of summary descriptions of the rooms that satisfy the search criteria. The guest shall be able to view the detailed description and photographs of any of the rooms. The guest shall be able to select several rooms to view a comparison of their features.

The description of the hotel includes the address, phone number, fax number, email address, facilities such as cafeteria, bar, conference rooms, swimming pool, recreational facilities, directions to the hotel and a map, among others. The description of a room shall include the room number, room size, bath room (shower or bath tub), television, Internet connection, number of beds and bed size, and availability, among others. The charges of a room may vary according to the travel season and may be adjusted manually as well as automatically (e.g., automatically increase by 5% each year, or according to changes of price index).

A guest shall be able to reserve rooms with reservations. A reservation may reserve one or more rooms for a period specified by the guest, provided that the room is not reserved during that period of time. A confirmation message shall be sent to the guest's email account. The rooms are reserved only to 6:00 pm for the guest on the first day of the reservation unless the reservation is guaranteed by a nonrefundable first night charge to a credit card provided by the guest. That is, the amount of one night charge for each room of the reservation is charged to the credit card after 6:00 pm of the first day of the reservation. When a reservation is made, the room becomes unavailable for the period of time specified by the reservation. A guest is allowed to update or delete a reservation, provided that the reservation is still outstanding. Other services may be added to this project. For example, confirmation and notification services can be selected by the customer, who may specify the means (such as email, text message, or phone call) to deliver the services."

5.3 Perform the brainstorming step for the hotel reservation system using the description produced in exercise 5.2.

Solution. Figure 5.4 of this manual shows the brainstorming results.

5.4 Perform the classification step from the brainstorming results produced in exercise 5.3 for the hotel reservation system.

Solution. The classification is shown in Figure 5.5 of this manual.

5.5 Draw a UML class diagram as a domain model based on the classification results produced in exercise 5.4.

Solution. The conversion of the classification result to a UML class diagram is straightforward and is omitted here.

The hotel reservation system (HRS) is intended to be an online reservation system. Authorized[4] hotel employees[1] shall be able to enter[3] and update[3] hotel[1] and room[1] information[1] online, including special offers[1]. Potential guests[1] shall be able to search[3] rooms using a number of[5] search criteria[1]. The HRS shall respond[3] with a list of summary descriptions of the rooms[2] that satisfy the search criteria. The guest shall be able to view[3] the detailed description[1] and photographs[1] of any[5] of the rooms. The guest shall be able to select[3] several[5] rooms to view a comparison of their features. The description of the hotel[2] includes[6] the address[1], phone number[1], fax number[1], email address[1], facilities[1] such as[6] cafeteria[1], bar[1], conference rooms[1], swimming pool[1], recreational facilities[1], directions[1] to the hotel and a map[1], among others. The description of a room shall include[6] the room number[1], room size[1], bath room[1] (shower[1] or bath tub[1]), television[1], Internet connection[1], number of beds[1] and bed size[1], and availability[1], among others. The charges of a room[2] may vary according to the travel season[1] and may be adjusted manually[4] as well as automatically[4] (e.g., automatically increase by 5%[5] each year, or according to changes of price index[1]).
A guest shall be able to reserve[3] rooms with reservations[1]. A reservation may reserve one or more[5] rooms for a period[1] specified by[3] the guest, provided that the room is not reserved during that period of time. A confirmation message[1] shall be sent to[3] the guest's email account[2]. The rooms are reserved[4] only to 6:00 pm[5] for the guest on the first day of the reservation[2] unless the reservation is guaranteed[3] by a nonrefundable[4] first night charge[1] to a credit card[1] provided by[3] the guest. That is, the amount of one night charge[2] for each room of the reservation[2] is charged to[3] the credit card after 6:00 pm of the first day of the reservation.
When a reservation is made, the room becomes unavailable for the period of time specified by the reservation. A guest is allowed to[3] update[3] or delete[3] a reservation, provided that the reservation is still outstanding[4]. Other services[1] may be added to[3] this project. For example, confirmation[1] and notification[1] services can be selected by[3] the customer[1], who may specify[3] the means[1] (such as[6] email[1], text message[1], or phone call[1], to deliver[3] the services.

Figure 5.4: Brainstorming hotel reservation system

5.6 Review the domain model class diagram produced in exercise 5.5 using the review checklist in Section 5.4.5. Produce a review report that answers the list of review questions.

Solution. The student needs to review and answer the review questions. The answers are graded according the domain model produced by the student. The student may improve the model after reviewing it. The instructor may arrange these as deemed appropriate.

5.7 Select another application and do as the above exercises for this application.

Solution. The solution to this exercise depends on the student's choice; and hence, no solution can be provided.

5.8 Suppose you are assigned to develop a graphical editor for a UML class diagram. This exercise requires you to perform the domain modeling steps and produce a domain model that describes the modeling notions depicted in Figure 5.1 of the textbook. Hint: This exercise is similar to exercise 5.1 but is a little bit more complex. The resulting domain model is a UML

Brainstorming Result	Classification	Rule
hotel[1]	(C) Hotel	1(a)
description of the hotel[2]	(AG) Part-of (Hotel Description, Hotel) (C) Hotel Description	2(a)
address[1]	(A) address	1(e)
phone number[1]	(A) phone number	1(e)
fax number[1]	(A) fax number	1(e)
email address[1]	(A) email address	1(e)
room[1]	(C) Room	1(a)
descriptions of the rooms[2]	(AG) Part-of (Room Description, Room) (C) Room Description	2(a)
room number[1]	(A) room number	1(e)
room size[1]	(A) room size	1(e)
bath room[1]	(A) bathroom style	1(e)
shower[1]	(V) shower (of bathroom style)	1(e)
bath tub[1]	(V) bath tub (of bathroom style)	1(e)
television[1]	(A) television	1(e)
Internet connection[1]	(A) Internet connection: Boolean	1(e)
number of beds[1]	(A) number of beds	1(e)
bed size[1]	(A) bed size	1(e)
availability[1]	(A) availability: Boolean	1(e)
special offers[1]	(A) special offers (on rooms)	1(e)
charge of a room[2]	(A) charge	1(e)
photographs	(A) photographs (of rooms): enumeration of URLs	2(b)
reserved[4]	(A) reserved: Boolean	4
hotel employees[1]	(C) Hotel Employee	1(a)
authorized[4]	(A) authorized: Boolean	4
enter[3] and update[3] hotel and room information[1]	(AS) enter (Employee, Hotel Description, Room Description) (AS) update (Employee, Hotel Description, Room Description)	3 3
Potential guests[1]	(C) Potential Guest	1(a)
search[3] rooms using a number of[5] search criteria[1]	(AS) search (Potential guest, Room) (1, 1^+) (C) Search Criteria (for search)	3, 5(b) 1(a)
view any[5] of the rooms	(AS) view (Potential Guest, Room Description) (1,*)	3, 5(b)
facilities[1]	(C) Facilities	1(a)
such as[9]	(I) ISA (Cafeteria, Bar, Conference Room, Swimming Pool, ..., Facilities)	9
cafeteria[1]	(C) Cafeteria	1(a)
bar[1]	(C) Car	1(a)
conference rooms[1]	(C) Conference rooms	1(a)
swimming pool1	(C) Swimming pool	1(a)
recreational facilities[1]	(C) Recreational facilities	1(a)
directions[1]	(C) Directions	1(a)
map[1]	(C) Map	1(a)
reserve[3] one or more[5]	(AS) reserve (Potential Guest, Room) (1,1+)	3
reservations[1]	(AC) Reservation	1(c)
period[1]	(A) period	1(e)
first night charge[1]	(A) first night charge	1(e)
credit card[1]	(C) Credit Card	1(a)
guaranteed[3]	(AS) guaranteed by (Reservation, Credit Card)	3
charged to[3]	(AS) charged to (Reservation, Credit Card)	3
update[3]	(AS) update (Guest, Reservation)	3
delete[3]	(AS) delete (Guest, Reservation)	3

Figure 5.5: Classifying hotel reservation brainstorming result

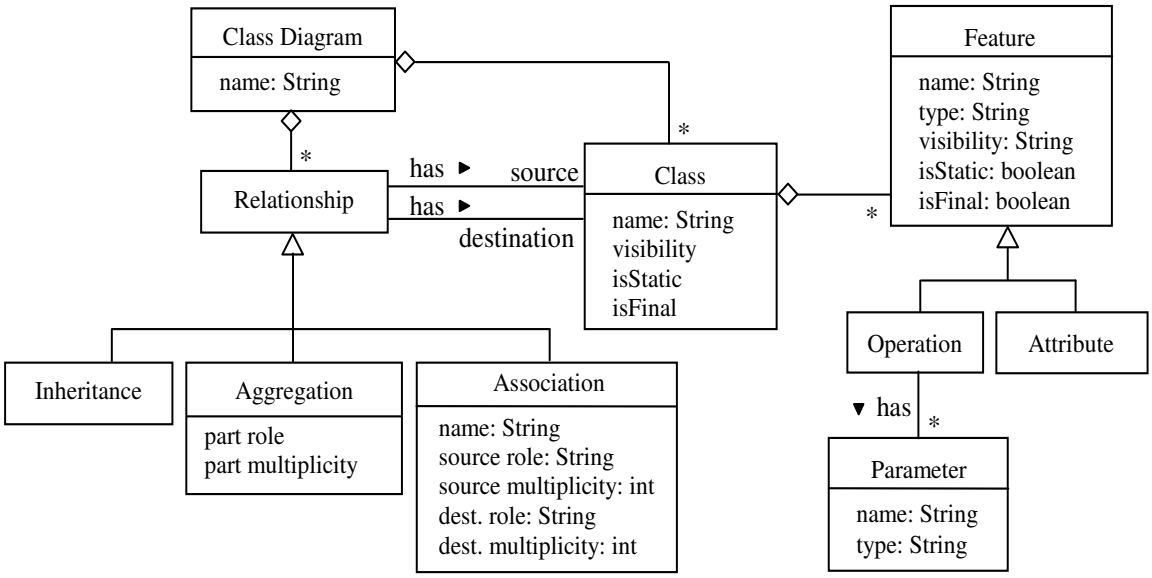


Figure 5.6: A partial domain model for a UML class diagram

class diagram that describes a UML class diagram.

Solution. Figure 5.6 of this manual shows the class diagram. Note the solution may also include a UML diagram notation part, which has classes corresponding to the classes shown in Figure 5.6 to represent the modeling notations for the modeling notions.

Chapter 6

Architectural Design

6.1 Construct a table with rows corresponding to the architectural styles in Figure 6.3, and columns corresponding to the design principles presented in this chapter. Fill in the entries to show which architectural styles apply which design principles.

Solution. See Figure 6.1.

6.2 For each architectural style listed in the previous exercise, briefly explain how it applies the design principles and what are the benefits.

Solution. Explanations for five of the styles are shown in Figure 6.2.

6.3 In exercise 4.4, you produced the software requirements specification (SRS) for a calendar

	Design for change	Separation of concerns	Information hiding	High cohesion	Low coupling	Keep it simple and stupid
N-tier	√	√	√	√		√
Client server	√		√		√	
Main program and subroutines		√		√	√	
Event-driven	√				√	
Persistence framework	√	√	√	√		√
Peer-to-peer	√				√	
Blackboard	√				√	

Figure 6.1: Mapping architectural styles to design principles

	Design for change	Separation of concerns	Information hiding	High cohesion	Low coupling	Keep it simple and stupid
N-tier	Design for change is achieved by reducing the dependencies among the tiers so that changes made to one tier would not affect the other.	Separation of concerns is accomplished by properly partitioning and assigning responsibilities to tiers.	(See design for change.)	A consequence of separation of concerns.		Design the tiers so each is responsible for one set of core functions, and lower-level responsibilities are delegated to the next tier.
Client server	Design the server and clients so change to the server has little impact on the clients.		Design the server so that the clients are shielded from server implementation detail.		Clients can be added and removed freely without affecting the server and other clients.	
Main program and subroutines		Design the modules so that each is assigned a distinct functionality.		Design the modules so that each accomplishes a core functionality.	Design the modules to minimize control and data dependencies among the modules.	
Event-driven	Make the components independent from each other and interact only with the controller. This facilitates change.				Components are loosely coupled via the controller, as a consequence of design for change.	
Persistence framework	Design the system so database functions are given to persistence framework. Thus, change to database has little impact to business objects.	Database concerns are separated from objects and assigned to the framework.	The persistence framework hides the database implementation detail.	Responsibilities to store and retrieve different types of objects are assigned to different database access commands, achieving high cohesion.		A consequence of high cohesion, resulting in "stupid objects."

Figure 6.2: Explanation of design principles applied

management system. In this exercise, do the following:

a. Identify the type of system and briefly justify your answer.

b. Produce an architectural design for the system.

c. Specify the functionality and interface for each of the subsystems and components in the architectural design.

d. Discuss which software design principles are applied, how they are applied, and the benefits of each of the principles. Also indicate the potential problems, if any.

Solution. The system is an interactive system. Therefore, a typical N-tier architecture should be used. The architecture design would consist of a presentation or GUI layer, a controller layer that decouples the presentation from the business objects, and a persistence object layer to store the calendar information. Figure 6.4(b) of the textbook provides a general architecture, which can be adapted for calendar management. The design principles applied are the ones checked for the N-tier and persistence framework architectural styles shown in Figure 6.1 of this manual.

6.4 Do the following for the Study Abroad Management System (SAMS) presented in Chapter 4.

a. Identify the type of system and briefly justify your answer.

b. Identify an architectural style and produce an architectural design for the system.

c. Specify the functionality and interface for each of the subsystems and components in the architectural design.

d. Discuss which software design principles are applied in the design, how they are applied, and the benefits of applying each of the principles. Also point out the potential problems associated with the application of the design principles, if any.

Solution. The answer to this exercise is very similar to the calendar system except that the contents are different. Figure 6.4(b) of the textbook gives the general architectural design, which can be adapted for SAMS.

6.5 Consider the Airport Baggage Handling System discussed in Chapter 3. The conveyor subsystem employs bar-code scanners and pushers to guide the pieces of luggage to travel toward their destinations on the conveyor belts. There is a software subsystem that works with these two types of devices. Determine the type of this subsystem. Note that the subsystem may involve more than one type of subsystem. Select the architectural style to apply. Also produce a sketch of the architectural design and specify the functionality for each of the subsystems and components.

Solution. The use of hardware components implies that it is an embedded system. Therefore, the subsystem that works with the bar-code scanner and pushers should use the event-driven architectural style. The state machine will keep tracks of subsystem state, handle events sent from these devices, and issue instructions to control these devices. If the system needs to process operator requests such as configuring and querying the devices, then there may also be an iterative components and an N-tier architecture could be used.

6.6 In exercise 4.5 you produced the SRS for the web-based multiple-room reservation system. In this exercise, you are required to identify the type of system and sketch an architectural design for the system. Also briefly specify the functionality for each of the subsystems and components in the architectural design.

Solution. This is a typical web-based interactive system. Therefore, a N-tier architecture could be used. Figure 6.4(b) of the textbook shows the general architecture. One needs only to tailor the layers for the room reservation application.

Part III

MODELING AND DESIGN OF INTERACTIVE SYSTEMS

Chapter 7

Deriving Use Cases from Requirements

7.1 You produced a requirements specification for a desktop virtual calculator, a telephone answering machine, and a web-based online email system in exercise 4.2. Derive use cases, related actors, and systems for each of these applications. Also specify the high-level use cases and draw use case diagrams for the use cases.

Solution. The desktop virtual calculator seems to have only one use case. That is, "perform computation."

Answering machine use cases:

UC 1. Check Messages (Actor: user, System: Answering Machine)

UC 2. Delete a Message (Actor: user, System: Answering Machine)

UC 3. Record Greeting Message (Actor: user, System: Answering Machine)

UC 4. Set Date and Time (Actor: user, System: Answering Machine)

UC 5. Check Messages Remotely (Actor: user, System: Answering Machine)

Email system use cases include:

UC 1. Login (Actor: User, System: Email System)

UC 2. Logout (Actor: User, System: Email System)

UC 3. Check Email (Actor: User, System: Email System)

UC 4. Send Email (Actor: User, System: Email System)

UC 5. Edit Address Book (Actor: User, System: Email System)

UC 6. Manage Folders (Actor: User, System: Email System)

UC 7. Set Preferences (Actor: User, System: Email System)

UC 8. Edit Personal Profile (Actor: User, System: Email System)

UC 9. Delete Email (Actor: User, System: Email System)

UC 10. Move Email (Actor: User, System: Email System)

UC 11. Forward Email (Actor: User, System: Email System)

UC 12. Reply to Email (Actor: User, System: Email System)

7.2 Derive use cases from the functional requirements for the calendar management system you produced in exercise 4.4. Also specify the high-level use cases.

Solution. (Omitted.)

7.3 For the National Trade Show Services (NTSS) requirements specification you produced in exercise 4.6, do the following:

a. Derive use cases from the requirements specification.

b. Specify the high-level use cases.

c. Draw use case diagrams.

d. Apply the use case partition rules if necessary to reduce the number of use cases allocated to each subsystem.

Solution. Figure 7.1 and Figure 7.2 of this manual show the use cases, high-level use cases, and use case diagrams for this exercise.

7.4 Do the following for the Car Rental System (CRS) given in Appendix D.1:

a. Derive use cases from the requirements you produced in exercise 4.8.

b. Construct a requirements-use case traceability matrix.

c. Specify the high-level use cases.

d. Draw use case diagrams.

Solution. A partial solution is given here. Example use cases that can be derived for the car rental system are (actor and system are omitted):

UC1. Create Account (a solution may or may not include this use case)

UC2. Search for a Car

UC3. Reserve a Car

UC4. Update a Reservation

UC5. Delete a Reservation

UC6. Checkout a Reserved Car

UC7. Make Block Reservation (this could be merged with UC3)

UC8. Process Reservation

UC9. Return a Car

UC10. Pay Invoice

UC11. Create Rental Contract

UC12. View Reservations

UC13. View Invoices

The requirements-use case traceability matrix depends on the requirements. That is, if a use

1. Create Account (Actor: User, System: Account Management)
TUCBW: User clicks on Create Account on NTSS homepage.
TUCEW: User sees confirmation that new account is ready to use.

2. Login (Actor: User, System: Account Management)
TUCBW: User enters credentials on login page.
TUCEW: User sees welcome message.

3. Logout (Actor: User, System: Account Management)
TUCBW: User clicks Logout on any page.
TUCEW: User sees farewell message.

4. Create Trade Show (Actor: Event organizer, System: NTSS/Event Management)
TUCBW: Event organizer clicks Create Trade Show button on Welcome page.
TUCEW: Event organizer sees the new trade show listed on the trade show list.

5. Organize Trade Show (Actor: Event organizer, System: NTSS/Event Management)
TUCBW: Event organizer selects Organize on trade show's page.
TUCEW: Event organizer sees confirmation that trade show organization activities are successfully scheduled.

6. Submit Event Proposal (Actor: Event organizer, System: NTSS/Event Management)
TUCBW: Event organizer clicks Create Event Proposal on home page.
TUCEW: Event organizer sees confirmation that proposal is submitted.

7. Cancel an Event Proposal (Actor: Event organizer, System: NTSS/Event Management)
TUCBW: Event organizer clicks Cancel Event Proposal on homepage.
TUCEW: Event organizer sees event removed from event list.

8. Edit Event Proposal (Actor: Event organizer, System: NTSS/Event Management)
TUCBW: Event organizer clicks Edit on a pending review proposal.
TUCEW: Event organizer sees confirmation that updated proposal is saved.

9. Evaluate Event Proposal (Actor: NTSS staff, System: NTSS/Staff)
TUCBW: NTSS staff clicks Evaluate Proposal on a submitted proposal.
TUCEW: NTSS sees the decision and feedback for the proposal updated in the system.

10. Submit Booth Lease Request (Actor: Exhibitor, System: NTSS/Booth Management)
TUCBW: Exhibitor clicks on Lease Booth on homepage.
TUCEW: Exhibitor sees booth lease request submitted successfully.

11. Withdraw Booth Lease Request (Actor: Exhibitor, System: NTSS/Booth Management)
TUCBW: Exhibitor clicks on Withdraw Booth Lease on booth lease request.
TUCEW: Exhibitor sees the booth lease request removed from the leased booth list.

12. Evaluate Booth Lease Request (Actor: NTSS staff, System: NTSS/Staff)
TUCBW: NTSS staff clicks Evaluate Booth Lease Request button on homepage.
TUCEW: NTSS staff sees the decision and feedback for lease booth requests updated.

13. Register for Event (Actor: Event Participant, System: NTSS/Registration)
TUCBW: Event participant clicks Register on the event's page.
TUCEW: Event participant is notified of successful registration.

14. Cancel Registration (Actor: Event Participant, System: NTSS/Registration)
TUCBW: Event participant clicks Cancel button on the event page the user is registered for.
TUCEW: Event participant sees confirmation of cancellation and event removed from the user's registered event list.

Figure 7.1: NTSS use cases and high-level use cases

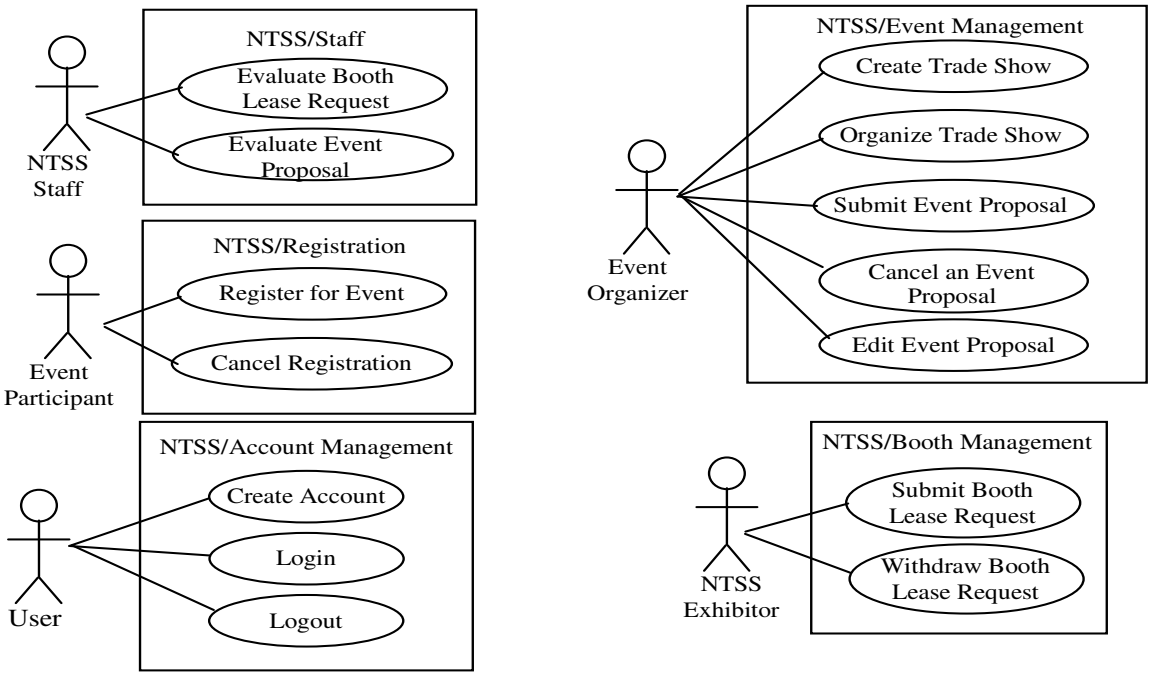


Figure 7.2: NTSS use case diagrams

case is derived from a requirement, then the entry for the requirement and use case is checked. The solution to this part is not difficult and is omitted.

The following are some of the high level use cases:

- UC2. Search for a Car

  TUCBW the customer clicks the Search Car link on any of the CRS Web pages.

  TUCEW the customer sees a list of cars that satisfy her/his search criteria.

- UC3. Reserve a Car

  TUCBW the customer

  (1) clicks the Reserve a Car link on any of the CRS Web pages, or

  (2) checks the cars he/she wants to reserve on the list of cars displayed by the Search for a Car use case and clicks the Reserve link on that page.

  TUCEW the customer sees a message stating that the reservation is completed success-

fully.

- UC4. Update a Reservation

  TUCBW the customer clicks the Update Reservation link on any of the CRS Web pages.

  TUCEW the customer sees a message stating that the reservation is updated successfully.

- UC5. Delete a Reservation

  TUCBW the customer clicks the Delete Reservation link on any CRS page.

  TUCEW the customer sees a message stating that the reservation is successfully deleted.

7.5 Formulate your functional requirements for a use case modeling tool that implements the methodology described in this chapter. Limit the specification to no more than a couple of pages. Derive the use cases, specify the high-level use cases, and produce the use case diagrams. Review these using the review checklist and produce a review report.

Solution. Figure 7.3 of this manual shows a list of requirements. Below are the use cases. The use case diagram is easy to draw and is omitted.

UC1. Derive Use Cases (Actor: user, System: UC Modeling Tool)

UC2. Generate Use Case Diagrams (Actor: user, System: UC Modeling Tool)

UC3. Edit Use Case Diagram (Actor: user, System: UC Modeling Tool)

Use Case Modeling Tool Requirements

R1. The IDE shall have a GUI interface.
R2. The IDE shall be able to load requirements from a file.
R3. The IDE shall allow the user to input requirements.
R4. The IDE shall allow the user to derive use cases by highlighting verb-noun phrases, the actor, and the system on the requirements.
R5. The IDE shall have the option to prompt the user to verify the use cases identified using the following verification criteria:
  R5.1 The IDE shall ask the user to verify that the identified use case is a business process.
  R5.2 The IDE shall ask the user to verify that the identified use case is initiated by an actor.
  R5.3 The IDE shall ask the user to verify that the identified use case ends with an actor.
  R5.4 The IDE shall ask the user to verify that the identified use case accomplishes something useful for the actor.
R6. The IDE shall display the use cases along with respective actors and systems in a table.
R7. The IDE shall automatically layout and display use case diagrams for the use cases identified.
R8. The IDE shall allow the user to edit use case diagrams generated.
R9. The IDE shall allow the user to save generated or edited use case diagrams.

Figure 7.3: Requirements for a use case modeling tool

Chapter 8

Actor-System Interaction Modeling

8.1 Write expanded use cases for the Deposit Money, Withdraw Money, Check Balance, and Transfer Funds use cases of an automatic teller machine (ATM) application.

Solution. Three expanded use cases are shown in Figure 8.1 of this manual including the login use case, which is used by the other two use cases.

8.2 Write expanded use cases for the Edit Report use case for a typical word processor.

Solution. Omitted.

8.3 Write expanded use cases for the Make Payment use case of a retail business. Assume that the customer can Pay with Check, Pay with Credit Card, and Pay with Cash.

Solution. Figure 8.2 of this manual shows the expanded use cases for the Make Payment use case.

8.4 Write expanded use cases for the Search Car, Reserve Car, Edit Reservation, and Cancel Reservation use cases for the online car rental application described in Appendix D.1.

Solution. Figure 8.3 shows a sample solution.

UC 1. Login

Actor: ATM Customer	System: ATM
	0) ATM displays the “Please insert card” screen.
1) TUCBW customer inserts an ATM card into the card slot.	2) ATM displays the Enter Pin dialog.
3) Customer enters the pin number and presses the Submit button.	4) ATM displays the Transaction Selection screen.
5) TUCEW customer sees the Transaction Selection screen.	

UC 2. Withdraw Money

Precondition: This use case assumes that the customer has logged into the ATM.	
Actor: ATM Customer	System: ATM
	0) ATM displays the Transaction Selection screen.
1) TUCBW customer presses the Withdraw button on the Transaction Selection screen.	2) ATM asks the customer to enter the amount to be withdrawn.
3) Customer enters the amount and presses the Submit button.	4) ATM displays a confirmation of the amount of cash dispensed, and dispenses the cash.
5) TUCEW customer presses the OK button on the confirmation dialog and gets the correct amount of cash dispensed.	

UC 3. Transfer Money

Precondition: This use case assumes that the customer has logged into the ATM.	
Actor: ATM Customer	System: ATM
	0) ATM displays the Transaction Selection screen.
1) TUCBW customer presses the Transfer button on the Transaction Selection screen.	2) ATM asks the customer to enter the bank account and the amount to be transferred.
3) Customer enters the bank account, amount to transfer, and presses the Submit button.	4) ATM displays a confirmation that the amount has been transferred to the bank account.
5) TUCEW customer sees the confirmation message and presses the OK button on it.	

Figure 8.1: Expanded use cases for an ATM

UC 1. Make Payment

Precondition: This use case assumes that the customer has finished scan of the items, and the system displays the total amount due.	
Actor: Retail Customer	System: Self Checkout Machine
	0) System displays the total amount due.
1) TUCBW customer presses the Finish and Pay button on the self checkout machine.	2) System displays three payment options: 2.1) Pay by a credit card. 2.2) Pay by a check. 2.3) Pay by cash.
3) Customer presses one of the payment options.	4) System continues with one of the following use cases: 4.1) Pay by Credit Card 4.2) Pay by Check 4.3) Pay by Cash
5) TUCEW customer sees the Thank You for Shopping message and the payment slip printed.	

UC 2. Pay by Credit Card

Precondition: This use case assumes that Pay by Credit Card is selected in the Make Payment use case.	
Actor: Retail Customer	System: Self Checkout Machine
	0) System displays the Please Insert a Credit Card message.
1) TUCBW customer inserts a credit card into the card slot and removes it momentarily.	2) System displays a dialog requesting the customer to enter the zip code.
3) Customer enters the zip code and presses the Enter key.	4) System displays the Thank You for Shopping message and prints the payment slip.
5) TUCEW customer sees the Thank You for Shopping message and the payment slip printed.	

UC 3. Pay by Check

Precondition: This use case assumes that Pay by Check is selected in the Make Payment use case.	
Actor: Retail Customer	System: Self Checkout Machine
	0) System displays a dialog requesting the customer to insert a check and enters his driver's license number.
1) TUCBW customer inserts a blank check into the check insertion slot, enter his driver's license number, and presses the Enter key.	2) System prints the payment amount and the driver's license number on the check, and asks the customer to sign and insert the check.
3) Customer signs the check and inserts it into the check insertion slot.	4) System displays the Thank You for Shopping message and prints the payment slip.
5) TUCEW customer sees the Thank You for Shopping message and the payment slip printed.	

UC 4. Pay by Cash

Precondition: This use case assumes that Pay by Cash is selected in the Make Payment use case.	
Actor: Retail Customer	System: Self Checkout Machine
	0) System displays a dialog requesting the customer to insert the required amount of cash.
1) TUCBW customer inserts cash into the bill and/or coin insertion slots until the required amount is inserted.	2) System displays the total amount inserted and the Thank You for Shopping message, and prints the payment slip.
3) TUCEW customer sees the Thank You for Shopping message, the payment slip printed, and the change dispensed.	

Figure 8.2: Make Payment and related expanded use cases

UC 1. Search Car

Actor: Customer	System: CRS
	0. CSR displays the homepage.
1. TUCBW customer clicks the Search Car link on the homepage.	2. CRS displays the Search Car form.
3. Customer fills the Search Car form and clicks the Submit button.	4. CSR displays a list of cars satisfying the search criteria.
5. TUCEW customer sees the list of cars satisfying his search criteria.	

UC 2. Reserve Car

Actor: Customer	System: CRS
	0. CSR displays a list of cars resulting from UC 1 (Search Car).
1. TUCBW customer clicks the link for the car he wants to reserve on the list of cars.	2. CRS displays the Reserve Car form.
3. Customer fills the Reserve Car form and clicks the Submit button.	4. CSR displays a reservation successful confirmation message.
5. TUCEW customer sees the reservation successful confirmation message.	

UC 3. Edit Reservation

Actor: Customer	System: CRS
	0. CSR displays the homepage.
1. TUCBW customer clicks Edit Reservation link on the homepage.	2. CRS displays a Search Reservation form with search options.
3. Customer fills the Search Reservation form and clicks the Submit button.	4. CSR displays the reservation satisfying the search reservation criteria.
5. Customer edits the reservation and clicks the Save button.	6. CSR displays a reservation updated confirmation message.
7. TUCEW customer sees the reservation updated confirmation message.	

UC 4. Cancel Reservation

Actor: Customer	System: CRS
	0. CSR displays the homepage.
1. TUCBW customer clicks Cancel Reservation link on the homepage.	2. CRS displays a Search Reservation form with search options.
3. Customer fills the Search Reservation form and clicks the Submit button.	4. CSR displays the reservation satisfying the search reservation criteria.
5. Customer clicks the Cancel Reservation button.	6. CSR displays a confirmation message requesting the customer to confirm the cancellation.
7. Customer checks the Confirm Cancellation checkbox and clicks the Submit button.	8. CSR displays a reservation cancelled message.
9. TUCEW customer sees the reservation cancelled confirmation message.	

Figure 8.3: Car rental system expanded use cases

8.5 In Appendix D.2, a National Trade Show Services (NTSS) business is described. In exercise 4.6 and exercise 7.3, you produced the requirements and derived use cases for this application. Now write the expanded use cases.

Solution. Figure 8.4 of this manual shows some of the expanded use cases for the NTSS project.

8.6 Formulate several requirements for a state diagram editor. Derive use cases from these requirements and produce the corresponding expanded use cases

Solution. The general requirements for a state diagram editor are:

"The state diagram editor shall allow a user to edit a new diagram or an existing diagram. The editor shall allow the user to add, delete, and edit states and transitions as well as undo and redo editing operations. The editor shall allow the user to perform other operations such as saving a diagram."

One important use case is Edit State Diagram, which has an expanded use case as shown in Figure 9.12 of this manual.

UC1. Create an Account

Actor: User	System: NTSS/Account Management
1. **TUCBW**: User clicks the Create Account link.	2. The system displays the account creation form.
3. The user enters credentials and clicks the Submit button.	4. The system displays a confirmation message or an error message.
5. **TUCEW**: User is shown the message that the account is created successfully and an email confirmation is sent.	

UC2. Login (See Figure 8.10 of the textbook)

UC3. Logout

Actor: User	System: NTSS/Account Management
	0. System displays the logout button on all pages.
1. TUCBW: User clicks the Logout button.	2. The system displays a Successfully Logged out message.
3. TUCEW: User sees the Successfully Logged out message.	

UC6. Submit Event Proposal

Precondition: The event organizer had logged into the system.	
Actor: Event Organizer	System: NTSS/Event Management
1. **TUCBW**: User clicks the Submit Event Proposal link on the homepage.	2. System displays a proposal form.
3. The user fills in proposal information in the form and clicks the Submit button.	4. System displays a message stating that the event proposal is submitted and pending review, and a payment slip has be emailed.
5. **TUCEW**: User see the event proposal submitted successfully message.	

UC7. Cancel an Event Proposal

Precondition: The use case assumes that the event organizer had logged in and the event proposal exists.	
Actor: Event Organizer	System: NTSS/Event Management
1. **TUCBW**: Event organizer clicks the Cancel Event Proposal button.	2. System displays proposals submitted by the event organizer.
3. The event organizer selects the event proposal and clicks Cancel.	4. System displays a message showing the 15% cancellation charges and asks the event organizer to confirm cancellation.
5.The event organizer clicks Confirm Cancellation button.	6. The system displays a message that the proposal has been cancelled and the money has been refunded.
7. **TUCEW**: User sees the message that the event proposal is cancelled and an email confirmation is sent.	
Post-condition: The event proposal status is changed to “cancelled.”	

Figure 8.4: NTSS expanded use cases - part 1

UC8. Edit an Event Proposal

Precondition: The use case assumes that the event organizer had logged in and the event proposal exists.	
Actor: Event Organizer	System: NTSS/Event Management
1. **TUCBW**: Event organizer clicks the View/Edit proposal button.	2. System displays the event proposals submitted by the event organizer.
3. The Event organizer selects the event proposal and clicks the Edit button.	4. System displays the proposal detail.
5. The event organizer edits the event proposal and clicks the Save button when editing is done.	6. System displays a message indicating the edited event proposal is saved.
7. **TUCEW**: The event organizer is shown the event proposal updated successfully message.	

UC9. Evaluate Event Proposal

Actor: NTSS Staff	System: NTSS/Event Management
1. **TUCBW**: Staff clicks the Evaluate Event Proposal button.	2. System displays a list of events pending review.
3. The staff selects the event proposal and clicks the Review button.	4. System displays the event proposal content.
5. The staff reviews the proposal, enters feedback and acceptance decision, and clicks the Submit button.	6. System displays the review result saved message.
7. **TUCEW:** Staff is shown the review result saved message.	

UC10. Submit Booth Lease Request

Actor: Exhibitor	System: NTSS/Booth Management
1. **TUCBW**: Exhibitor clicks the Submit Booth Lease Request button on the homepage.	2. System displays booth floor plans with available booths highlighted and prompts exhibitor to select a booth.
3. The exhibitor selects booths to lease, enters exhibitor information, and clicks the Submit button.	4. System displays a summary including payment amount, and requests the exhibitor to confirm the lease request.
5. The exhibitor reviews the information and clicks the Confirm button.	6. System displays a confirmation that selected booths are reserved for the exhibitor, and a payment email has been sent.
7. **TUCEW**: Exhibitor is displayed the confirmation message.	

UC11. Withdraw Booth Lease Request

Actor: Exhibitor	System: NTSS/Booth Management
Precondition: The use case assumes that the user had successfully created a booth lease request.	
1. **TUCBW**: Exhibitor clicks the Cancel Booth Lease Request button.	2. System displays the booth lease requests submitted by the event exhibitor.
3. Exhibitor selects the booth lease request and clicks the Cancel button.	4. System displays the 10% cancellation charges and asks the exhibitor to confirm cancellation.
5. Exhibitor clicks Confirm Cancellation button.	6. The system displays a message stating that the booth lease request is cancelled, and the amount refunded.
7. **TUCEW**: Exhibitor sees the message that the booth lease request is cancelled and refunded.	
Post-condition: The booth lease request status has changed to “cancelled.”	

Figure 8.4: NTSS expanded use cases - part 2

UC12. Evaluate Booth Lease Request

Actor: NTSS Staff	System: NTSS/Staff
1. **TUCBW:** NTSS staff clicks the Evaluate Booth Lease Request button.	2. System displays a list of booth lease requests pending review.
3. NTSS staff selects the booth lease request and clicks the Review button.	4. System displays the booth lease request detail.
5. NTSS staff enters feedback and rejection reason, if any, and clicks the Accept or Reject button.	6. System displays the message that the review result is saved and the decision is emailed to the exhibitor.
7. **TUCEW**: NTSS staff sees the review result saved message.	

UC13. Register for Event

Precondition: The use case assumes that the user had successfully logged in to the system	
Actor: Event Participant	System: NTSS/Registration
1. **TUCBW**: Event Participant clicks the Register Event button.	2. System displays a list of events that the participant can register for.
3. Event participant selects the event and clicks the Register button.	4. System prompts the participant to enter registration information.
5. Event participant enters the required information and clicks the Submit button.	6. System displays a confirmation message that registration is successful.
7. **TUCEW**: Event participant sees the registration is successful message.	

UC14. Cancel Registration

Precondition: This use case assumes that the event participant had registered to an event.	
Actor: Event Participant	System: NTSS/Event Management
1**. TUCBW**: Event participant clicks the Cancel Registration button on any page.	2. System displays a list of events that the user has registered for.
3. Event participant selects the event and clicks the Submit button.	4. System displays the cancellation charges and asks the user to confirm cancellation.
5. Event participant clicks the Confirm Cancellation button.	6. System displays a message stating that cancellation is successful and the refunded amount.
7. **TUCEW**: Event participant sees the cancellation message.	
Post-condition: The registration status has changed to “cancelled.”	

Figure 8.4: NTSS expanded use cases - part 3

Chapter 9

Object Interaction Modeling

9.1 Write a scenario for the Order a Dish use case of a manually operated restaurant. For simplicity, assume that only one dish is ordered.

Solution. An underlined scenario is shown in Figure 9.1 of this manual.

9.2 Highlight the subjects, subject actions, the objects acted upon, and data and objects required by the subject action in the Order a Dish scenario.

Solution. See solution to Exercise 9.1.

9.3 Represent the Order a Dish scenario using a scenario table.

1. The customer enters the restaurant.
2. The waiter greets and sits the customer.
3. The waiter hands the *menu* to the customer.
4. The customer selects and orders the *dish*.
5. The waiter writes the *order* on a ticket.
6. The waiter hands over the *ticket* to the chef.
8. The chef cooks the dish.
9. The chef informs the waiter that *dish is ready*.
10. The waiter gets the *dish*
11. The waiter serves the customer.
12. The customer enjoys the dish.
13. The customer asks waiter for the *check*.
14. The waiter hands the *check* to the customer.
15. The customer pays the *check*.
16. The customer tips the waiter.

Legend:

Underscore:	subject
Dashed underscore:	subject action
Double underscore:	object acted upon
Italic:	parameters

Figure 9.1: Order dish scenario

#	Subject	Subject Action	Parameters	Object Acted Upon
1.	The customer	enters		the restaurant.
2.	The waiter	greets and sits		the customer.
3.	The waiter	hands	the *menu*	to the customer.
4.	The customer	selects and orders	the *dish*.	
*5.	The waiter	writes	the *order*	on a ticket.
6.	The waiter	hands over	the *ticket*	to the chef.
8.	The chef	cooks		the dish.
9.	The chef	informs	dish is ready	the waiter.
10.	The waiter	gets	the *dish*	
11.	The waiter	serves		the customer.
12.	The customer	enjoys		the dish.
13.	The customer	asks for	*check*	from waiter.
14.	The waiter	hands	the *check*	to the customer.
15.	The customer	pays	the *check*.	
16.	The customer	tips		the waiter.

Figure 9.2: Order dish scenario table

Solution. Figure 9.2 of this manual shows the scenario table.

9.4 Construct an informal sequence diagram for the Order a Dish scenario.

Solution. See Figure 9.3.

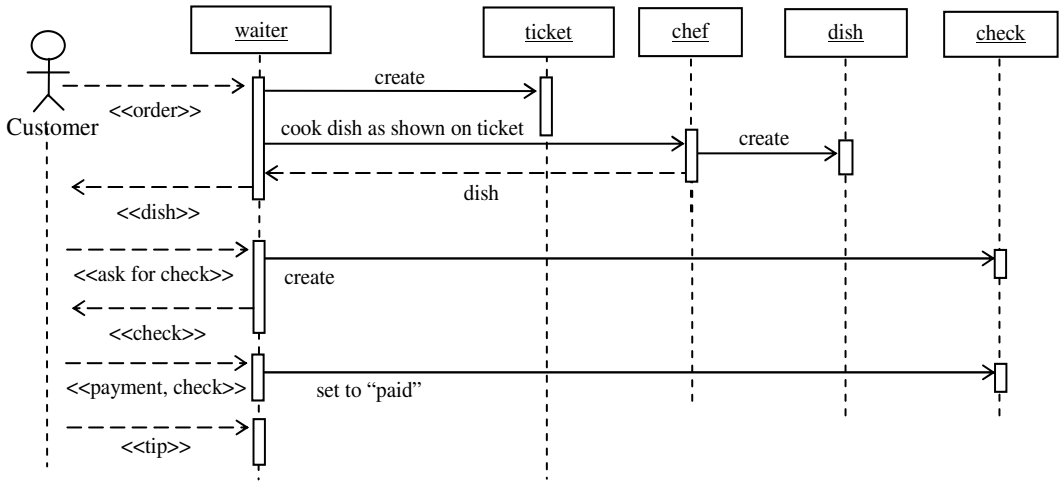


Figure 9.3: Informal sequence diagram for order a dish

9.5 Produce scenarios, scenario tables, and design sequence diagrams for the Deposit Money, Withdraw Money, Check Balance, and Transfer Money expanded use cases you produced for the ATM application in exercise 8.1.

3. Customer enters the (withdraw) *amount* and presses the Submit button (on the ATM GUI).
4.1. The ATM GUI withdraws the *amount* from the withdraw money controller.
4.2. The withdraw money controller verifies *amount* with the customer account.
4.3. The customer account returns true or false to withdraw money controller.
4.4. If true is returned then
  4.4.1. The withdraw money controller creates a success message with withdraw *amount*.
  4.4.2. The withdraw money controller calls the dispenser to dispense the withdraw *amount*.
  4.4.3. The withdraw money controller deducts the withdraw *amount* from the customer account.
  4.4.5. The withdraw money controller saves the customer *account* with the database manager.
  4.4.3. else
  4.4.4. The withdraw money controller creates a message stating *"funds are insufficient to fulfill request."*
4.5. The withdraw money controller returns the *message* to the ATM GUI.
4.6. ATM GUI displays the *message* to the customer.

Figure 9.4: Underlined scenario for the nontrivial step of withdraw money expanded use case

#	Subject	Subject Action	Parameters	Object Acted Upon
3.	Customer	enters	*amount*	ATM GUI
4.1.	ATM GUI	withdraws	*amount*	withdraw money controller.
4.2.	withdraw money controller	verifies	*amount*	account.
4.3.	account	returns	true or false to	withdraw money controller.
4.4.	If true is returned then			
4.4.1.	withdraw money controller	creates	*amount*	message.
4.4.2.	withdraw money controller	dispense	*amount*	dispenser
4.4.3.	withdraw money controller	deducts	*amount*	account.
4.4.5.	withdraw money controller	saves	*account*	database manager.
4.4.3.	else			
4.4.4.	withdraw money controller	creates	*"funds are insufficient to fulfill request."*	message
4.5.	withdraw money controller	returns	*message*	ATM GUI.
4.6.	ATM GUI	displays	*message*	customer.

Figure 9.5: Withdraw money scenario table

Solution. The login scenario and scenario table are similar to the one shown in Figure 9.17 of the textbook. For the Withdraw Money expanded use case (UC2) shown in the solution to Exercise 8.1, step 4) is the nontrivial step. Figures 9.4 to 9.7 show the scenario, scenario table, informal sequence diagram, and design sequence diagram, respectively, for the nontrivial step. The other use cases can be solved similarly.

9.6 For the online car rental application described in Appendix D.1, produce scenarios, scenario tables, and design sequence diagrams for the nontrivial steps of the Search Car, Reserve Car, Edit Reservation, and Cancel Reservation expanded use cases you produced in exercise 8.4.

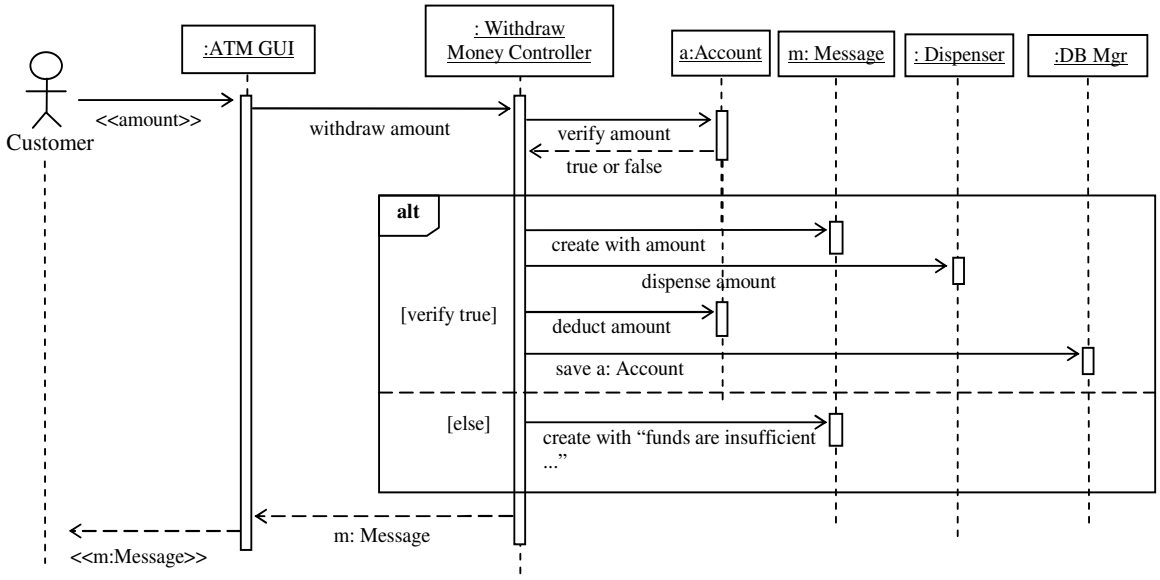


Figure 9.6: Withdraw money informal sequence diagram

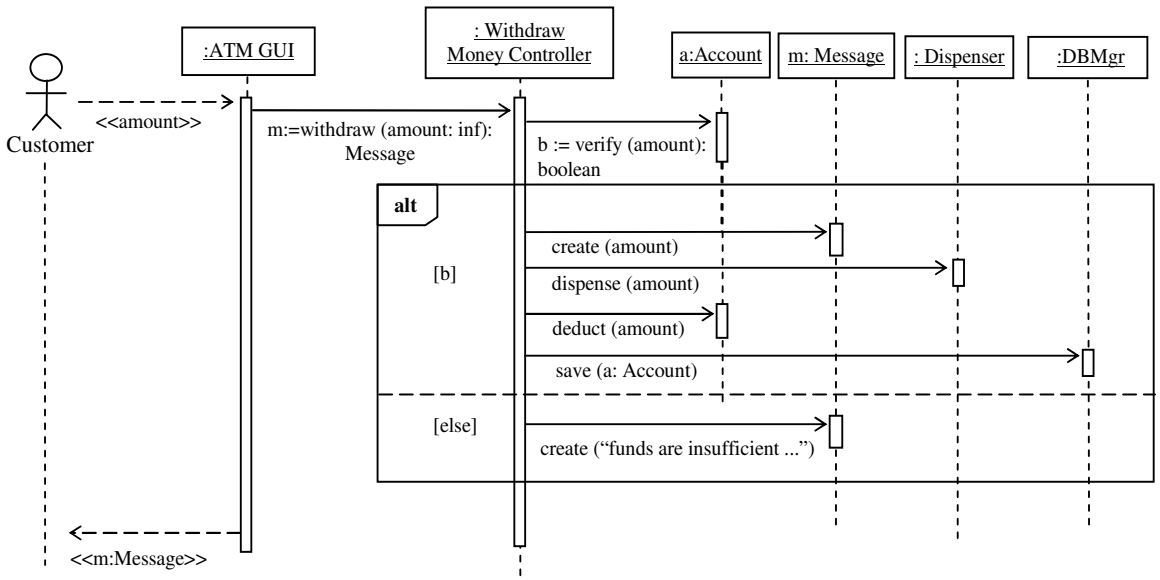


Figure 9.7: Withdraw money design sequence diagram

UC1. Search for Car

Actor: Customer	System: CRS
	0. CRS displays the homepage.
1. TUC BW customer clicks Search for Car link on the homepage.	2. CRS displays a search form allowing the customer to specify search criteria.
3. Customer specifies the search criteria and clicks the Submit button.	*4. CRS displays a list of cars satisfying the search criteria.
5. TUCEW customer sees the search result.	

Figure 9.8: Search car expanded use case

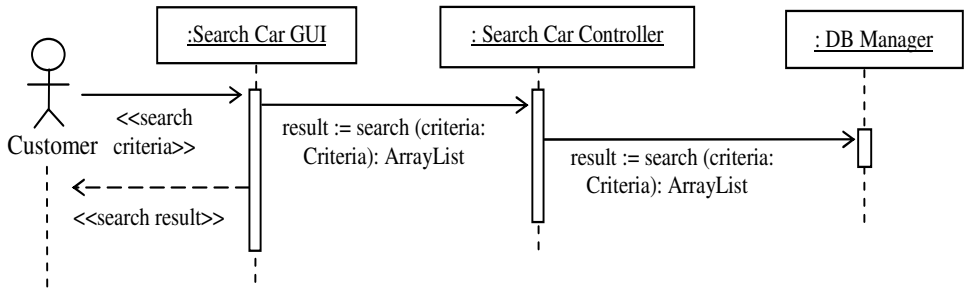


Figure 9.9: Search for car sequence diagram

Solution. Figures 9.8 through 9.11 of this manual show the expanded use cases and sequence diagrams for the Search for Car, and Reserve Car use cases.

9.7 Produce scenarios, scenario tables, and design sequence diagrams for the nontrivial steps of the expanded use cases you produced for the state diagram editor problem in exercise 8.6.

Solution. As shown in Figure 9.12 of this manual, the nontrivial actor request and nontrivial step are steps 3 and 4. A high-level scenario description for these two steps could be:

UC2. Reserve Car

Precondition: Customer had searched car and CRS displays a list of cars satisfying search criteria.	
Actor: Customer	System: CRS
	0. CRS displays a list of cars satisfying search criteria.
1. TUCBW customer clicks the Reserve button next to the car he wants to reserve.	2. CRS displays a reservation form.
3. Customer fills the reservation form and clicks the Submit button.	*4. CRS displays a confirmation message stating that the car is reserved and an email is sent to the customer.
5. TUCEW customer sees the confirmation message.	

Figure 9.10: Reserve car expanded use case

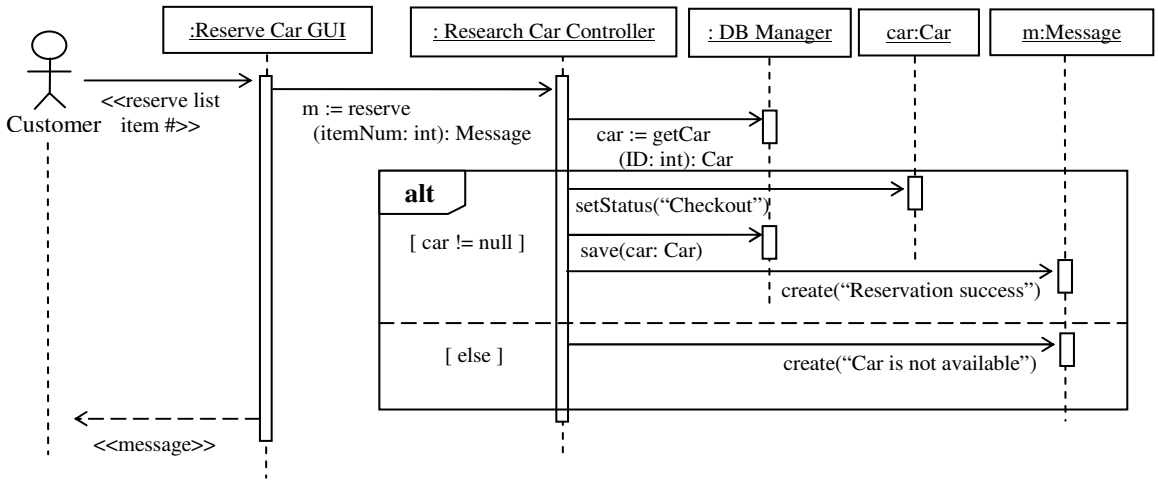


Figure 9.11: Reserve car sequence diagram

UC1. Edit State Diagram

Actor: Editor User	System: State Diagram Editor
	0) System displays the editor main window.
1) TUCBW editor user clicks File on the menu bar then selects 1.1) New Diagram, or 1.2) Open Diagram, and 1.2.1) locates the diagram and clicks the OK button.	2) System accordingly displays: 2.1) a blank diagram, or 2.2) a State Diagram Selection Dialog, and 2.2.1) displays the state diagram selected.
3) Editor user repeatedly performs one of the following editing operations: 3.1) User clicks the State button. 3.1.1) User clicks somewhere in the drawing area. 3.2) User clicks the Transition button. 3.2.1) User presses mouse on the source state, drags to the destination state and releases. 3.3) User double-clicks a state or transition. 3.3.1) User edits the state or transition and clicks the OK or Cancel button. 3.4) User clicks Edit on the menu bar then selects Undo or Redo.	4) System responds as follows: 4.1) System changes the pointer to a crosshair. 4.1.1) System depicts a state shape with a dummy name. 4.2) System changes the pointer to a crosshair. 4.2.1) System depicts a transition with a dummy label from the source state to the destination state. 4.3) System displays an Edit State/Edit Transition Dialog. 4.3.1) System displays the modified or unchanged state diagram. 4.4) System displays state diagram with the previous operation undone/redone.
5) When done with editing user clicks File on the menu bar then selects 5.1) Save, or 5.2) Save As. 5.2.1) User fills in the requested information and clicks the OK button.	6) System responds as follows: 6.1) System displays "Diagram Saved" in the status bar, or 6.2) System displays a Save State Diagram As dialog. 6.2.1) System displays "Diagram Saved As ..." in the status bar.
7) TUCEW editor user sees the diagram saved message in the status bar.	

Figure 9.12: Edit state diagram expanded use case

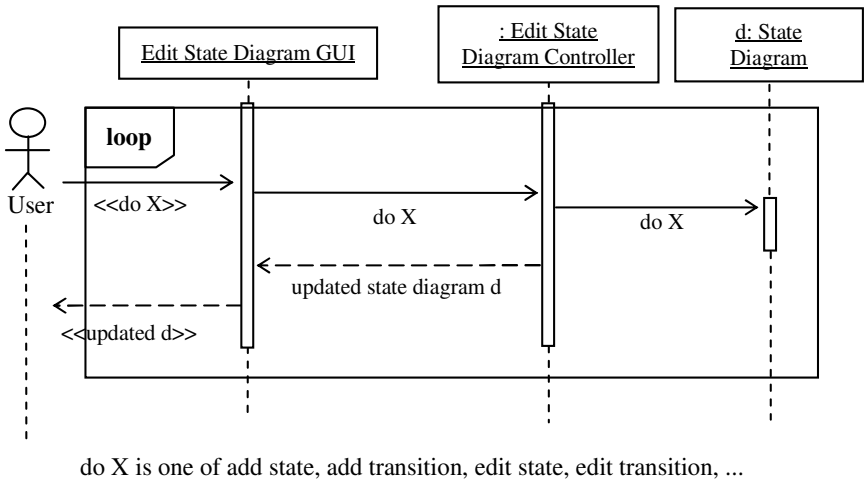


Figure 9.13: Sequence diagram for Edit State Diagram

3. User repeatedly performs an editing operation doX.

4. System accordingly displays the updated state diagram.

where doX is one of (other operations such as delete are omitted):

- add a state
- add a transition
- edit a state
- edit a transition
- undo last operation
- redo last undone operation

Figure 9.13 of this manual shows the informal sequence diagram. The design sequence diagram is similar except that the informal messages between objects are rplaced by function calls, and the dashed arrow lines between objects are replaced by return values from function calls.

9.8 Suppose that you are to design and implement a software tool for OIM using the method-

ology presented in this chapter. The tool should support the methodology steps discussed and automatically generate the design artifacts whenever possible. This includes automatic generation of sequence diagrams from the scenario tables. Thus, there is no need for the user to draw the sequence diagrams manually although the user may need to edit the generated diagrams. The following are some of the analysis and design tasks:

a. Produce a list of requirements for this software tool.

b. Derive use cases from the requirements.

c. Construct a requirements-use case traceability matrix.

d. Construct a domain model for this application.

e. Specify the high-level and expanded use cases.

f. Perform OIM for some of the nontrivial use cases. Which of the OIM steps can be fully or highly automated and how?

Solution. List of requirements:

R1 The OIM shall allow the user to enter, or import a scenario description.

R2 The OIM shall allow the user to classify scenario components: actors, actions, objects, and parameters.

R2.1 The user shall be able to highlight a phrase and classify it by selecting a classification.

R2.2 The OIM shall capture and display the phrase in a scenario table.

R2.3 The OIM shall provide a legend for the highlighted phrases.

R3 The OIM shall allow the user to create a scenario table instead of entering a scenario description.

R3.1 The OIM shall display the Scenario table.

R3.2 The OIM shall allow the user to export/save a scenario table as an XML file.

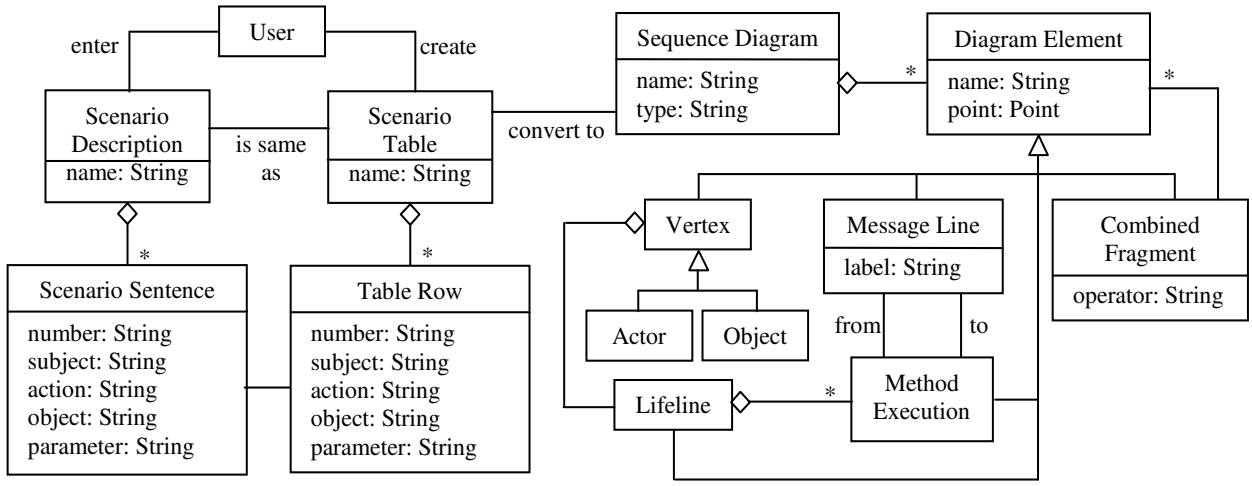


Figure 9.14: Domain model for OIM

R3.3 The OIM shall allow the user to edit a scenario table.

R4 The OIM shall generate a sequence diagram from a scenario table automatically.

Use cases derived from the requirements are:

UC1. Enter Scenario Description

UC2. Classify Scenario Components

UC3. Create Scenario Table

UC4. Generate Sequence Diagram

There is a one-to-one correspondence between the requirements and the use cases — that is Ri derives UCi, i=1,2,3,4. The requirement-use case traceability matrix is a 4x4 matrix showing the correspondence. It is easy to draw and is omitted here. Figure 9.14 of this manual shows the domain model, where not all attributes are displayed to save space. The Classify Scenario Components expanded use case is displayed in Figure 9.15 of this manual.

The nontrivial step is step 2. The scenario description for this nontrivial step is the following. Figure 9.16 of this manual shows the sequence diagram.

UC2. Classify Scenario Components

Precondition: A scenario description has been entered or loaded into the scenario description pane.	
Actor: User	System: OIM Tool
	0) System displays the scenario description.
1) TUCBW user repeatedly highlights, right clicks and selects Subject, Subject Action, Object, or Parameter to classify the highlighted phrase.	2) System accordingly displays the captured phrase in the scenario table under the scenario description pane.
3) TUCEW user sees the completed scenario table.	

Figure 9.15: Highlight Scenario Component expanded use case

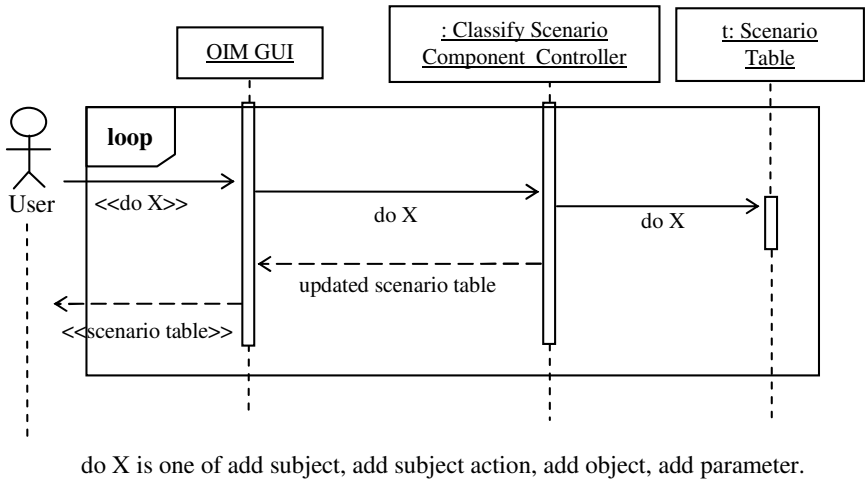


Figure 9.16: Classify scenario components sequence diagram

1. User repeatedly highlights and classifies a scenario component on the OIM GUI.
2. OIM GUI adds the classified scenario component to the scenario table.
3. OIM GUI displays the updated scenario table to the user.

9.9 Modify the expanded use case in Figure 9.8 of the textbook to allow a patron to start the use case before or after log on. If the patron has not logged on, the use case will redirect the patron to the Log On use case. After the patron has logged on, the use case will resume the checkout process. Perform the steps described in this chapter to construct the design sequence diagram for the modified expanded use case.

Solution. Omitted.

Chapter 10

Applying Responsibility-Assignment Patterns

10.1 Construct a table with columns for the controller, expert, and creator patterns and rows for the design principles presented in Chapter 6. Check the entries to indicate which patterns support which design principles. Provide explanations to justify the result.

Solution. Figure 10.1 shows the matrix as well as the explanation.

10.2 See the Online Car Rental application described in Appendix D.1. Its use cases include Search for Cars, Reserve Cars, Edit Reservation, and Cancel Reservation. Do the following and at the same time apply the controller, expert, and creator patterns whenever appropriate:

a. Specify the expanded use cases and identify the nontrivial steps if you have not specified these.

b. Produce scenarios, scenario tables, and sequence diagrams for the nontrivial steps.

Solution. See solution for Exercise 9.6.

10.3 Identify and explain the improvements that the controller, expert, and creator patterns intro-

	Controller	Expert	Creator
Design for Change	The controller pattern decouples the presentation and the business objects. This make it easy to change either of them.	It reduces coupling; and hence, it facilitates change.	A consequence of low coupling.
Separation of Concerns	Different concerns are assigned.		
Information Hiding	Changes to the business objects are shielded from the presentation due to the use of the controller.		
High Cohesion	A consequence of "separation of concerns."	Responsibility is assigned to the object that has the information to fulfill the request. This results in high cohesion.	Responsibility to create an object is assigned to the creator who needs the former to fulfill its functionality. Thus, high cohesion is achieved.
Low Coupling		"Do it myself" reduces coupling.	Letting the creator create the object it depends, rather than relying on a third object, reduces coupling. This make software easy to change.
Keep It Simple and Stupid	Presentation layer only deals with presentation the user interface.	No need to use an object, which must know which object has the information to fulfill the request. Thus, use of the expert pattern supports "keep it simple and stupid."	

Figure 10.1: Design principles supported by 3 GRASP patterns

duce to the behavioral design for the Online Car Rental system. Hint: Compare the design with one that does not apply the patterns to identify the improvements.

Solution. The improvements by the three GRASP patterns is described in the chapter. The student's solution only needs to adapt the improvements presented to this car rental application.

10.4 Do the same as in exercise 10.2 but for the Edit Class Diagram use case for a UML Class Diagram Editor.

Solution. Figure 10.2 shows the expanded use case and the sequence diagram.

10.5 Identify and explain the improvements that the controller, expert, and creator patterns introduce to the behavioral design for the UML Class Diagram Editor. Hint: Compare the design with one that does not apply the patterns to identify the improvements.

Solution. The improvements are as presented in the chapter, when the three patterns are described.

UC1. Edit Class Diagram

Actor: User	System: Class Diagram Editor
1) TUCBW user clicks "File" then selects 1.1) "New Diagram" or 1.2) "Open Diagram" 1.2.1) User navigates to appropriate directory and selects the diagram to open. 3) User repeatedly performs following operations: 3.1) Clicking "Class" button and the canvas to add a class. 3.2) Clicking "Inheritance," "Aggregation," or "Association" button to add a relationship. 3.3) Selecting classes or relationships and pressing the Delete key to delete them. 3.4) Selecting classes and relationships, and dragging them to move. 5) TUCEW user sees the updated class diagram.	0) System displays editor GUI. 2) System displays 2.1) a blank diagram, or 2.2) a choose-file dialog, and then 2.2.1) the selected diagram. *4) System accordingly displays the class diagram updated by the editing operation. Step 4 is nontrivial.

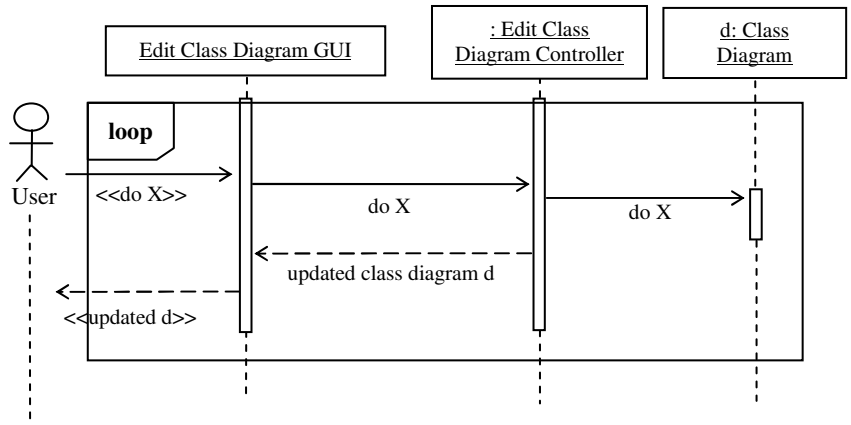


Figure 10.2: Expanded use case and sequence diagram for edit class diagram use case

10.6 Which classes in the UML Class Diagram Editor design could be a singleton? Justify your answer.

Solution. The Edit Class Diagram GUI and Edit Class Diagram Controller classes can be singletons because the application needs only one instance of these classes.

10.7 Write an essay to discuss the similarities and differences of the controller, expert, and creator patterns.

Solution. These three GRASP patterns all solve the responsibility assignment problem. They answer the question as "who should be responsible for" a given responsibility. They differ in the design problem each of them solves. In particular, the patterns solve the following problems, respectively.

1. **Controller:** Who should be responsible for *decoupling the presentation from business objects*? It is an application of the model-view-controller (MVC) pattern.
2. **Expert:** Who should be responsible for *handling a given request*? In other words, which object should be assigned a given operation (or function).
3. **Creator:** Who should be responsible for *creating a given object*?

10.8 Describe three realistic, not dummy or hypothetic, situations to apply the controller, expert, and creator patterns.

Solution. There are many applications. Some of them are the following:

Controller: Banking systems, online retailing systems, email systems that support different types of user interfaces such as web-based, cellphone, and smart devices. The controller pattern uses one use case controller to fulfill requests sent from different types of user interfaces.

Expert: Login, reset password that needs to authenticate the user are excellent applications

of the expert pattern. Without using the expert pattern, not only coupling is increased but security is also compromised.

Creator: Diagram editor, modeling tools such as the OIM tool described in one of the exercises of Chapter 9 are good applications of the creator pattern. In these applications, the aggregate object such as the diagram object may create its components.

10.9 In exercise 8.6, you produce the expanded use cases for a state diagram editor. In this exercise, you are required to identify the nontrivial steps. For each nontrivial step, write a scenario, construct a scenario table, and convert the scenario table to a design sequence diagram. Apply the controller, expert, creator, and singleton pattern if appropriate during this process. Preferably, the patterns should be applied when you describe the scenarios.

Solution. The Edit State Diagram sequence diagram, that is, the solution to Exercise 9.7, has applied the controller pattern.

Chapter 11

Deriving a Design Class Diagram

11.1 Derive a DCD from the design sequence diagram in Figure 9.21.

Solution. The DCD is shown in Figure 11.1 of this manual.

11.2 Derive a DCD from the design sequence diagram you produced for the Order a Dish use case in the exercises of Chapter 9. If you have not produced the sequence diagram, do the relevant exercises in Chapter 9 and then produce the DCD.

Solution. First, the informal messages between the objects in Figure 9.2 of this manual are replaced with function calls. From the resulting sequence diagram, the DCD is derived as

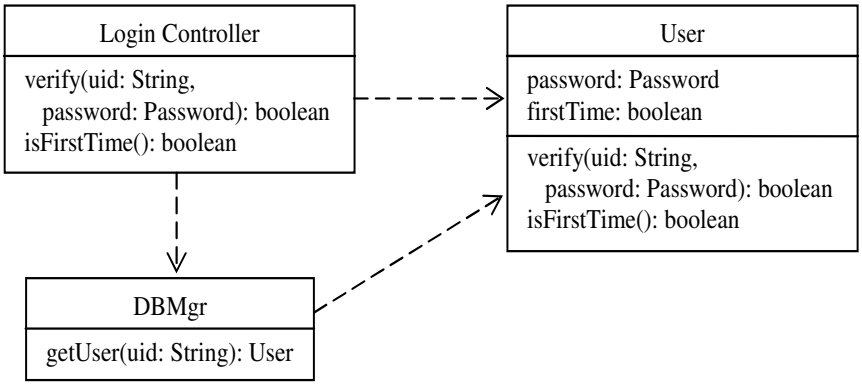


Figure 11.1: DCD derived from Figure 9.21

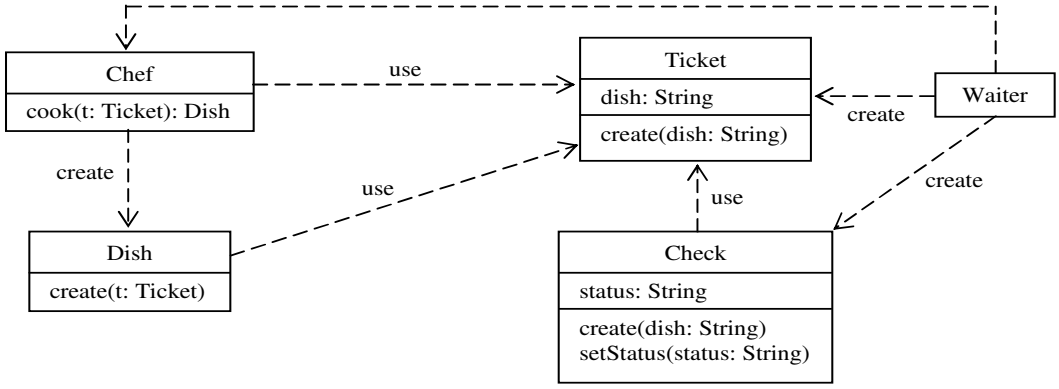


Figure 11.2: DCD for order dish sequence diagram

shown in Figure 11.2.

11.3 Derive a DCD from the design sequence diagrams you produced for the state diagram editor in exercise 9.7.

Solution. The sequence diagram shown in Figure 9.13 of this manual is an informal sequence diagram. Moreover, the "do X" message represents a generic message, which can mean any of the state diagram editing operations. The steps described in this chapter (Chapter 11) can still be applied except that the "do X" message needs to be substituted with all possible editing operations. After the substitutions, the classes and attributes are derived using the derivation rules described in the chapter. The following paragraph illustrate these.

First, the classes are derived from the sequence diagram: Edit State Diagram GUI, Edit State Diagram Controller, and State Diagram. Two of these have "do X" as the label of an incoming edge. The label needs to be replaced by all editing operations applicable to the class. The parameters and return types are also specified, see Figure 11.3 of this manual. From the parameters and return types, other classes as well attributes are recursively derived. For example, from the add and get operations, one can derive two classes: State, and Tran. The resulting DCD is shown in Figure 11.3 of this manual.

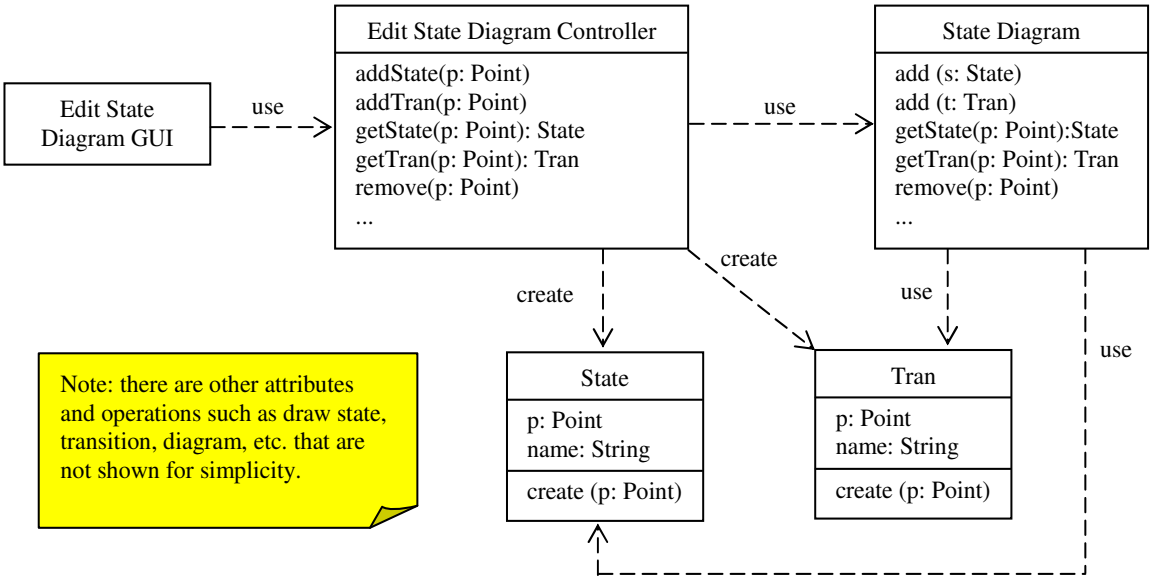


Figure 11.3: DCD derived from edit state diagram sequence diagram

11.4 Derive a DCD from the design sequence diagrams that you produced for the ATM application in exercise 9.5.

Solution. Figure 11.4 of this manual shows the DCD derived from the withdraw money sequence diagram.

11.5 Derive a DCD from the design sequence diagrams that you produced for the car rental application Exercise 9.6 in Chapter 9.

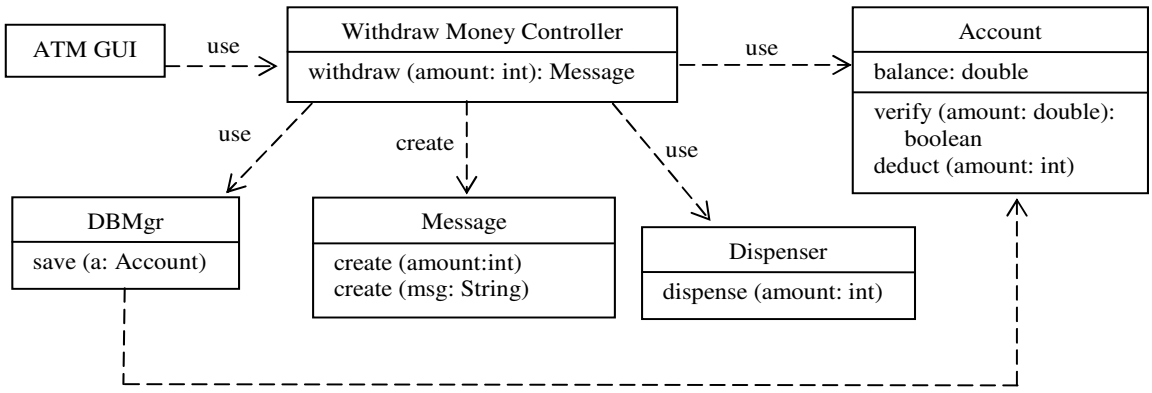


Figure 11.4: DCD derived from withdraw money sequence diagram

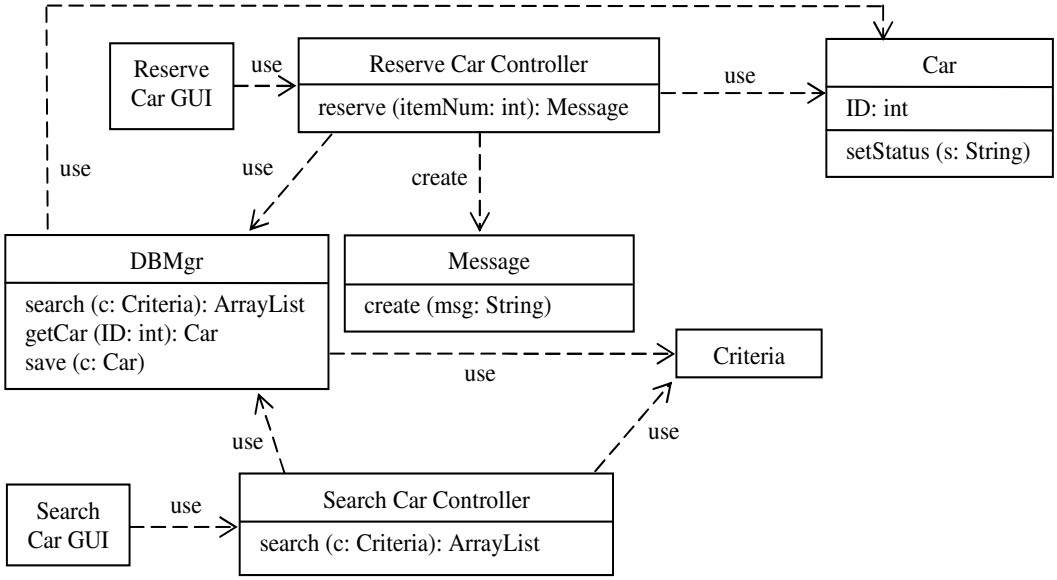


Figure 11.5: DCD for two use cases of car rental system

Solution. Figure 11.5 shows the DCD for the Search for Car and Reserve Car sequence diagrams shown in Figures 9.9 and 9.11.

Chapter 12

User Interface Design

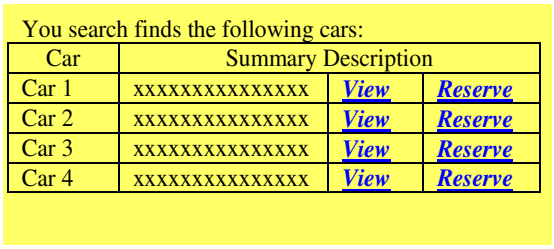
For each of the following use cases, first produce an expanded use case description if you have not done so in previous chapters. Then perform the user interface design steps to produce a user interface design for the use case.

12.1 The Reserve a Car use case for an online car rental application. A description of such an application is presented in Appendix D.1.

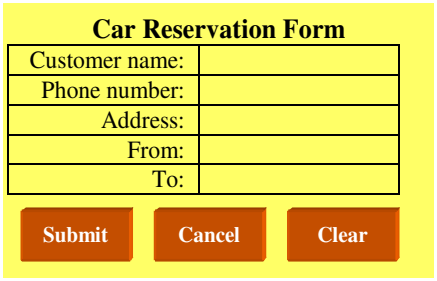
Solution. The expanded use case for Reserve a Car use case is shown in Figure 9.10 of this solution manual. From the expanded use case, one can identify three major system displays: (1) display of a list of cars as search result, that is, cars that satisfy the search criteria, (2) a car reservation form, and (3) a reservation confirmation message. The results for the first three steps of the UI design is shown in Figure 12.1.

12.2 The Login use case. When the user logs in the first time, the system shall direct the user to a page to define authentication questions to be used when the user wants to reset the password in the future. Assume that each user needs to define five questions and provide the answers to these questions.

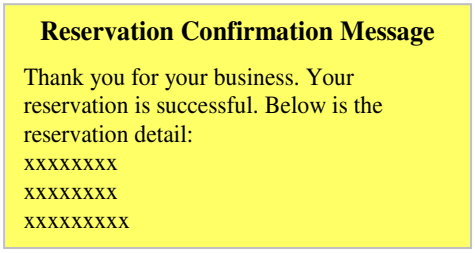
Expanded Use Case Step	System Display	Information Displayed	User Input	User Actions
0)	Search result page.	List of cars resulting from search car use case.		
2)	Car reservation form.	Attributes of Reservation class in the car rental system domain model (see Figure 5.20 in the textbook)		Clicking the Reserve button next to the car to be reserved.
4)	Confirmation message.	Reservation successful, or car is not available.	Reservation information.	Fill reservation form and clicks the Submit button.



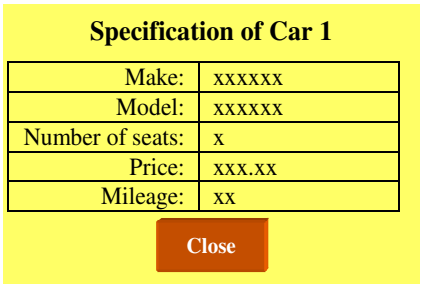
(0) Search result page



(2) Reservation form



(4) Confirmation message



View detail of a car

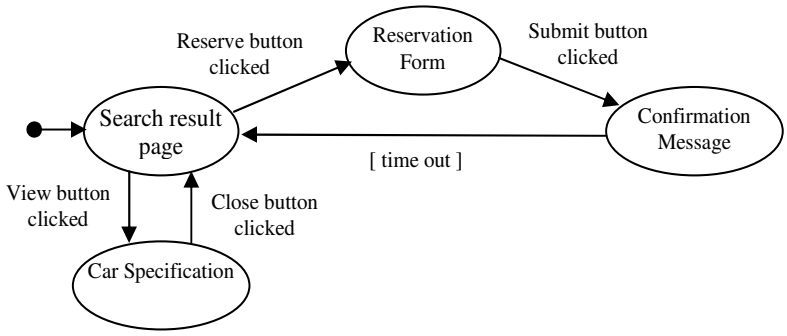


Figure 12.1: User interface design for reserve car use case

Solution. Figure 12.2 shows the UI design for the login use case.

12.3 The Checkout Books and Return Books use cases of an online library system that allows the patrons to check out and return books. The precondition for these two use cases are that the patron has logged into the system. The books that are checked out are mailed to the patron. The patron mails the returned books through the mail.

Solution. The UI design for these two use cases are similar to the UI design for the reserve car use case (see solution to Exercise 12.1).

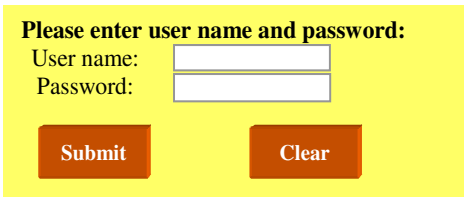
12.4 In Appendix D.4, a class diagram editor application is described. Perform a user interface design for this application.

Solution. The solution to this exercise is similar to the UI design for the state diagram editor presented in Chapter 12 of the book.

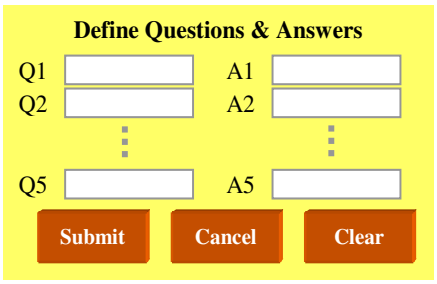
12.5 Check the design of the above user interfaces using the user interface design review checklist presented in Section 12.4.7. Document any problems detected. Modify the design to fix the problems.

Solution. Omitted.

Expanded Use Case Step	System Display	Information Displayed	User Input	User Actions
0)	Login page	Login and password fields.		
2)	Define authentication questions & answers page.	A form with fields for entering five authentication questions and corresponding answers.	user name and password	User enters user name and password and clicks the Login button. Need to define Q&A only for first time login.
4)	Home page	Welcome message etc.	Authentication information.	Fills questions and answers, and clicks the Submit button.



(0) Login page



(2) Define Q&A form

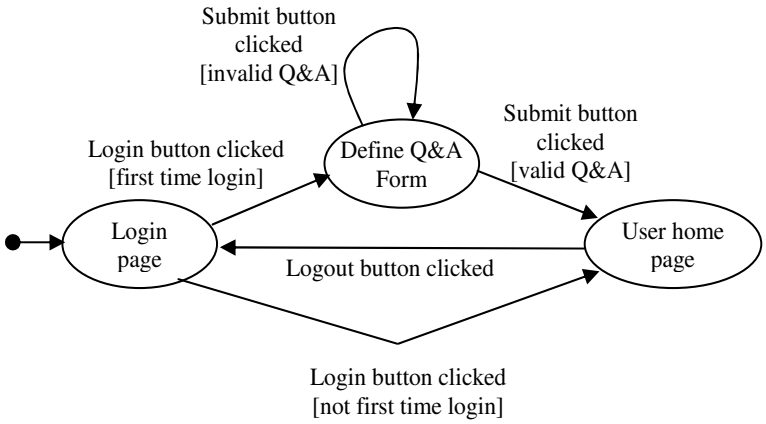


Figure 12.2: UI design for the login use case

Part IV

MODELING AND DESIGN OF OTHER TYPES OF SYSTEMS

Chapter 13

Object State Modeling

13.1 Refine the cruise control state diagram in Figure 13.12 with lower-level state diagrams. Check for consistency between the external events and the triggering events of the transitions as well as the response actions and messages to the external entities.

Solution. The Cruising state is refined here. Figure 13.1 of this manual shows the domain model for the Cruising state. Three stimuli come into this state. They cause the cruise control to leave the Cruising state. One instruction reads the wheel sensor and one instruction controls the gas throttle. The Cruising state has two lower-level states: Coasting, and Accelerating. If

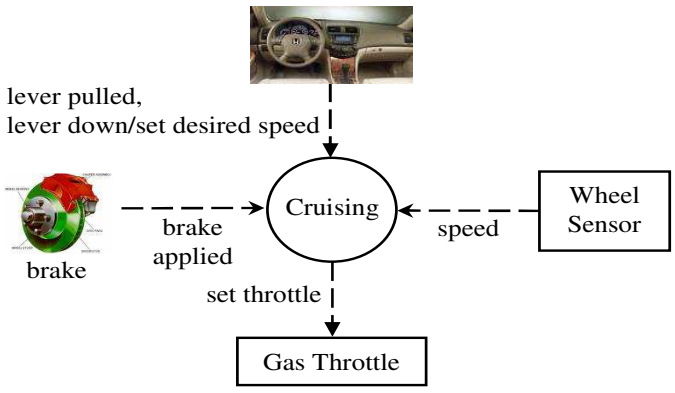


Figure 13.1: Cruising state domain model

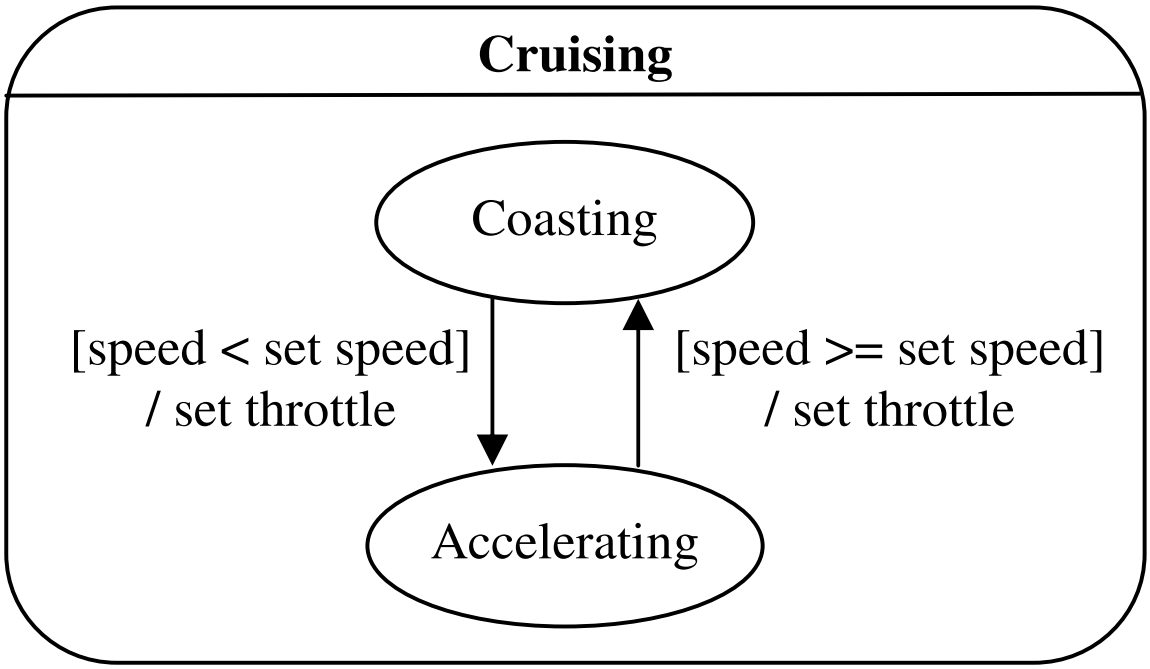


Figure 13.2: Refinement of the Cruising state

the wheel sensor indicates that the set speed is reached, the cruise control enters the Coasting state. If the wheel sensor shows that the speed is lower than the set speed, the cruise control instructs the gas throttle to accelerate and enters the Accelerating state. Figure 13.2 shows the state diagram.

13.2 Construct a domain model to explain a UML state diagram that includes CSS and CCS.

Solution. Figure 13.3 shows the domain model.

13.3 Apply the steps presented in this chapter to the railroad crossing system (RCS) described as the following:

An RCS includes a central control that controls the traffic light, the bell, and the gate at the crossing. When a train approaches the crossing from either direction at a certain distance, a sensor device senses the train's arrival and communicates this to the central control. The central control informs the traffic light control, bell control, and gate control. After receiving

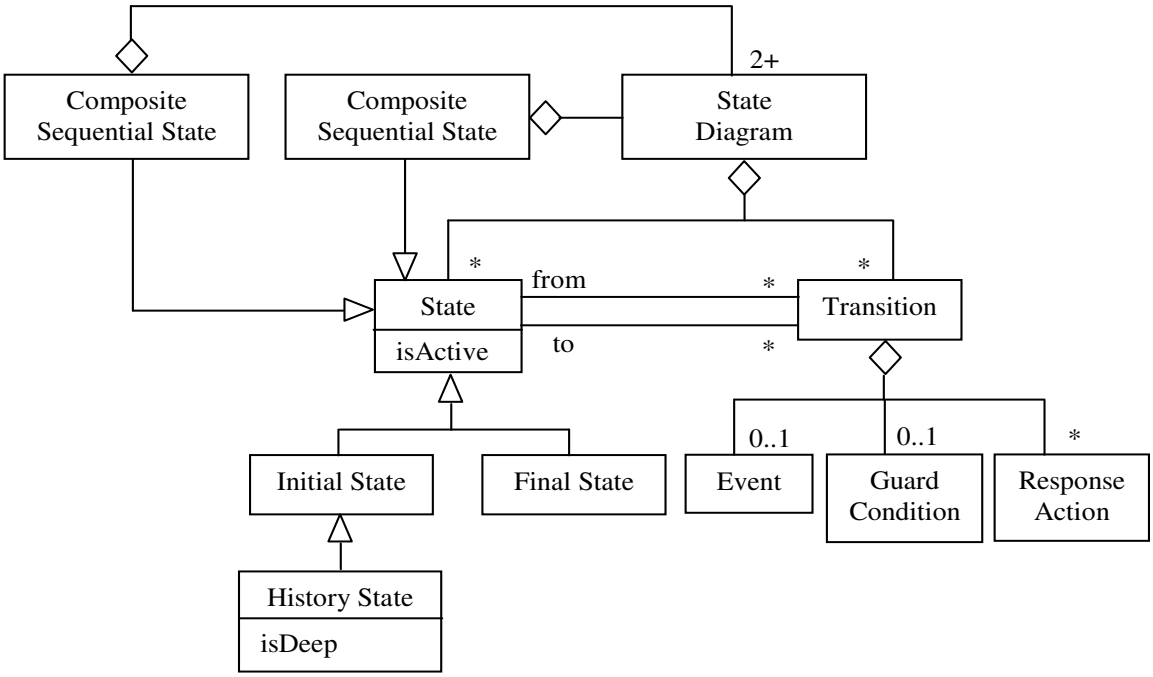
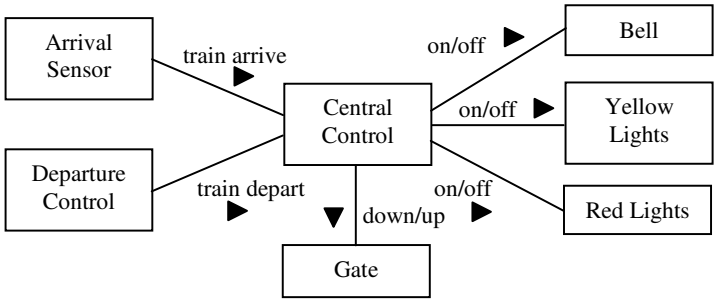


Figure 13.3: State diagram domain model

the signal from the central control, the traffic light control turns on the flashing yellow warning light for a given amount of time, which must be long enough to allowthe traffic to stop before the crossing. It then changes the light from flashing yellow to flash red. The bell control turns on the bell. After the red light begins to flash, the gate begins to close. During this process, the train continues its course with its own speed within the specified speed limit. After the train has completely passed the crossing, another sensor device at the other end of the railroad senses the train's departure and communicates this event to the central control. The central control instructs the gate control to lift up the gates, the traffic light control to turn off the light, and the bell control to turn off the bell.

Solution. Figure 13.4 shows the domain model and state transition table. Converting the table to state diagram is trivial and omitted.

13.4 Construct a state diagram for a typical digital watch, described as follows: A digital watch



	train arrived	train departed	timer off
Idle	Train Arrival / start timer; send on and down signals to devices.		
Train Arrival			Train Passing / stop timer; send change yellow to red flashing light signal.
Train Passing		Train Leaving / start timer.	
Train Leaving			Idle / stop timer; send off and up signals to devices.

Figure 13.4: Railroad crossing domain model and transition table

has four modes: the display mode, the set-alarm mode, the stopwatch mode, and the set-time-and-date mode. It has three buttons: a mode button to go to the different modes, a start-stop button for advancing values in the different modes, and a light button for selection, lighting, and stopping the buzzer. In the display mode, press the mode button once to go to the set-alarm mode, twice to go to the stopwatch mode, three times to go to the settime-and-date mode, and four times to go back to the display mode. In each of the above modes, use the light button to step through the different values that can be set. For example, set hour, then set minute, and so on. Use the start-stop button to increment the values. Use the mode button to exit and return to the display mode.

Solution. Figure 13.5 shows one solution.

13.5 Produce a mapping between the cruise control state diagram in Figure 13.12 and the state pattern in Figure 13.17. An example mapping is shown in Figure 13.16.

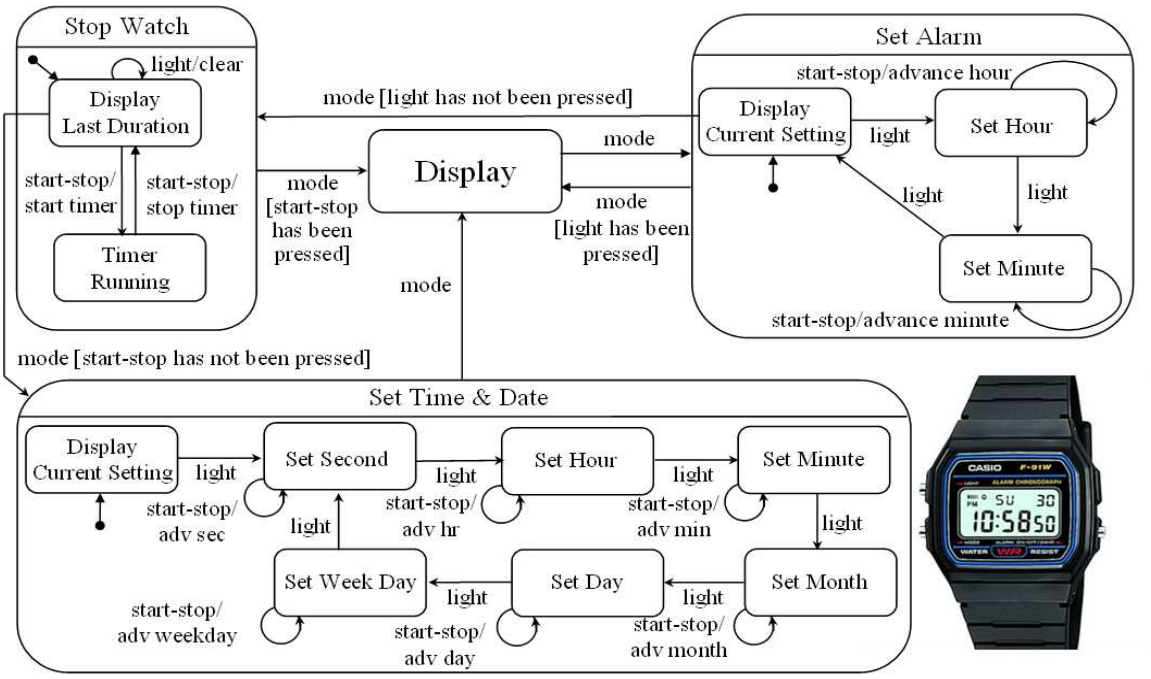


Figure 13.5: State diagram for a digital watch

Solution. The mapping is straightforward because the state names and transition names are in one-to-one correspondence. The exceptions are the State class and Cruise Control class. The former maps to the concept of a state in the state diagram, and the latter maps to the subject, which is the cruise control.

13.6 A simple lawn mower agent is capable of mowing a rectangular shape area. The lawn is conceptually viewed as consisting of 2 foot by 2 foot squares, fitting the size of the solar-powered mower. Actions of the agent include:

1. Move forward: the mower moves forward one square.
2. Turn left: the mower makes a left turn.
3. Turn right: the mower makes a right turn.
4. Cut (grass): the mower cuts the grass in the current square.
5. Percept: the agent perceives the environment.

6. Turn on-off: the agent turns on-off the power of the mower.

The garden landscape consists of lawns, trees, shrubs, buildings, pools, and more. The landscape may be represented by an m x n array of integers:

- A[i,j] = 10 means that an obstacle is in the square.
- A[i,j] = 9 means that there is a trap, such as a pool.
- A[i,j] = 2 means that there is grass in the square that needs to be cut.
- A[i,j] = 1 means that the grass is already cut.
- A[i,j] = 0 means that the square is pavement and it is safe to cross over it.

At any given time, the agent knows its direction (N, S, E, W) and location (i, j). Initially, the agent is at A[0,0], facing east, and the mower is off. After mowing the lawn, the agent returns to the initial state that is, it is at location A[0,0], facing east, and the mower is off.

The agent maintains a set mowing schedule, for example, mowing once a week during the highgrowth season, biweekly during the rest of the growing season, and not mowing during the rest of the year. The agent wakes up and mows the lawn according to the schedule. Do the following for this exercise:

a. Construct a domain model for the mowing agent application.

b. Construct a state diagram for the mowing agent.

c. Apply the state pattern to this application.

d. Implement the mowing agent state pattern and demonstrate that the software works by printing the traversal of the agent on a 20 square by 20 square area.

Solution. This exercise is difficult. The purpose is to motivate students' interest with a realistic application. A good enough solution, judged by the instructor, is enough. Perhaps,

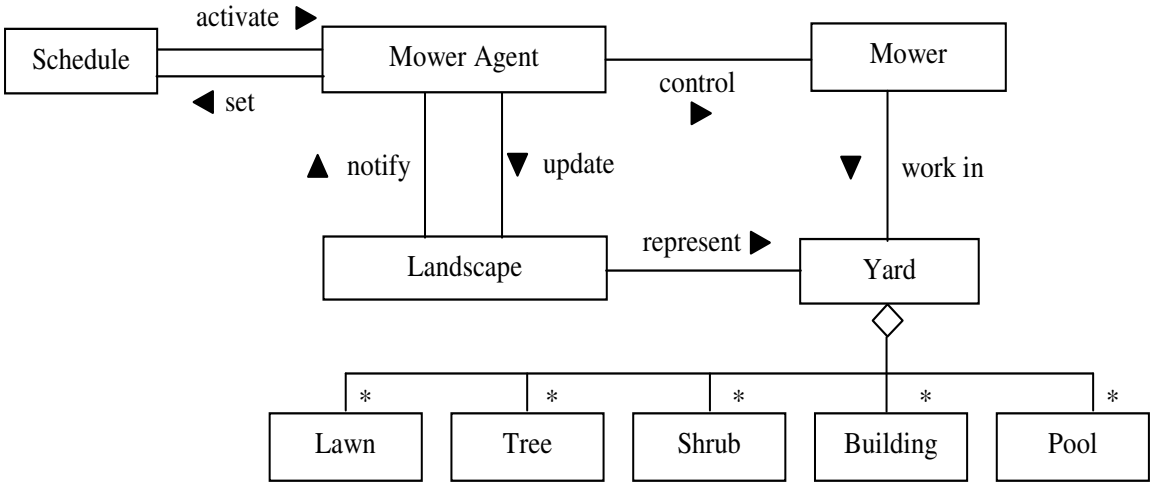


Figure 13.6: Lawn mower domain model

equal weights could be given to the underneath algorithm (i.e., how the agent cut the lawn), the state diagram, and the state pattern.

Figures 13.6 shows the domain model. Figure 13.7 shows the state diagram, where the transitions are explained using three decision tables in Figure 13.8. Each decision table represents a state of the mower. That is, the mower is mowing toward east, north, or west. The state pattern is shown in Figure 13.9.

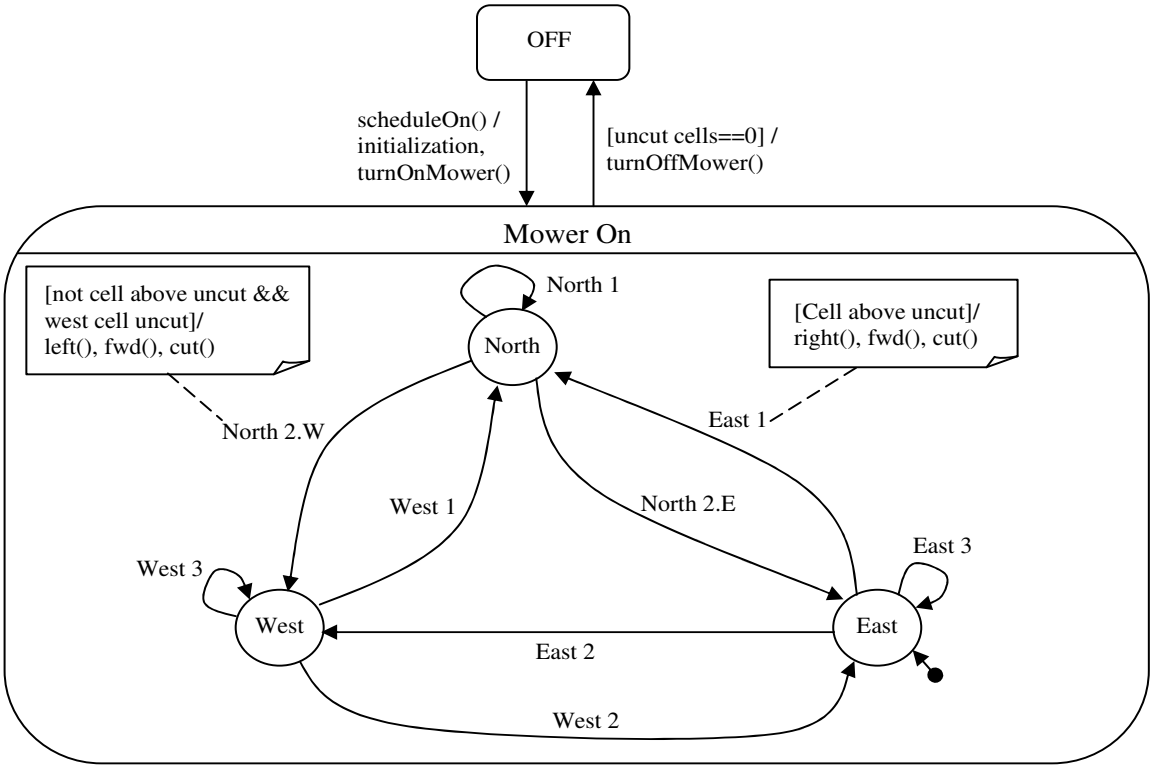


Figure 13.7: Mower agent state diagram

East	1	2	3	4
Cell above uncut?	Y	N	N	N
Obstacle/cut cell ahead?	-	Y	N	Y
Obstacle below?	-	N	-	Y
Rule count	4	1	2	1
Turn	L	R		
Move fwd & cut	X	X	X	
Turn		R		
Go to nearest uncut cell				X
Next state	N	W	E	

North	1	2	3
Cell ahead uncut?	Y	N	N
Is E/W cell uncut?	-	Y	N
Rule count	2	1	1
Turn		R/L	
Move fwd & cut	X	X	
Go to nearest uncut cell			X
Next state	N	E/W	

West	1	2	3	4
Cell above uncut?	Y	N	N	N
Obstacle/cut cell ahead?	-	Y	N	Y
Obstacle below?	-	N	-	Y
Rule count	4	1	2	1
Turn	R	L		
Move fwd & cut	X	X	X	
Turn		L		
Go to nearest uncut cell				X
Next state	N	E	W	

Action	Meaning
Turn L/R	Turn left/right.
Move fwd & cut	Move forward, update (x, y), cut grass if needed, update status.
Go to nearest uncut cell	Go to the nearest uncut cell, update direction & (x, y).
Next state N/E/W	Return North, East, or West as next state.

Figure 13.8: Decision table describing state transitions

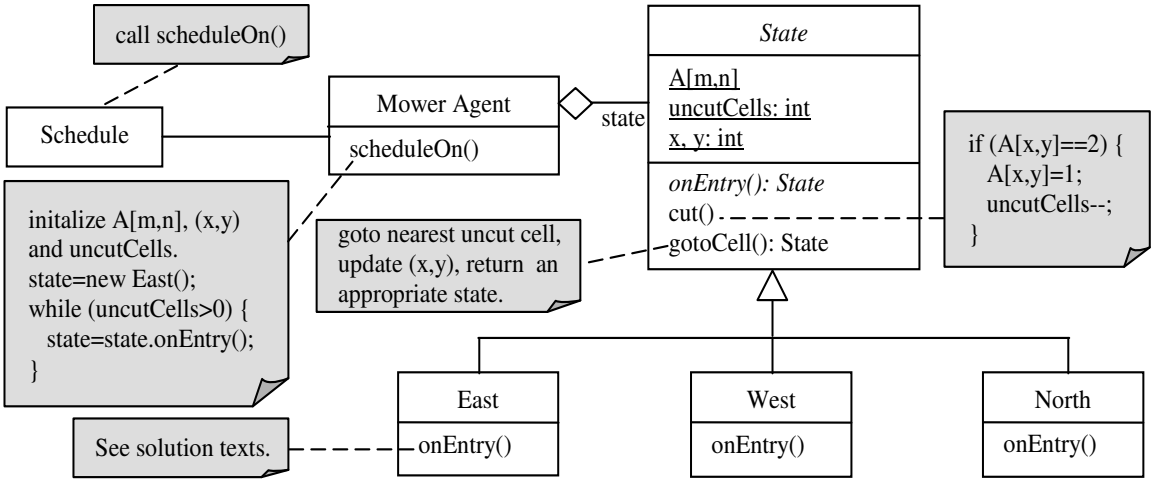


Figure 13.9: Lawn mower state pattern

Below is the implementation of the onEntry() method of the East state. The onEntry() method for the other two states can be implemented similarly.

```
if (A[x,y-1]==2) {
   turn left; y--; cut();
   return new North();
}
if (A[x+1,y]==1 || A[x+1,y]>2) {
   if (A[x,y+1]>2) return gotoCell();
   else {
      turn right; y++; cut(); turn right;
      return new West();
   }
} else {
   x++; cut();
   return this;
}
```

13.7 A pizza vending machine ...

Solution. Omitted.

13.8 Extend the pizza vending machine ...

Solution. Omitted.

Chapter 14

Activity Modeling for Transformational Systems

14.1 Refine the Obtain an Offer Letter activity discussed in Section 14.5.4 to produce an activity diagram, taking into consideration conditional branching, concurrency, and synchronization.

Solution. Figure 14.1 shows a sample solution.

14.2 Develop a description for each of the following applications and apply the activity modeling steps to each of them:

a. An online build-to-order application.

b. A claim-handling system for an insurance company that issues home insurance as well as automobile insurance.

Solution. Omitted.

14.3 Produce an activity diagram that describes the steps for activity modeling.

Solution. Figure 14.2 shows a sample activity diagram for this exercise.

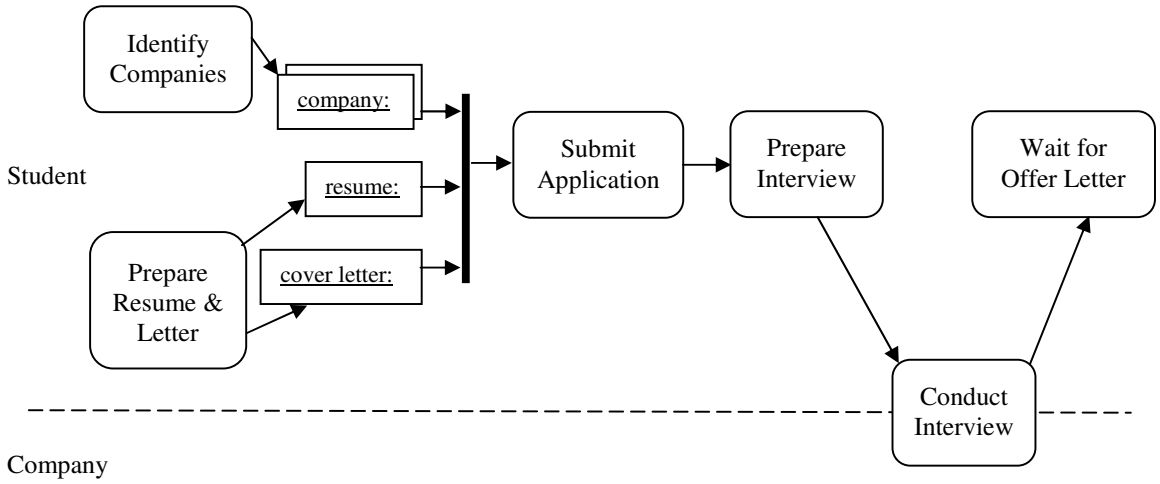


Figure 14.1: Obtain an offer letter activity diagram

14.4 Produce an activity diagram that describes your academic department's undergraduate or graduate advising.

Solution. Omitted.

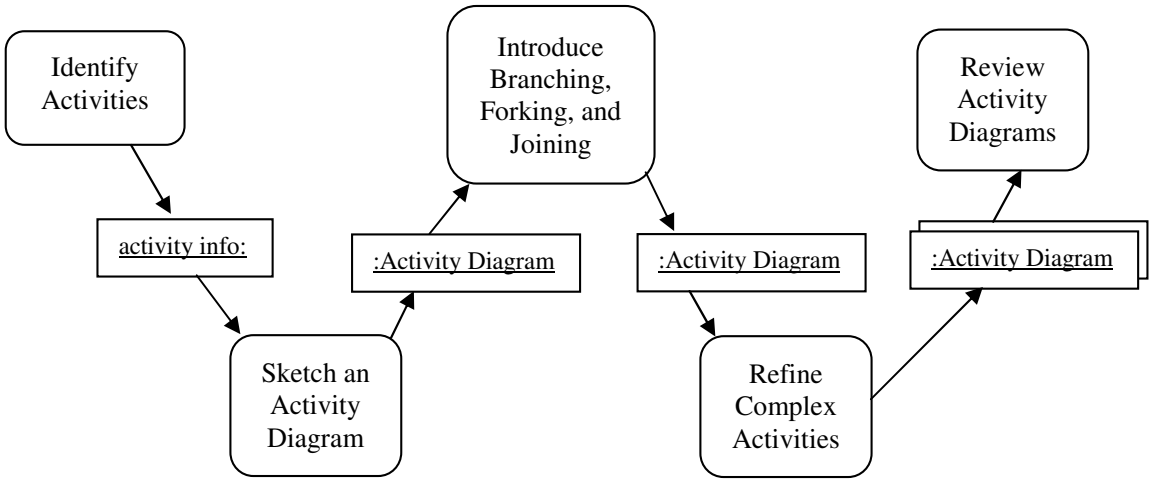


Figure 14.2: Activity diagram for activity modeling

Chapter 15

Modeling and Design of Rule-Based Systems

15.1 Perform the following for the business rules of the point-of-sale shipping software presented at the beginning of this chapter:

a. Construct a decision table using the systematic decision table construction method.

b. Consolidate the decision table and check the completeness, consistency, and non-redundancy of the result. Correct any errors, if any.

c. Generate code from the consolidated, error-free decision table.

Solution. Figure 15.1 shows the decision table produced using the systematic decision table construction method. The consolidated decision table is the same as the one shown in Figure 15.1 in the textbook. To generate code, the consolidated decision table is reorganized as shown in Figure 15.2(a). The pseudo-code generated is shown in Figure 15.2(b).

15.2 Do the following for the progressive decision table construction algorithm presented in this chapter:

	1	2	3	4	5	6	7	8	9
Weight≤150 lbs	Y	Y	Y	Y	Y	Y	Y	Y	Y
Length (inches)	≤108	≤108	≤108	>180&≤119	>180&≤119	>180&≤119	>119	>119	>119
Length plus girth (inches)	≤130	>130&≤165	>165	≤130	>130&≤165	>165	≤130	>130&≤165	>165
Rule Count	1	1	1	1	1	1	1	1	1
Rate by weight	X								
Rate by greater of weight and dim weight		X		X	X				
Add $45 surcharge to each package		X		X	X				
Reject package			X			X	X	X	X

	10	11	12	13	14	15	16	17	18
Weight≤150 lbs	N	N	N	N	N	N	N	N	N
Length (inches)	≤108	≤108	≤108	>180&≤119	>180&≤119	>180&≤119	>119	>119	>119
Length plus girth (inches)	≤130	>130&≤165	>165	≤130	>130&≤165	>165	≤130	>130&≤165	>165
Rule Count	1	1	1	1	1	1	1	1	1
Rate by weight									
Rate by greater of weight and dim weight									
Add $45 surcharge to each package									
Reject package	X	X	X	X	X	X	X	X	X

Figure 15.1: A systematic decision table

	1	3	2	5	4	6
Weight ≤150 lbs	Y	Y	Y	Y	Y	N
Length (inches)	≤108	≤108	>108 &≤119	≤119	>119	–
Length plus Girth (inches)	≤130	>130 &≤165	≤165	>165	–	–
Rule Count	1	1	2	2	3	9
Rate by Weight	X					
Rate by greater of Weight and Dim. Weight		X	X			
Add $45 surcharge to each package		X	X			
Reject package				X	X	X

(a) reorganizing the consolidated decision table

```
if (weight <= 150)
  if (length <= 108)
    if (length + girth <=130)
      rate by weight;
    else
      if (length + girth <= 165) {
        rate by greater of weight and dim.
          weight;
        add $45 surcharge to each package;
      } else
        reject package;
  else
    if (length <= 119)
      if (length + girth <= 165) {
        rate by greater of weight and dim.
          weight;
        add $45 surcharge to each package;
      } else
        reject package;
else
  reject package;
```

(b) pseudo-code generated

Figure 15.2: Generating code from a decision table

	1	2	3	4	5	6
Has next condition?	Y	N	---	---	Y	Y
Does condition have next value?	N	N	Y	N	Y	Y
Does condition combination lead to an action?	N	N	N	N	Y	N
Rule Count	1	1	2	2	1	1
Enter next condition, make it the current condition	X					
Enter next condition value			X			
Enter "---" in remaining condition entries					X	
Check appropriate actions					X	
Begin a new rule by copying condition values of current rule					X	
Select a condition with unconsidered values, make it the current condition						X
Continue with rule	3		5	1	3	3
Halt		X				

Figure 15.3: Decision table for progressive decision table construction

a. Construct a decision table for the progressive decision table construction algorithm using the progressive decision table construction method.

b. Check the completeness, consistency, and non-redundancy of the decision table. Correct any errors, if any.

c. Consolidate the error-free decision table and check the completeness, consistency, and non-redundancy of the resulting decision table.

Solution. Figure 15.3 shows the decision table. It is complete, consistent, and non-redundant.

15.3 Write an algorithm for checking the completeness of a decision table. The algorithm must also calculate the rule counts from the indifferences.

Solution. There are many different ways to write the algorithm. One easy way is to loop over the rules, and count the number of indifferences in each rule. If the decision table is a limited entry decision table, then the rule count for a rule is 2^n, where n is the number of indifferences. A sum is updated as the rule count of each rule is computed. The table is complete if the sum is equal to 2^N, where N is the number of conditions of the decision table.

15.4 Convert your decision table completeness-checking algorithm into a decision table. Check the

completeness, consistency, and non-redundancy of the decision table. Correct any errors.

Solution. Omitted.

15.5 Construct a decision table from the admission criteria described below. In addition, check the completeness, consistency, and non-redundancy of the decision table.

The computer science (CS) department of a university uses a set of admission criteria to process applications to its graduate program. Applications that clearly satisfy all the admission criteria are classified as accept. Applications that do not satisfy, or barely satisfy, the criteria are classified as reject. The remaining applications are marked as "pending investigation." Applicants must have paid the application fee. The admission criteria are the following:

a. A four-year undergraduate degree in a technical area.

b. A 3.2 grade point average (GPA) or higher on a 4.0 scale on the last two years of undergraduate course work.

c. Relevance of the student's degree (background) to the CSE curriculum. The ranking is high, average, and low.

d. Ranking of the undergraduate degree-granting institution. The ranking is high, average, and low.

e. A sum of verbal and quantitative scores of 1,150 or higher on the GRE and

i. GRE quantitative score ≥ 700

ii. GRE verbal score ≥ 400

f. International applicants must have a test of English as a Foreign Language (TOEFL) score of ≥ 90 on the iBT, or a score ≥ 7.0 on the International English Language Testing System (IELTS).

	1	2	3	4	5	6	7
Is fee paid?	Y	N	Y	Y	Y	Y	Y
Relevance to CSE	H	-	-	H	A	L	-
Institution rank	H	-	-	A	H	-	L
All-other conditions satisfied	Y	-	N	Y	Y	Y	Y
Rule count	1	18	9	1	1	3	3
Accept	X						
Reject			X			X	X
Pending investigation				X	X		
Wait for fee paid		X					

All-other: a conjunction of all other admission conditions.

Figure 15.4: Admission decision table

```
rule-set ::= rule [rule]
rule ::= condition-list '==>' action ';'
condition-list ::= condition [‘&’ condition-list]
condition ::= condition-name ‘=’ condition-value
action ::= ‘accept’ | ‘reject’ | ‘pending’ | ‘wait’
condition-name ::= ‘fee-paid’ | ‘relevance’ |
                   ‘inst-rank’ | ‘all-other’
condition-value ::= ‘Y’ | ‘N’ | ‘H’ | ‘A’ | ‘L’
```

Legend:
::= defined as
‘abc’ literal abc
[abc] abc is optional
a | b selectively a or b

Figure 15.5: Admission decision rules grammar

Solution. Figure 15.4 shows the decision table. It is complete, consistent, and non-redundant.

15.6 Produce a grammar for the business rules you produce in exercise 15.5.

Solution. Figure 15.5 shows a grammar for the admission decision table.

15.7 Construct a class diagram to represent the business rule grammar you produced in exercise 15.6. Show with UML notes how each of the classes implements its functions to interpret the semantics of the class.

Solution. Figure 15.6 shows a UML class diagram. It is assumed that before evaluating an application (app), the relevant attribute values are stored in a hash table.

15.8 Design a rule-based framework that allows the rules to change on the fly. That is, the rules can be stored in a text file and loaded into the memory when the system starts. The user can modify the rules by changing the text file. After modification, the user can reload the rules

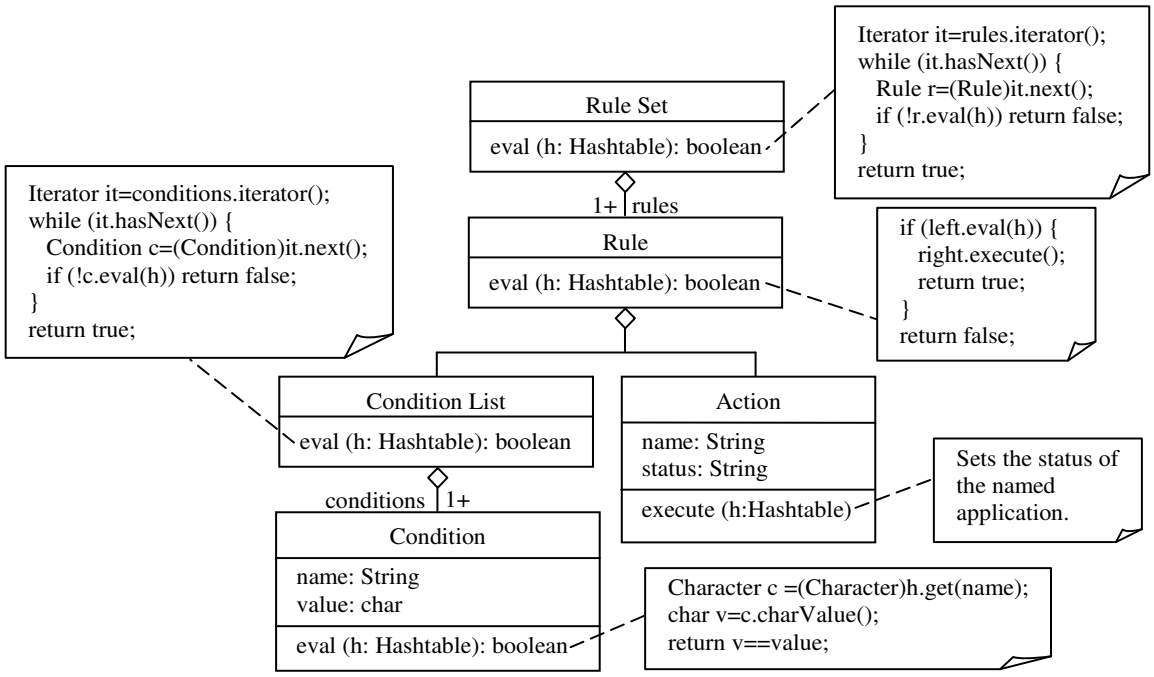


Figure 15.6: UML class diagram for the admission grammar

without having to restart the system. Produce a design class diagram to show the rule-based framework. Indicate the design patterns applied. Implement and demonstrate the rule-based framework.

Solution. There are many possible solutions. One is described here. The framework consists of two main parts: (1) representation of rules, and (2) a graphical user interface (GUI) for entering and updating rules. Figure 15.8 of the textbook defines the grammar for representing a business rule. The code for implementing the classes is also given in the figure. The implementation is easy and straightforward. A rule set is simply a collection of rules. To be able to save the rules, the rule set must implement the Java serializable interface. Solutions to previous exercises illustrate how these can be accomplished.

The GUI provides entries for specifying the conditions and actions of each rule. It then adds the rule to the rule set. At the end, it saves the rule set using Java ObjectOutputStream API.

The rules are then loaded and stored in a collection. The solution to the previous exercise illustrates how to use an iterator to evaluate the rules.

A hash table is used to store the objects, variables, and constants to be used as the context to evaluate the rules. Each leaf node retrieves the objects, variable, or constants from the hash table and use them to evaluate the leaf node and return the result to the parent node. The solution to the previous exercise shows how to accomplish this.

Part V

APPLYING SITUATION-SPECIFIC PATTERNS

Chapter 16

Applying Patterns to Design a State Diagram Editor

16.1 Produce the skeleton code for the design in Figure 16.11 from the skeleton code for the strategy pattern. Hint: Substitute the classes and operations in the existing design for the classes and operations in the strategy pattern.

Solution. This is easy and straightforward. It is omitted.

16.2 What are the advantages of applying the visitor pattern to the state diagram editor as shown in Figure 16.15?

Solution. The advantages are the benefits of the visitor pattern:

- High cohesion is achieved because each concrete checker performs only one type of checking.
- The concrete checker classes may be reused in other application.
- It is easy to add concrete checker classes. The new checker class needs to implement only the polymorphic check(e: Diagram Element) method for each diagram element class.

- It is easy to add a concrete Diagram Element class.

16.3 Discuss the similarities and differences of the two patterns in each of the following pairs of patterns. Moreover, give two situations that are not in the textbook or the lecture notes in which each pattern can be exclusively applied to exactly one situation. Describe how each pattern is applied and why the other is not applicable.

a. Decorator and visitor

b. Memento and state

c. Abstract factory and builder

d. Strategy and visitor

e. Flyweight and singleton

f. Observer and controller

g. Abstract factory and creator

Solution.

a. Decorator and visitor: Figure 16.1 summarizes the similarities, differences and exclusive applications.

b. Memento and state: The only similarity of these two patterns is that they are both behavioral patterns. The other aspects of these two patterns are very different. This also makes it easy to find exclusive applications for each of the patterns.

c. Abstract factory and builder: See the summary in Figure 16.2.

d. Strategy and visitor: see Figure 16.3.

e. Flyweight and singleton: Figure 16.4 shows a comparison of these two patterns.

f. Observer and controller: see Figure 16.5.

g. Abstract factory and creator: Figure 16.6 shows the differences and similarities.

Decorator	Visitor
Similarities	
It adds functionality to classes of a structure.	It adds functionality to classes of a structure.
It has a decorator hierarchy.	It has a visitor hierarchy.
Decorators can be added and removed freely.	Visitors can be added and removed freely.
It supports design for change, separation of concerns, high cohesion, low coupling, and designing "stupid objects."	It supports design for change, separation of concerns, high cohesion, low coupling, and designing "stupid objects."
Differences	
A concrete decorator adds the same functionality to different classes.	A concrete visitor adds different functionalities to different classes.
It is a structural pattern – it adds or removes a decorator object and relationship with it.	It is a behavioral pattern – it adds the functionality by executing a function on the objects visited.
It uses single dispatch	It uses double dispatch
It does not need to change class structure interface.	It needs to change the interface of the class structure – add accept（v: Visitor).
Decorator does not use an iterator.	Visitor usually uses an iterator to visit each element of a structure.
Exclusive Applications	
Special offers can be realized by decorators, which can be added to, or removed from, a retail item.	Visitors can compute various sales information including average sales per unit, and sales increases for items with and without a special offer.

Figure 16.1: Comparing decorator and visitor

Abstract Factory	Builder
Similarities	
It is a creational pattern – it produces products.	
It supports design for change, separation of concerns, information hiding, high cohesion, low coupling, and designing "stupid objects."	
It is easy to add concrete factories.	It is easy to add concrete supervisors or concrete builders.
Differences	
It differs from builder in the design problem it solves – it provides the flexibility to change the family of products and hides this from the client.	It differs from abstract factory in the design problem solved – it provides the flexibility to change the process and steps of the process.
It may use factory method to produce families of products.	It is a generalization of factory method, which can vary the steps but not the process.
Exclusive Applications	
Abstract factory can be used to provide tools for different programming languages. By changing the concrete factory, tools for different programming languages can be obtained. For example, JUnit for Java, CppUnit for C++, etc.	A software testing environment that support different testing processes and the steps of the processes use different testing tools. For example, processes include conventional test process, test driven development, regression testing.

Figure 16.2: Comparing abstract factory and builder

Strategy	Visitor
Similarities	
It adds functionality to an existing object.	It adds functionality to classes of a structure.
It has a strategy hierarchy	It has a visitor hierarchy.
It may break encapsulation.	It may break encapsulation.
Both support design for change, separation of concerns, high cohesion, and designing "stupid objects." For example, it is easy to add concrete strategies and concrete visitors.	
Differences	
Concrete strategies have same functionality but differ in other aspects such as quality and performance. They are not type dependent.	Concrete visitors have different functionalities and they are type dependent.
A concrete strategy will perform the same functionality to different objects.	A concrete visitor adds different functionalities to different classes.
It uses single dispatch	It uses double dispatch
It does not need to change class structure interface.	It needs to change the interface of the class structure – add accept（v：Visitor).
It does not need an iterator because it applies to only one object.	It uses an iterator or the like to visit elements of a class structure.
The client must know the strategies and how to choose them.	The client may or may not know which visitor to use or being used.
Exclusive Applications	
Discount policies for different seasons can be implemented as strategies.	Visitors can compute various sales information average sales per unit time and sales increase, and so on for items with and without a special offer.

Figure 16.3: Strategy and visitor

Flyweight	Singleton
Similarities	
Both concern the creation of objects for special situation, although flyweight was classified as a structural pattern.	
Both use shared objects – flyweights are shared objects, and a singleton itself is shared by components.	
The shared objects may be updated, and may cause unexpected effect.	
Differences	
The design problem is how to "create" numerous objects that share the same intrinsic representation.	The design problem is how to create a limited number of globally accessible objects of a class.
It uses object references to a shared flyweight object to "create" numerous objects as needed.	It uses a private constructor and a static getInstance() method to control creation of limited number of instances.
Exclusive Applications	
Game development in which numerous occurrences of an image object need to be created and manipulated.	A database manager for a single-user software system.

Figure 16.4: Flyweight and singleton compared

Controller	Observer
Similarities	
Both are aimed at decoupling – controller decouples the presentation from the business objects while observer decouples the event handlers from the event source. As a consequence, both accomplish low coupling.	
Both patterns support design for change, separation of concerns, high cohesion, low coupling, and designing "stupid objects."	
Both patterns increase reusability due to low coupling.	
Both patterns are variations of the model-view-controller (MVC) pattern. In particular, observer is a simplified version of the MVC, and controller is a special application of the MVC.	
Differences	
It is introduced to decouple presentation from business objects, and handle actor requests. Multiple presentations can be added and removed more easily.	It decouples event handlers from an event source so handlers can be added and removed freely.
Controller is a GRASP pattern. It implements part of a use case's behavior.	Observer is a situation-specific, GoF behavioral pattern.
Exclusive Applications	
All interface-based, interactive applications.	An online newsletter publication, and subscription management system is one possible application of the observer pattern.

Figure 16.5: Similarities and differences between controller and observer

Abstract Factory	Creator
Similarities	
Both patterns concern object creation.	
Differences	
It is a GRASP pattern, and answers the question: who should be assigned the responsibility to create a given object?	It is a situation-specific, GoF creational pattern. It solves the design problem: how to change the families of products created without affect the client?
Exclusive Applications	
An integrated development environment (IDE) that hides the differences between language-dependent tools from the client that uses the tools. Abstract factory can be used to get different families of tools for different programming languages.	Object creation is a common task of OO design and programming. Therefore, creator is useful in many OO systems.

Figure 16.6: Comparison between abstract factory and creator

16.4 The preorder traversal processes the node visited first, and then visits the left child and the right child of the node. Write an algorithm to implement an iterator for the preorder traversal.

Solution. The preorder traversal algorithm is:

```
public void preorder(Tree t)
{
   if (t==null) return;
   process (t);
   preorder(t.left);
   preorder(t.right);
}
```

There are at least two approaches to implement a preorder iterator. The easiest way runs above algorithm and saves each node visited in an ArrayList object. The preorder iterator delegates the request to the ArrayList iterator, or simply returns the ArrayList iterator as the preorder iterator. This approach is not efficient when the tree is large because two copies of the nodes need to be saved. Another approach uses a stack. When a node is visited, push its children onto the stack and process them when the node is processed. This is a recursive algorithm and easy to write.

16.5 The interpreter pattern (see Chapter 15) executes the val (c:Context):int function of each node of the parse tree during a postorder traversal. Design and implement a postorder traversal algorithm as a postorder iterator and apply it to evaluate the following business rule:

Order.getAmout() *geq* $250.00 ==> Order.setDiscountRate(0.10)

It states “if the order amount is greater than or equal to $250.00 then set the discount rate to 10

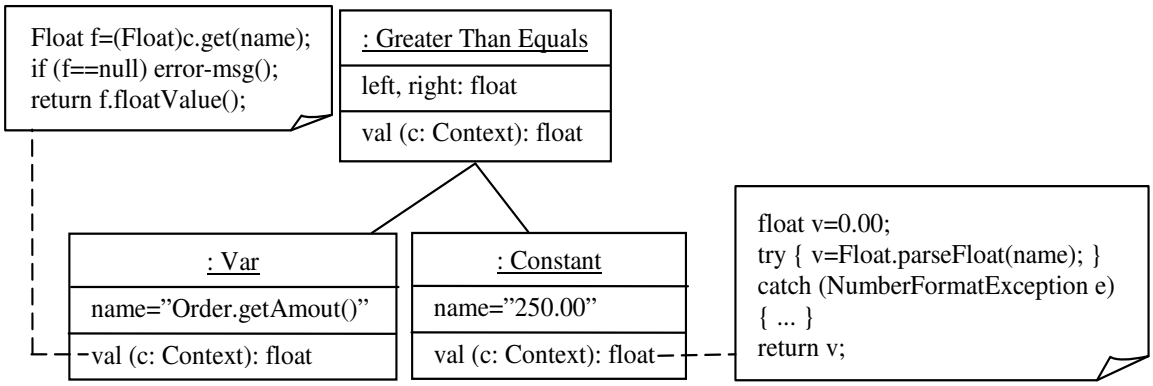


Figure 16.7: Parse tree for a discount rule

In addition, discuss the pros and cons to use the iterator pattern to implement the traversal algorithm.

Solution. The implementation of the postorder traversal iterator is similar to that of the preorder iterator discussed in the previous exercise. Therefore, it is not repeated here. Figure 16.7 shows a parse tree for above rule. To apply the postorder iterator, it is assumed that each node of the parse tree also has a isLeaf(): boolean method that tells if the node is a leaf node. The following code shows how to apply a postorder traversal iterator to evaluate the rule.

```
Iterator it=parseTree.postorderIterator();
while (it.hasNext()) {
   Node node=(Node)it.next();
   if (node.isLeaf())
     stack.push(node.val(c));
   else {
     Node node.right=stack.pop();
     Node node.left=stack.pop();
```

```
            stack.push(node.eval(c));
        }
    }
```

Using the iterator makes the software independent of the data structure used to implement the parse tree. However, compared with the conventional postorder traversal, the algorithm using the postorder iterator to evaluate the rule is not as straightforward.

16.6 Design, implement, and test the subsystem for purchase order processing described in Section 16.5.4. Assume that there are three levels of approval authorities: project manager, engineering director, and cooperate finance. They can approve each purchase that is within $10,000, $100,000, and $1 million, and not to exceed a total of $100,000, $1 million, and $10 million per year, respectively.

Solution. The solution should apply the chain of responsibility pattern. Figure 16.8 shows one possible solution.

16.7 Which other patterns can be applied to simplify the design and implementation of the purchase-order processing subsystem? How can this be accomplished? What assumptions must be made?

Solution. One possibility is to use the observer pattern. That is, the employee submits a purchase request, which is passed as a parameter to the request2Purchase(...) method. The purchase request is also a concrete observable, which will notify the employee when the request is approved or disapproved.

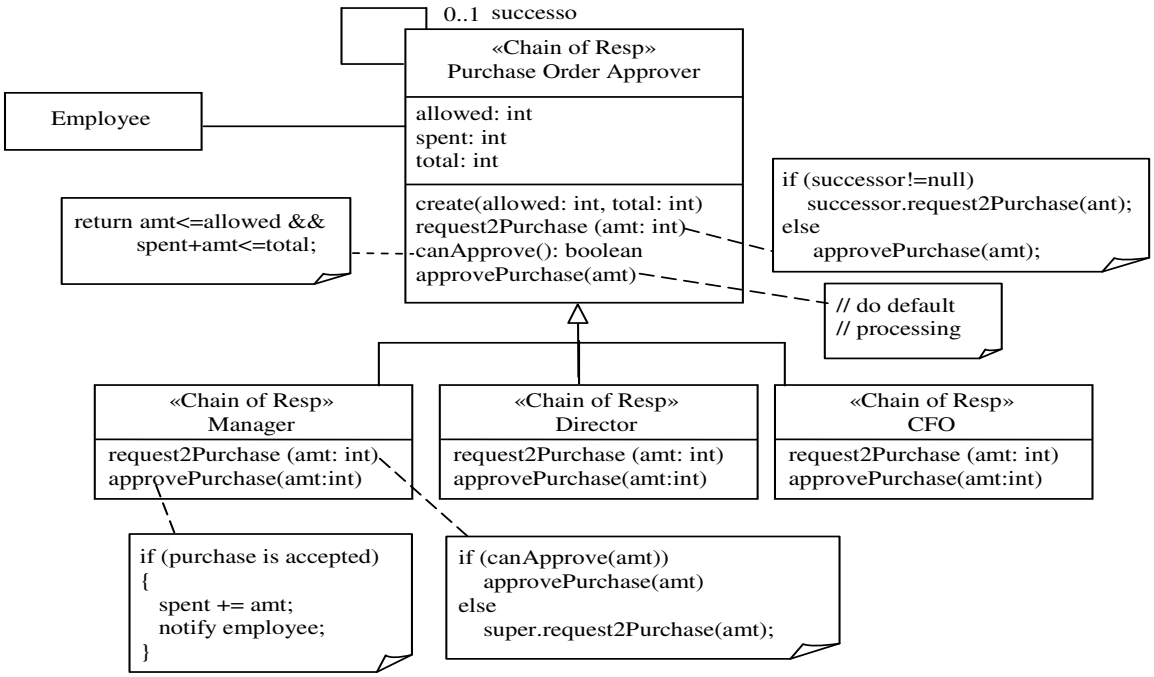


Figure 16.8: Purchase order processing with chain of responsibility

Chapter 17

Applying Patterns to Design a Persistence Framework

17.1 Discuss the similarities and differences of the two patterns in each of the following pairs of patterns. Moreover, give two situations that are not in the textbook or the lecture notes in which each pattern can be exclusively applied to exactly one situation. Describe how each pattern is applied and why the other pattern cannot be applied.

a. Bridge and command

b. Flyweight and prototype

c. Singleton and prototype

d. Abstract factory and factory method

e. Adapter and proxy

f. Bridge and strategy

g. Adapter and bridge

h. Decorator and proxy

Bridge	Command
Similarities	
Both implement a common interface.	
Both support design for change – it is easy to add concrete implementations in bridge and concrete command in command.	
Differences	
Bridge is a structural pattern.	Command is a behavioral pattern.
The concrete implementations implement the same functionality.	The concrete commands implement different functions.
Often, the client and the implementations are in one-to-one relationship.	The client execute several commands at a time.
The implementation and interface can be changed independently. The implementations can be changed without affecting the client.	The client is affected if the command interface is changed.
Operations cannot be undone and redone.	Operations can be undone and redone.
Operations are performed immediately.	Command objects can be saved and executed at a later time.
Exclusive Applications	
A fault-tolerant software that can switch to a different implementation when the current component causes a fault.	An agent-oriented system in which intelligent agents can generate and modify action plans on the fly, and execute a plan at a later time. Command can be used to implement the actions and composite can be used to store the plans.

Figure 17.1: Bridge and command

Solution. See Figure 17.1 through Figure 17.8.

17.2 Design and implement a prototype management system as shown in Figure 17.21, that is, the Prototype interface and the PrototypeMgr. Use the system to create cars of different makes and models, and documents of different types such as books, periodicals, and conference proceedings. Demonstrate how the system works using sample data.

Solution. Figure 17.9 of this manual shows a solution with two car prototypes. Document prototypes can be defined and used similarly.

17.3 Design and implement a protection proxy that allows a client to access a protected object only if the client has the correct pass code. Moreover, only one client can access the object at any time.

Solution. Figure 17.10 of this manual shows a sample implementation in Java.

Flyweight	Prototype
Similarities	
It uses a manager to manage a pool of flyweights.	It uses a manager to manage a pool of prototypes.
It may define a common interface for all flyweights.	It defines a common interface for all prototypes.
Flyweights are shared objects.	Prototypes are shared objects.
Flyweights can be added and removed easily.	Prototypes can be added and removed easily.
A flyweight can be added if it is not found in the pool of flyweights.	A prototype can be added if it is not in the pool of prototypes.
Differences	
The design problem is how to avoid creating numerous instances of a class.	The design problem is how to reduce the number of classes.
The solution is sharing flyweight objects using references.	The solution is reusing prototype objects through cloning.
It does not require cloning and implementation of the clone method.	It requires cloning and implementation of the clone operation.
In some sense, flyweight can be prototype that supports only shallow copying.	In some sense, prototype with only shallow copying can be flyweight.
Exclusive applications	
A gaming application that needs to show numerous occurrences of certain patterns. For instance, the same image of a character needs to appear at many different places.	A gaming application involving complex objects of different types. A complex object may contain other objects, which may perform different actions at the same time; thus, deep copying is required and prototype should be used.

Figure 17.2: Flyweight and prototype

Singleton	Prototype
Similarities	
Both are situation specific, creational patterns.	
Differences	
It solves the design problem of how to control the creation of a limited number of globally accessible instances of a class.	Its design problem is how to reduce the number of classes needed to be defined.
Exclusive applications	
A common system calendar.	An application involving complex structures consisting of complex components. Prototypes of such structures can be stored and cloned to create similar structure.

Figure 17.3: Singleton and prototype

Abstract Factory	Factory Method
Similarities	
Both are situation specific, creational patterns.	
Differences	
Abstract factory is used to vary the families of products created.	Factory method is used to vary the individual products created.
The families of products are used by a client.	The individual products are used by the template method of the factory method.
Changing the concrete factory changes the family of product created; and hence, it changes the behavior of the client.	Changing the concrete classes that implement the hook methods changes the behavior of the template method; and hence, it changes the behavior of the client.
Exclusive applications	
Abstract factory can be used to provide tools for different programming languages. By changing the concrete factory, tools for different programming languages can be obtained. For example, JUnit for Java, CppUnit for C++, etc.	An IDE that supports one software process but different development methodologies. The methodologies implement the phases of the process differently.

Figure 17.4: Abstract factory and factory method

Object Adapter	Bridge
Similarities	
It has a hierarchy of adaptee classes.	It has a hierarchy of implementation classes.
It delegates client requests to one of the adaptee objects and this can be changed dynamically.	It delegates client requests to one of the concrete implementation objects and this can be changed dynamically.
Concrete adaptee classes can be added and removed easily.	Concrete implementation classes can be added and removed easily.
Both are structural patterns.	
Differences	
It is an “after-fact” solution to an interface mismatch problem.	It is a “before-fact” application of the design for change principle.
It converts one interface to another.	The interfaces can be the same.
Exclusive Applications	
An application that can selectively reuse one of several third party components implementing similar functions.	A fault-tolerant software that can switch to a different implementation when the current component causes a fault.

Figure 17.5: Adapter and proxy

Bridge	Strategy
Similarities	
Both support design for change, separation of concerns, and high cohesion.	
It has a hierarchy of concrete implementations.	It has a hierarchy of concrete strategies.
Concrete implementations can be added easily.	Concrete strategies can be added easily.
Concrete implementations implement the same functionality.	Concrete strategies implement same functionality but differ in the algorithmic aspect.
Differences	
It is a structural pattern.	It is a behavioral pattern – the concrete strategies differ in the algorithms they implement.
It allows the client interface and implementation to change independently.	It allows the strategies to apply to different class structures.
The client is unaware of the concrete implementations and which one it uses.	The client must know the strategies and how to choose them.
Exclusive Applications	
A fault-tolerant software that can switch to a different implementation when the running component causes a fault.	Discount policies for different seasons can be implemented as strategies.

Figure 17.6: Bridge and strategy

Object Adapter	Bridge
Similarities	
It has a hierarchy of adaptee classes.	It has a hierarchy of implementation classes.
It delegates client requests to one of the adaptee objects and this can be changed dynamically.	It delegates client requests to one of the concrete implementation objects and this can be changed dynamically.
Concrete adaptee classes can be added and removed easily.	Concrete implementation classes can be added and removed easily.
Both are structural patterns.	
Differences	
It is an “after-fact” solution to an interface mismatch problem.	It is a “before-fact” application of the design for change principle.
It converts one interface to another.	The interfaces can be the same.
Exclusive Applications	
An application that can selectively reuse one of several third party components implementing similar functions.	A fault-tolerant software that can switch to a different implementation when the current component causes a fault.

Figure 17.7: Adapter and bridge

Decorator	Proxy
Similarities	
It adds functionality to an object.	It adds functionality to an object – different types of proxy add different types of functionality.
A decorator encapsulates the object decorated and delegates client requests to it.	A proxy encapsulates the real object and delegates client requests to it.
Decorator defines a common interface for decorators and objects decorated.	Proxy defines a common interface for the proxy and the object represented by the proxy.
Both decorator and proxy are structural patterns.	
Differences	
There can be a hierarchy of decorators.	The relationship between a proxy and the real object it represents is generally fixed.
New types of decorator can be added easily.	There is a limited number of proxy types.
A concrete decorator can add its functionality to different objects of different classes.	A proxy adds its functionality to only one object.
A concrete decorator can be added to and removed from an object freely.	The functionality added by a proxy to a real object is permanent – it cannot be removed once added.
More than one type of decorator can be applied to a given object.	It is not common that more than one type of proxy is applied to a real object.
Decorators are application dependent.	The four types of proxies can be applied in different applications. In this sense, proxies are not application dependent.
Exclusive applications	
Special offers can be realized by decorators, which can be added to, or removed from, a retail item.	An application that uses at least one of the four types of proxies. For example, a protection proxy that implement role-based access control.

Figure 17.8: Decorator and proxy

```
package prototype;

public interface Prototype {
  public Object clone();
}

package prototype;

import java.util.*;
import java.io.*;

public class ProtoMgr {
  Hashtable prototypes;
  private static ProtoMgr instance;
  private ProtoMgr() {
    File protoFile=new File ("prototypes.dat");
    if (protoFile.exists()) {
      try {
        ObjectInputStream ois=new
           ObjectInputStream(
           new FileInputStream("protoFile.dat"));
        prototypes=(Hashtable)ois.readObject();
        ois.close();
      } catch (ClassNotFoundException e) {
        e.printStackTrace();
      } catch (IOException e1) {
        e1.printStackTrace();
      }
    } else {
      prototypes=new Hashtable();
    }
  }
  public static ProtoMgr getInstance() {
    if (instance==null) instance=new ProtoMgr();
    return instance;
  }
  public Object get(String key) {
    return prototypes.get(key);
  }
  public void add(String key, Object prototype) {
    prototypes.put(key, prototype);
  }
}

package prototype;

public class Car implements Prototype {
  String make, model;
  int horsepower, seats, yr;
  public Car(String make, String model,
     int horsepower, int seats, int yr) {
    this.make=make; this.model=model;
    this.horsepower=horsepower;
    this.seats=seats; this.yr=yr;
  }
  public Object clone() {
    return this.clone();
  }
  public void setMake(String make) {
    this.make=make;
  }
  public void setModel(String model) {
    this.model=model;
  }
  public void setHorsepower(int horsepower) {
    this.horsepower=horsepower;
  }
  public void setSeats(int seats) {
    this.seats=seats;
  }
  public void setYr(int yr) {
    this.yr=yr;
  }
  public String toString() {
    return make+" "+model+" "+horsepower+" "
         +seats+" "+yr;
  }
  // other operations
}

package prototype;

public class PrototypeMain {
  public static void main(String[] args) {
    Car ford=new Car("Ford", "Mustang", 662, 4,
2013);
    ProtoMgr.getInstance().add("Ford Mustang",
ford);
    Car
gm=(Car)ProtoMgr.getInstance().get("GM");
    if (gm==null) {
      gm=new Car("GM", "Buick", 200, 4, 2012);
      ProtoMgr.getInstance().add("GM Buick",
gm);
    }
    Car
myFord=(Car)ProtoMgr.getInstance().get("Ford
Mustang");
    myFord.setHorsepower(300);
    myFord.setSeats(2);
    System.out.println("My Ford is: "+myFord);
  }
}
```

Figure 17.9: Prototype manager sample code

```
package proxy;

public interface ProxyIF {
  public void operation();
}

package proxy;

/** object to protect */
public class Secret implements ProxyIF {
  public void operation() {
     System.out.println("Secret operation() is “+
       “performed.");
  }
}
```

```
package proxy;

public class ProtecProxy implements ProxyIF {
  private Secret secret;
  int hashCode, count;
  public ProtecProxy(String pass, Secret secret) {
    hashCode=pass.hashCode();
    this.secret=secret;
    count=0;
  }
  public void operation() {
    secret.operation();
  }
  public void operation(String pass)
      throws IllegalAccessException  {
    if (pass.hashCode()==hashCode &&
        count==0) {
      count = 1;
      operation();
      count = 0;
    } else
      new IllegalAccessException("Invalid pass");
  }
}
```

Figure 17.10: A protection proxy with a smart counter

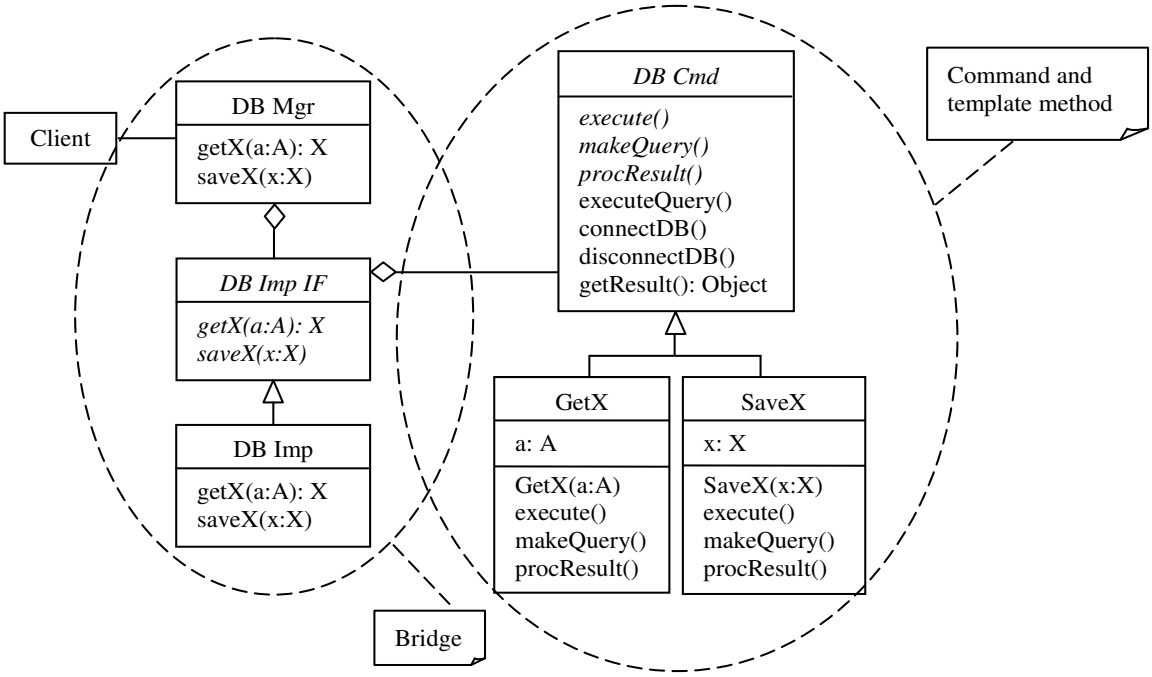


Figure 17.11: Persistence framework using bridge, command and template method

17.4 Design and implement a persistence framework for accessing a local database using only the bridge and command patterns. The design does not need to consider that the database management system (DBMS) may change in the future. As tests, store and retrieve Book and Patron objects with the persistence framework. Choose your implementation language and DBMS, or as directed by your instructor.

Solution. Figure 17.11 shows a design of the framework, where X denotes any class that needs to be stored and retrieved. The implementation part is omitted here.

17.5 Modify the design and implementation in exercise 17.4 to allow undo and redo database operations.

Solution. Figure 17.12 shows a sample design. The getX(...) operation does not change the database, and hence, undo() and redo() do nothing. The save(x: X) operation needs to retrieve the existing x and save it for later undo. To undo a save(x: X) operation, one needs

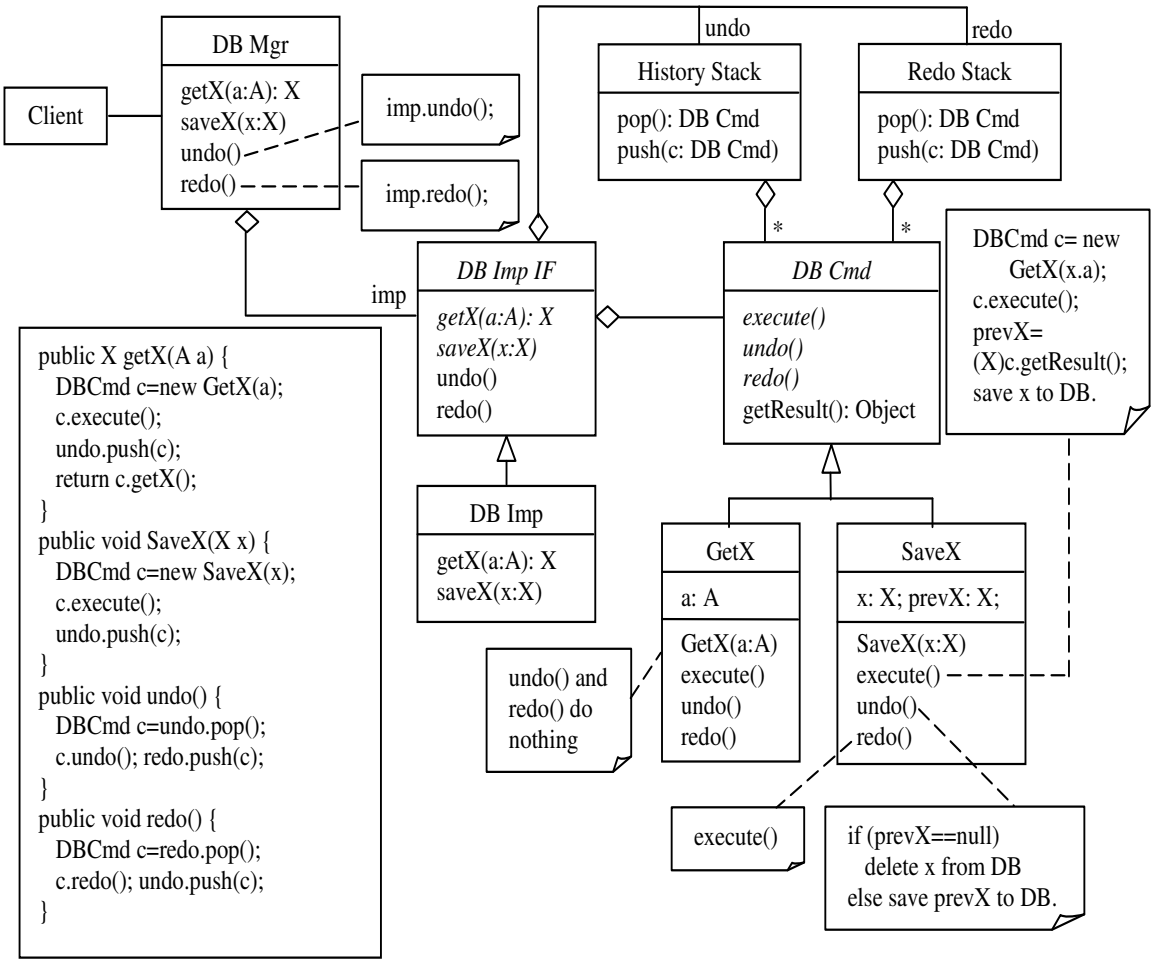


Figure 17.12: Persistence framework with undo and redo

to consider two cases. If x did not exist before the save(x: X) operation was performed, then one needs to delete x from the database. Otherwise, one needs to save the previous object x back to the database.

17.6 Design and implement a persistence framework to access a local database. The design should consider that the local DBMS may be replaced in the future. Choose your implementation language and DBMS, or as directed by the instructor.

Solution. The design in Figure 17.11 is also a solution for this exercise.

17.7 Change the design and implementation of the persistence framework in exercise 17.6 to

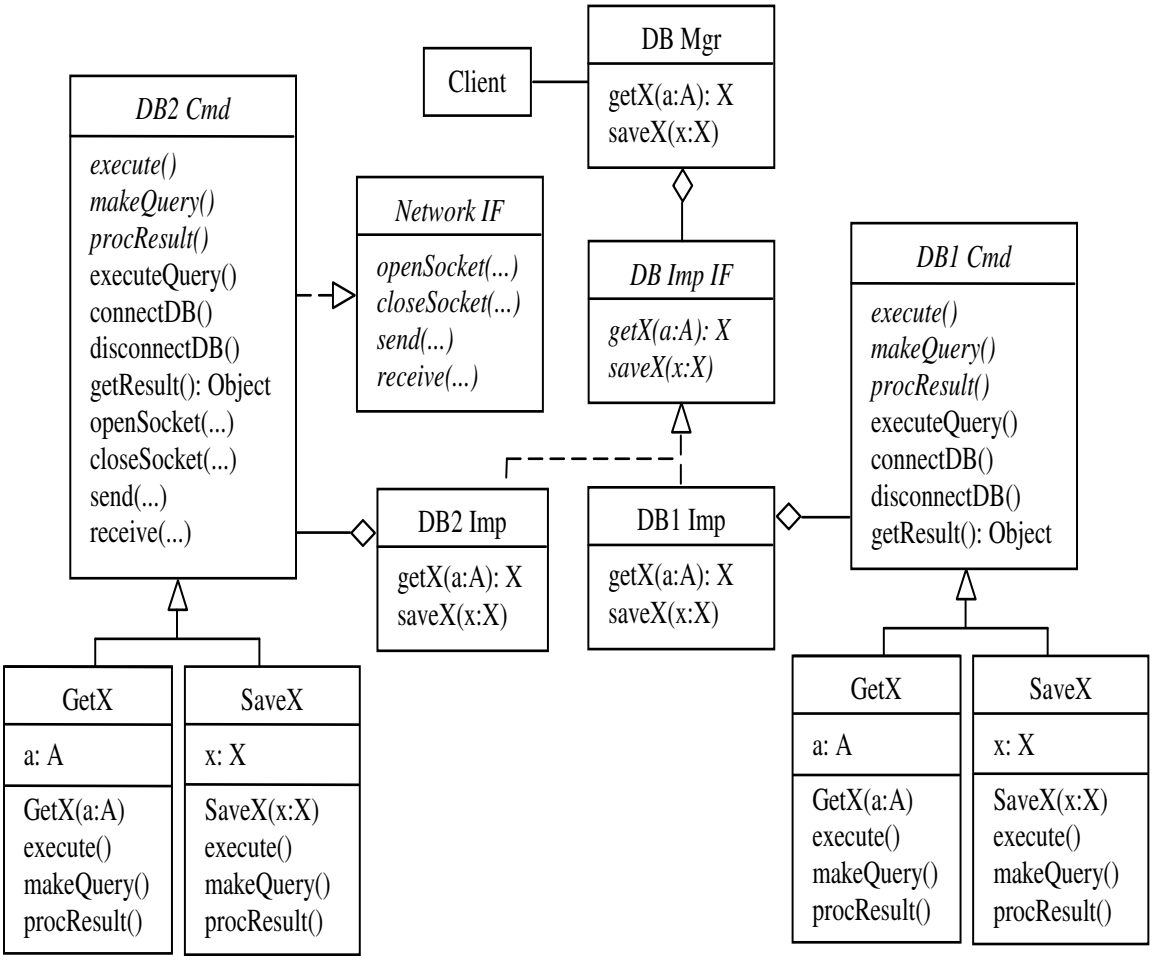


Figure 17.13: Persistence framework supporting a remote database

take into account that the persistence framework needs to support local as well as remote databases.

Solution. Figure 17.13 provides a solution.

17.8 Describe two situations in which the prototype pattern can be applied to reduce the number of classes in one, but not the other situation.

Solution. The prototype pattern can be applied to reduce the number of classes if the classes differ only in attributes. One such application is drawing polygon, rectangle, line, and triangle shapes. Such a shape can be stored as a collection of Point objects and drawn using a loop.

Thus, prototype can be used. Prototype cannot be used to reduce the number of concrete animal classes such as Cat, Dog, Snake because animals behavior differently.

Part VI

IMPLEMENTATION AND QUALITY ASSURANCE

Chapter 18

Implementation Considerations

Chapter 16 describes the design of a state diagram editor. It produces a number of design diagrams. The following three exercises are related to the design of the editor. Each program file must include the file header and description of classes using the format in Figures 18.2 and 18.3.

18.1 Produce a directory structure for the state diagram editor and assign the editor-related classes presented in Chapter 16 to the directories and subdirectories.

Solution. Figure 18.1 shows a possible design of the directory structure. In the figure, patterns are assigned to separate directories. This facilitate reuse of the patterns. For example, to reuse the strategy pattern, one needs to copy the strategy directory, or import the strategy package.

18.2 Implement the classes in Figure 16.6 using test-driven development. Use JUnit to write and run the test cases, and Cobertura to ensure that 100% branch coverage is accomplished.

Solution. The order to test-driven develop the classes is: Diagram Element, State, Transition, and State Diagram, taking into account the dependencies between the classes. The skeleton code for the Diagram Element abstract class is shown in Figure 16.8 of the textbook.

Directory	Parent Directory	Classes
editor		
src	editor	
diagram	src	*Diagram Element* State Transition State Diagram
patterns	src	
strategy	patterns	*Layout Algorithm* Force-Based Layout Orthogonal Layout Hierarchical Layout
visitor	patterns	*Checker* State Checker Trans Checker
memento	patterns	Memento Diagram State
abstract factory	patterns	*Abstract Factory* UML 1 Factory UML 2 Factory UML 1 State UML 2 State UML 1 Transition UML 2 Transition UML 1 State Diagram UML 2 State Diagram
state	patterns	Controller State Init Add State Add Transition Trans Source Selected
observer	patterns	State Button Listener Trans Button Listener
chain of responsibility	patterns	*Helper* Editor Helper Widget Helper Dialog Helper Condition Helper Button Helper
decorator	patterns	UML Note
controller	src	Edit State Diagram Controller
gui	src	Editor GUI

Figure 18.1: Designing directory structure for the editor

This class does not implement any functionality; and hence, TDD is omitted.

Next, TTD for the State class is performed. The State class has three methods: the constructor, draw, and intersect. The draw method draws to a Graphics2D object a circle with a dummy state label. One can use a mock object to check whether the draw method correctly draws the circle with the dummy state label. Figure 18.2 shows how this is accomplished. First, a subclass, called MockGraphics, of Graphics2D is created. This subclass overrides the drawArc and drawString methods of Graphics2D. In particular, the redefined methods simply save the respective calls to these functions to a collection, which in Figure 18.2 is TreeSet.

Figure 18.3 shows the JUnit test cases for the State class. One should generate the skeleton code for the State class before running the test cases. The skeleton code includes the State class constructor, an empty draw method, an empty intersect method, and get methods. The skeleton code should fail all the test cases except the testStateConstructor test case. Using Corbertura to measure the code coverage is described in the appendix of the textbook and is omitted here.

18.3 Implement the classes in Figure 16.26 using test-driven development and pair programming. Form the pair yourselves or according to the instructions given by the instructor. Use JUnit to write and run the test cases, and Cobertura to ensure that 100% branch coverage is accomplished.

Solution. Omitted.

18.4 Implement the persistence framework presented in Chapter 17. Assume that the application is a library information system and the database is a local database. Do this exercise using test-driven development, pair programming, and JDBC, or as instructed by the instructor. In addition, assume that the classes in Figure 11.10 have the following attributes: Document(cn:

```
package example;

import java.util.*;

public class DrawActions
    extends TreeSet {
    public DrawActions() {
      super();
    }
    public TreeSet getActions() {
      return this;
    }
}
```

```
package example;

import java.awt.*;
import java.awt.image.ImageObserver;
import java.awt.image.renderable.RenderableImage;
import java.util.Map;
import java.awt.geom.AffineTransform;
import java.awt.image.BufferedImageOp;
import java.text.AttributedCharacterIterator;
import java.awt.font.FontRenderContext;
import java.awt.image.RenderedImage;
import java.awt.font.GlyphVector;
import java.awt.image.BufferedImage;
import java.awt.RenderingHints.Key;

public class MockGraphics
    extends Graphics2D {
  DrawActions actions = new DrawActions();

  public MockGraphics() {
    super();
  }

 public void drawArc(int x, int y, int width,
    int height, int startAngle, int arcAngle) {
    String funcCall="drawArc(" + x + "," + y + "," +
        width + "," + height + "," +
        startAngle + "," + arcAngle + ");";
    actions.add(funcCall.replaceAll(" ", ""));
  }

  public void drawString(String str, int x, int y) {
    actions.add("drawString(" + str + "," + x + "," + y + ");");
  }
  public DrawActions getActions() { return actions; }
  // implementation of other abstract methods here
```

Figure 18.2: Using a mock object in test-driven development

```
package test;

import java.util.Iterator;
import junit.framework.*;
import java.awt.*;
import example.*;
import editor.*;
import junit.swingui.TestRunner;

public abstract class StateTest
    extends TestCase {
  Point center;
  protected State state;
  protected MockGraphics mockGraphics;
  protected Point inPoint;
  protected Point outPoint;
  public StateTest(String name) {
    super(name);
  }

  public void setUp() {
    center=new Point(60, 60);
    state = new State("State0", center);
    mockGraphics = new MockGraphics();
    inPoint=new Point(70, 60);
    outPoint=new Point(90, 60);
  }
  public void testStateConstructor() { // test constructor
    assertEquals("testState: names are not equal.", "State0", state.getName());
    assertTrue("testState: locs are not equal.", state.getLoc().getX() == 60 &&
               state.getLoc().getY() == 60);
  }
  public void testDraw() {
    mockGraphics.drawArc(state.getLoc().x, state.getLoc().y, 25, 25, 0, 360);
    mockGraphics.drawString(state.getName(), 60 - state.getName().length()*3, 60 + 5);
    DrawActions expected=mockGraphics.getActions();
    Iterator it=expected.iterator();
    MockGraphics g=new MockGraphics();
    state.draw(g);
    DrawActions actual=g.getActions();
    assertTrue("actual not contain expected: ", actual.containsAll(expected));
    assertTrue("expected not contain atual: ", expected.containsAll(actual));
  }
  public void testIntersect() {
    assertTrue("incorrect intersect", state.intersect(inPoint) &&
        !state.intersect(outPoint));
  }
  public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new StateTest("testStateConstructor") {
      protected void runTest() {
        testStateConstructor();
      }
    });
    suite.addTest(new StateTest("testDraw") {
      protected void runTest() {
        testDraw();
      }
    });
    suite.addTest(new StateTest("testIntersect") {
      protected void runTest() {
        testIntersect();
      }
    });
    return suite;
  }
  public static void main(String[] args) {
    new TestRunner().start(new String[] {"test.StateTest"});
  }
}
```

Figure 18.3: State class test cases

String, title: String, author: String, publisher: String, ISBN: String, year: String, duration: String) Loan(patron: Patron, document: Document, date: String, dueDate: String), the date has the format as MM/DD/YYYY. Patron(id: String, name: String, address: String, tel: String)

Solution. The implementation should be easy because the textbook has provided detailed descriptions of the classes of the framework. Therefore, the solution is not provided here.

18.5 Implement the design of the business rules class diagram you produced in exercise 15.7.

Solution. Figure 18.4 shows a sample implementation.

18.6 Implement the design of the lawn mowing agent you produced in exercise 13.6.

Solution. Figure 18.5 shows a sample implementation.

18.7 Implement the graphical user interface for the state diagram editor presented in Chapter 16.

Solution. Some sample code is shown in Figure 18.6.

18.8 Design and implement a web page or two that allows the user to search for documents in a library information system. Do this exercise based on the persistence framework you produced in the above exercise. Use JSP or a technology as instructed by your instructor. An introduction to JSP is given in Appendix B. Hint: You need to design and implement a search use case controller that interacts with the DBMgr. You also need to extend the persistence framework to provide the search capability.

Solution. Omitted.

```
package rulebase;

public class Application {
  String status;
  public void setStatus(String status) {
    this.status=status;
  }
}
```

```
package rulebase;

import java.util.*;
public class Condition {
  private String name; char value;
  public Condition(String name,
    char value) {
    this.name = name; this.value = value;
  }
  public boolean eval(Hashtable h) {
    Character c = (Character) h.get(name);
    char v = c.charValue();
    return v == value;
  }
}
```

```
package rulebase;

import java.util.*;

public class ConditionList {
  private ArrayList conditions =
    new ArrayList();
  public void add(Condition c) {
    conditions.add(c);
  }
  public boolean eval(Hashtable h) {
    Iterator it=conditions.iterator();
    while (it.hasNext()) {
      Condition c = (Condition) it.next();
      if (!c.eval(h)) {
        return false;
      }
    }
    return true;
  }
}
```

```
package rulebase;
import java.util.*;
public class Rule {
  private ConditionList left;
  private Action right;
  public Rule (ConditionList left,
    Action right) {
    this.left=left; this.right=right;
  }
  public boolean eval(Hashtable h) {
    if (left.eval(h)) {
      right.execute(h);
      return true;
    }
    return false;
  }
}
```

```
package rulebase;
import java.util.*;
public class RuleSet {
  private ArrayList rules=new ArrayList();
  public void add(Rule r) {
    rules.add(r);
  }
  public boolean eval(Hashtable h) {
    Iterator it=rules.iterator();
    while (it.hasNext()) {
      Rule r = (Rule) it.next();
      if (r.eval(h)) return true;
    }
    return false;
  }
}
```

Figure 18.4: Implementation of a mini rule-based system

```
package lawnmower;
public abstract class State {
  protected static int[][] A = new
int[20][20];
  protected static int uncutCells=400;
  protected static int x, y;
  public void cut() {
     if (A[x][y]==2) {
       A[x][y]=1; uncutCells--;
     }
  }
  public State gotoCell() {
    // goto nearest uncut cell;
    State s=new East(); // assumed
      //resulting state
    return s;
  }
  public abstract State onEntry();
}
```

```
package lawnmower;

public class East extends State {
  public State onEntry() {
    if (A[x][y-1]==2) {
       //turn left;
       y--; cut();
       return new North();
    }
if (A[x+1][y]==1 || A[x+1][y]>2) {
   if (A[x][y+1]>2) return gotoCell();
   else {
      //turn right;
      y++; cut();
      //turn right;
      return new West();
   }
} else {
   x++; cut();
   return this;
}

  }
}
```

```
package lawnmower;

public class North
    extends State {
    public State onEntry() {
      /** to be implemented */
      return null;
    }
}
```

```
package lawnmower;

public class West
    extends State {
    public State onEntry() {
      /** to be implemented */
      return null;
    }
}
```

```
package lawnmower;

public class LawnMowerAgent {
  private State state;
  public void scheduleOn() {
    // initalize A[m,n],
    //(x,y) and uncutCells.
    state = new East();
    while (State.uncutCells > 0) {
      state = state.onEntry();
    }
  }
}
```

Figure 18.5: Lawn mower agent implementation

```
package editor;

import java.awt.Point;
import java.awt.Graphics2D;
public abstract class DiagramElement {
   String name;
   Point loc;
   public DiagramElement(String name, Point loc) {
     this.name=name; this.loc=loc;
   }
   public abstract void draw(Graphics2D g);
   public abstract boolean intersect(Point p);
   public void add(DiagramElement e) { }
   public DiagramElement get (String name) {
     return null;
   }
   public void remove (DiagramElement e) {
   }
   public String getName() { return name; }
   public void setName(String name) { this.name=name; }
   public Point getLoc() { return loc; }
   public void setLoc(Point p) { this.loc=p; }
}
```

```
package editor;

import java.awt.Point;
import java.awt.Graphics2D;
public abstract class DiagramElement {
   String name;
   Point loc;
   public DiagramElement(String name, Point loc) {
     this.name=name; this.loc=loc;
   }
   public abstract void draw(Graphics2D g);
   public abstract boolean intersect(Point p);
   public void add(DiagramElement e) { }
   public DiagramElement get (String name) {
     return null;
   }
   public void remove (DiagramElement e) {
   }
   public String getName() { return name; }
   public void setName(String name) { this.name=name; }
   public Point getLoc() { return loc; }
   public void setLoc(Point p) { this.loc=p; }
}
```

```
package editor;

import java.awt.Graphics2D;
import java.awt.Point;
import java.awt.Graphics;

public class Transition
    extends DiagramElement {
  int from, to; // from and to state number;
  Point dest;
  public Transition(String name, Point loc) {
    super(name, loc);
  }
  public void setDest(Point p) {
    this.dest=p;
  }
  public void draw(Graphics2D g) {
      if (from!=to) { // only draw to a different state
        drawArrowLine(g);
        drawArrowArc(g);
      }
   }
```

Figure 18.6: State diagram editor code (to be continued)

```
public void drawArrowLine(Graphics g) {
      Graphics2D g2=(Graphics2D)g;
      g2.drawLine(loc.x, loc.y, dest.x, dest.y);
   }
   public void drawArrowArc(Graphics g) {
      Graphics2D g2=(Graphics2D)g;
      double dx=(dest.x-loc.x);
      double dy=(dest.y-loc.y);
      double theta=0;
      if (dx==0) theta=Math.atan(dy/0.01);
      else {
         if (dx>0)
            theta = Math.atan(dy / dx);
         else
            theta =3.14+ Math.atan(dy/dx);
      }
      double alpha=60*3.14/180-theta;
      int x[]=new int[]{(int)(loc.x+dx-10*Math.sin(alpha)), dest.x,
         (int)(loc.x+dx-10*Math.cos(theta-30*3.14/180))};
      int y[]=new int[]{(int)(loc.y+dy-10*Math.cos(alpha)), dest.y,
         (int)(loc.y+dy-10*Math.sin(theta-30*3.14/180))};
      g2.drawPolyline(x, y, 3);
      g2.drawString(name, loc.x+(float)dx/2-name.length()*2, loc.y+(float)dy/2);
   }

  public boolean intersect(Point p) {
    return p.getY()/p.getX()==dest.getY()/dest.getX();
  }
}

package editor;

import java.awt.Graphics2D;
import java.awt.Point;
import java.util.*;
public class StateDiagram
    extends DiagramElement {
  private Collection children=new ArrayList();
  public StateDiagram(String name, Point loc) {
    super(name, loc);
  }
  public void draw(Graphics2D g) {
    Iterator it=children.iterator();
    while (it.hasNext()) {
      DiagramElement e=(DiagramElement)it.next();
      e.draw(g);
    }
  }
  public boolean intersect(Point p) {
    Iterator it=children.iterator();
    while (it.hasNext()) {
      DiagramElement e=(DiagramElement)it.next();
      if (e.intersect(p)) return true;
    }
    return false;
  }
  public void add(DiagramElement e) {
    children.add(e);
  }
  public void remove(DiagramElement e) {
    children.remove(e);
  }
}
```

Figure 18.6: State diagram editor code (continued)

Chapter 19

Software Quality Assurance

19.1 What is the cyclomatic complexity of the flowchart in Figure 18.5? Apply the three approaches to compute the cyclomatic complexity, that is, counting the number of regions plus one, counting the number of atomic binary decision plus one, and counting the number of nodes and edges. Check that these results are identical. If not, explain why they are not identical.

Solution. There are four closed regions, therefore, the cyclomatic complexity is 5. There are four atomic binary conditions, therefore, teh complexity is also 5. There are 12 nodes and 15 edges. The cyclomatic complexity is number of edges less number of nodes plus 2. That is, 15-12+2=5.

19.2 Test-driven development (TDD) requires "write tests first, then write code." This is not always easy because the functions of a class may be too complex and difficult to decompose. However, test cases can be generated from a flowchart rather than a flow graph. A flowchart is a detailed design produced before coding. A flow graph represents an implementation, which is not available before coding. This exercise requires you to produce the test cases from the flowchart in Figure 18.5. Check that the correct number of test cases are generated.

	Fan-In	Fan-Out
Checkout Controller	1	4
Checkout GUI	0	1
DBMgr	1	2
Document	3	0
Loan	2	2
Patron	2	1
User	1	0
Total	10	10

Figure 19.1: Fan-in and fan-out for the DCD in Figure 11.10

Solution. There are five basis-path test cases:

1. (B)-(1)-(2)-(3)-(1)-(2)-(3)-(4)-(5)-(6)-(7)-(8)-(1)-(2)-(3)-(4)-(5)-(6)-(7)-(8)-(9)-(10)-(5)-(6)-(7)-(8)-(9)-(E)
2. (B)-(1)-(2)-(3)-(4)-(5)-(6)-(7)-(8)-(9)-(E)
3. (B)-(1)-(2)-(3)-(1)-(2)-(3)-(4)-(5)-(6)-(7)-(8)-(1)-(2)-(3)-(4)-(5)-(6)-(7)-(8)-(9)-(10)-(5)-(6)-(7)-(8)-(9)-(E)

19.3 Compute the conventional design metrics shown in Figure 19.2 for the design class diagram in Figure 11.10. Hint: Section 19.3.3 describes how this can be done. For conditional calls, check the design sequence diagram, where conditional calls may be indicated.

Solution. Only fan-in and fan-out can be applied. Figure 19.1 shows these metrics.

19.4 For the design class diagram in Figure 11.10, compute the object-oriented quality metrics presented in Section 19.3.4. Hint: Since the metrics are computed before coding; therefore, implementation data are not available. Use your best guess to determine the number of atomic binary decisions of a method, or sketch a flowchart to describe the detailed design for the method. For methods that call other methods of other classes, check the sequence diagram to find out.

Solution. Figure 19.2 shows the computed metrics.

Quality Metrics	Checkout Controller	Checkout GUI	DBMgr	Document	Loan	Patron	User
Weighted Methods per Class (WMC)	3		4	2	1		
Depth of Inheritance Tree (DIT)	0	0	0	0	0	1	0
Number of Children (NOC)	0	0	0	0	0	0	1
Coupling Between Object Classes (CBO)	It is equal to the fan-out of the class.						
Response for a Class (RFC)	7	1	3	2	1		
Lack of Cohesion in Methods (LCOM)	0	0	3	-1*	0		

* According to the original definition, the LCOM is set to zero if it is less than 0.

Figure 19.2: OO metrics for Figure 11.10 of textbook

19.5 Practice peer review, inspection, and walkthrough on a number of sample requirements specifications, design specifications including diagrams, and source code. For the review and inspection, try to use the checklists presented in previous chapters.

Solution. Omitted.

19.6 Exercise 4.2 produces the software requirements specification (SRS) for each of a telephone answering system, a vending machine, and a web-based only email system. Perform the three types of requirements review to these SRSs using the review checklists presented in Section 4.5.6. Write a review report for each of the SRSs.

Solution. Omitted.

Chapter 20

Software Testing

20.1 For the average function presented in Figure 20.8 do the following:

1. Write a brief functional description for the function.
2. Generate functional test cases based on the functional description.
3. Identify and specify the partitions and generate partition test cases.
4. Generate boundary value test cases.
5. Implement the average function in a class; also implement the test cases using JUnit or any other test tool as designated by the instructor.
6. Compile and run the test cases. Record any failures and errors that are reported. Analyze and briefly explain why each of the failures and errors occurs and how to fix them. Correct the failures and errors until the CUT passes all the test cases.

Solution. Described below are answers to the first five questions:

1. Brief functional description: The average(k: int) function computes the average of the first k elements of an integer list.

2. Let L=(I1, I2, ..., In) be a list of integers. The functional test cases should consider the following cases:

   1. k<n: The expected result is (int)((I1 + I2 + ... + Ik)/k).

   2. k>=n: The expected result is (int)((I1 + I2 + ... + In)/n).

3. The partitions for k are: (1) 0 <= k <= n, and (2) k < 0.

   The partitions for L are: (1) Lists that contain more than k elements, and (2) lists that contain less than k elements.

   Thus, there are three test cases: (1) 0 <= k <= n and a list with more than k elements, (2) 0 <= k <= n and a list with less than k elements, and (3) k < 0 and a list of a number of elements.

4. The boundary value test cases are: k=0, k=1, k=n-1, k=n, k=n+1, n=0, n=1, n=an extremely large number, and a null reference (or pointer) to a list.

5. Figure 20.1 shows some of the test cases.

20.2 Perform basis path testing to the average function shown in Figure 20.8. Implement the CUT in Java and the test cases in JUnit, or according to instructions given by the instructor. Run the test cases and analyze the test results. Correct any problems and write a brief test report about the problems; include why they occur and how they are fixed.

Solution. The average function has two atomic binary conditions. Therefore, it should have three basis paths. However, only two of them are feasible:

(1) B-1-2-3-E, and

(2) B-1-2-4-5-6-5-7-E

B-1-2-4-5-7-E is not feasible because when n>0, the loop must be exercised at least once. The implementation of the two test cases is shown in Figure 20.2.

```
package average;

import junit.framework.*;
import junit.swingui.TestRunner;

public class AverageTest
    extends TestCase {
  protected Average avg;
  public AverageTest(String name) {
    super(name);
  }
  public void setUp() {
    avg=new Average();
  }
  public void testKLessThanN() {
    int[] aList=new int[] {1, 2, 3, 4, 5};
    int k=(int)Math.random()*5+1;
    int expected=0;
    for (int i=0; i<k; i++)
       expected += aList[i];
    expected=(int)expected/k;
    int result=avg.average(k, aList);
    assertTrue("k<n failed: ",
        expected==result);
  }
public void testKGreaterThanN() {
    int[] aList=new int[] {1, 2, 3, 4, 5};
    int k=(int)Math.random()*100+5;
    int expected=0;
    int j=Math.min(k, aList.length);
    for (int i=0; i<j; i++)
       expected += aList[i];
    expected=(int)expected/j;
    int result=avg.average(k, aList);
    assertTrue("k>=n failed: ",
        expected==result);
  }

  public static Test suite() {
    return new
        TestSuite(AverageTest.class);
  }
  public static void main(String[] args)
  {
    new TestRunner().start(new String[]
        {"average.AverageTest"});
  }
}
```

Figure 20.1: Some test cases for the average function

```
package basispath;

import junit.framework.*;
import test.StateTest;
import junit.swingui.TestRunner;
import average.*;

public class AverageTest
    extends TestCase {
  protected Average avg;
  public AverageTest(String name) {
    super(name);
  }
  public void setUp() {
    avg=new Average();
  }
  public void testNEquals0() {
    int[] aList=new int[] {};
    int k=(int)Math.random()*100;
    int expected=0;
    int result=avg.average(k, aList);
    assertTrue("n=0 failed: ",
        expected==result);
  }
public void testNGreaterThan0() {
    int[] aList=new int[] {1, 2, 3, 4, 5};
    int k=(int)Math.random()*5+1;
    int expected=0;
    int j=Math.min(k, aList.length);
    for (int i=0; i<j; i++) expected +=
        aList[i];
    expected=(int)expected/j;
    int result=avg.average(k, aList);
    assertTrue("n>0 failed: ",
        expected==result);
  }

  public static Test suite() {
    return new
        TestSuite(AverageTest.class);
  }
  public static void main(String[] args)
  {
    new TestRunner().start(new String[]
        {"average.AverageTest"});
  }
}
```

Figure 20.2: Basis path test cases for the average funciton

20.3 Perform test-driven development to implement a singly-linked list that provides at least the following functions. Note: You are not allowed to use any Java API such as ArrayList and LinkedList as a replacement for the singly linked list — i.e., you must implement the singly linked list. Moreover, you are not allowed to implement a doubly linked list.

- public void insert(Object o1, Object o2): insert object o1 in front of object o2 in the linked list.
- public Object find(String attrName, String attr- Value): Returns the object with the given attribute name attrName with the given attribute value attrValue. If more than one such object is found, the first such object is returned. If not found, it returns null. You need to use Java reflection to do this. This function is, in fact, a polymorphic function, that is, the attrValue could be int, double, etc. But for this homework, you can assume that the attribute value is string type.
- public Object remove(Object obj): Removes the object referred to by the object reference obj from the linked list.
- public int size(): Returns the size of the linked list.

Solution. Figure 20.3 shows some of the test cases for the singly-linked list. The implementation of the singly-linked list is shown in Figure 20.4.

20.4 Apply cause-effect testing to test the singly linked list. Implement the test cases in JUnit and measure the code coverage using Cobertura or other coverage tool as determined by the instructor. Your test cases must achieve 100% branch coverage. Cobertura is described in Appendix C.

Solution. Cause-effect testing of the singly-linked list (SLL) should be performed for each of the member functions of the SLL. That is, a decision table should be constructed for each

```
package sll;

import junit.framework.*;
import junit.swingui.TestRunner;

public class SLLTest
    extends TestCase {
  protected SLL emptyList;
  protected SLL nonEmptyList;
  protected Node n1, n2, n3, n4;
  public SLLTest(String name) {
    super(name);
  }
  public void testEmptyListInsert() {
    Node n=new Node("2");
    emptyList.insert(n, null);
    String result=
        emptyList.toString().trim();
    assertEquals("Empty list insert
        failed.", "2", result);
  }
  public void testNonEmptyListInsert() {
    Node n4=new Node("4");
    nonEmptyList.insert(n4, n3);
    String
        result=nonEmptyList.toString().tr
        im();
    assertEquals("non-empty list insert
        failed.", "1 2 4 3", result);
  }
public void testFind() {
    Node result=nonEmptyList.find("data",
        "3");
    assertEquals(“Test of find failed.”,
        n3, result);
  }
  public void testNotFound() {
    Node result=nonEmptyList.find("data",
        "5");
    assertNull("test NotFound failed.",
        result);
  }
  public void setUp() {
    emptyList=new SLL();
    nonEmptyList=new SLL();
    n1=new Node("1"); n2=new Node("2");
    n3=new Node("3");
    nonEmptyList.insert(n3, null);
    nonEmptyList.insert(n2, n3);
    nonEmptyList.insert(n1, n2);
  }
  public void tearDown() {
    // do nothing;
  }
  public static void main(String[] args)
  {
    new TestRunner().start(new String[]
        {"sll.SLLTest"});
  }
}
```

Figure 20.3: Singly-linked list test cases

member function, taking into account conditions of the input to the member function and the outcomes expected. Figure 20.5 shows a decision table for the insert(Object o1, Object o2) operation for the linked list. Figure 20.6 shows some test cases implemented in JUnit.

20.5 Design equivalence partitioning and boundary value analysis test cases to test the singly-linked list and implement them in JUnit. Your test cases must achieve 100% branch coverage as measured by Cobertura or other coverage tool as determined by the instructor.

Solution. Omitted.

20.6 Repeat the last exercise, but instead of using equivalence partitioning and boundary value analysis, use basis path testing.

Solution. Basis path testing is a white-box testing technique. It should be applied to each nontrivial function of the CUT. Here, applying basis path to test the insert function of the

```
package sll;

public class Node {
  String data;
  Node next;
  public Node() {}
  public Node(String data) {
    this.data=data;
  }
  public String getData() { return data; }
  public void setData(String data) {
    this.data=data;
  }
  public Node getNext() { return next; }
  public void setNext(Node n) {
    this.next=n;
  }
  public String toString() {
    return data;
  }
  public boolean equals(Node n) {
    return n.getData().compareTo(data)==0;
  }
}
```

```
package sll;

public class SLL {
  Node head;
  int size;
  public SLL() {}

  public SLL(Node head) {
    this.head = head;
  }

  public void insert(Node n1, Node n2) {
    if (n1 == null) {
      throw new NullPointerException("n1
        is null");
    }
    if (head == null) {
      head = n1; size++;
      return;
    }
    if (n2 == null) {
      throw new NullPointerException("n2
        is null");
    }
    if (n2 == head) {
      n1.setNext(n2);
      head = n1; size++;
      return;
    }
    Node n = new Node();
    n.data = n2.getData();
    n.next = n2.getNext();
    n2.setData(n1.getData());
    n2.setNext(n); size++;
  }
```

```
  public Node find(String attr, String
       value) {
    Node p=head;
    while (p!=null) {
      if (p.data.compareTo(value)==0)
        return p;
      p=p.next;
    }
    return null;
  }
  public boolean remove(Node n) {
    if (n==null)
      throw new NullPointerException
        ("Node to remove is null.");
    if (head==null)
      throw new NullPointerException
        ("Linked list is empty");
    if (head==n) {
      if (head.getNext()==null) {
        head=null;
      } else {
        head.setData
       (head.getNext().getData());
        head.setNext
       (head.getNext().getNext());
      }
      size--;
      return true;
    }
    if (n.getNext()==null) {
      Node p=head;
      while (p.getNext()!=n)
        p=p.getNext();
      p.setNext(null); size--;
      return true;
    }
    n.setData(n.getNext().getData());
    n.setNext(n.getNext().getNext());
    size--;
    return true;
  }
  public int size() {
    return size;
  }
  public String toString() {
    Node p = head;
    String r = "";
    while (p != null) {
      r += p.getData() + " ";
      p = p.next;
    }
    return r;
  }
}
```

Figure 20.4: An implementation of a singly-linked list

	1	2	3	4	5
o1==null	Y	-	N	N	N
o2==null	-	Y	N	N	N
o1==o2	-	-	Y	N	N
sll is an empty list	-	-	-	Y	N
Rule Count	8	4	2	1	1
null pointer exception	X	X			
illegal operation exception			X		
resulting list contains only o1				X	
resulting list contains o1 followed by o2					X

Figure 20.5: Cause-effect decision table for insert to linked list

```
package sll;

import junit.framework.*;
import junit.swingui.TestRunner;

public class InsertCauseEffectTest
    extends TestCase {
  protected SLL cut;
  public InsertCauseEffectTest() {
    super();
  }
  public InsertCauseEffectTest(String
       name) {
    super(name);
  }
  public void setUp() {
    cut=new SLL();
  }
  public void tearDown() {
    // do nothing
  }
  public void testNullExistingNode() {
    boolean nullPointerException=false;
    try {
      Node n1=new Node("Node to insert."),
        n2=null;
      cut.insert(n1, n2);
    } catch (NullPointerException e) {
      nullPointerException=true;
    }
    assertTrue("Failed to catch null
         pointer exception.",
        nullPointerException);
  }
  public void testNullNode2Insert() {
    // similar to testNullExistingNode
  }
  public void testInsertSameNode() {
    // similar to testNullExistingNode
  }
  public void testInsertIntoEmptyList() {
    Node n=new Node("Node 1");
    cut.insert(n, null);
    assertTrue("Fail to insert into an
        empty list.",
       cut.size()==1 &&
       cut.find("data", "Node 1") != null
       && cut.find("data", "Node
       1").getData().
       compareTo("Node 1") == 0);
  }
  public static void main(String[] args)
  {
    new TestRunner().start(new String[] {
       "sll.InsertCauseEffectTest"});
  }
}
```

Figure 20.6: JUnit test cases for insert into a linked list

singly-linked list is illustrated as shown in Figure 20.7. On the left of the figure, the flow graph for the insert function is shown. On the right of the figure is the five test cases. Implementation in JUnit is omitted.

20.7 Repeat the last exercise, but instead of using basis path testing, apply the ClassBench state testing as described in Section 20.7.2.

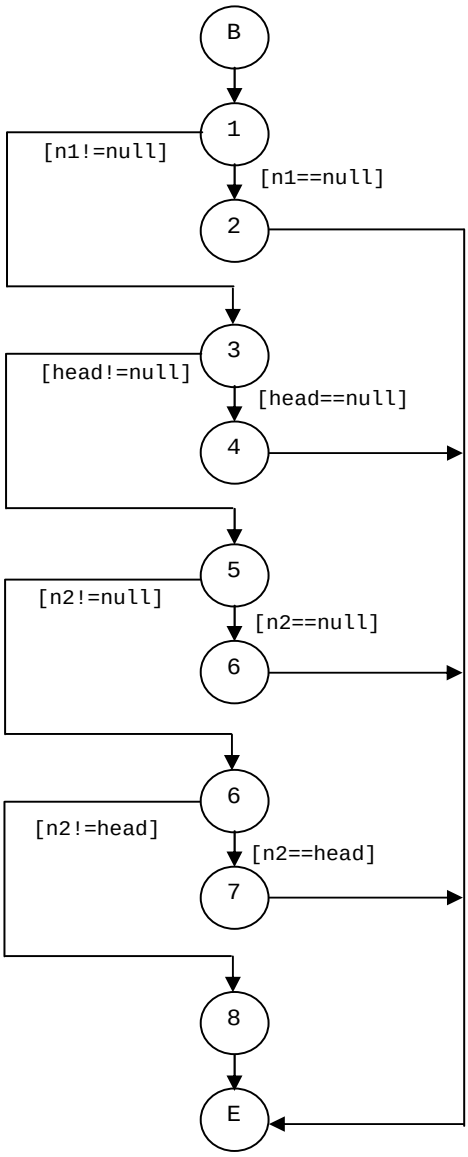
Solution. To apply the ClassBench approach to testing the singly linked list, one constructs a state diagram as shown in Figure 20.8. After calling the constructor, the state of the linked list is in the EMPTY state. The properties of the empty state is checked: size() should return 0, and find(...) should return null for all attribute name and value pairs.

One then inserts Nodes n9, n8, ..., n0 into the linked list, resulting in the ALL state, assume that n0, n1, ..., n9 represent all the elements. Properties of the ALL state is checked by invoking size() and find(...). Finally, one removes all these elements and checks the resulting state.

Figure 20.9 shows the JUnit test cases derived from the test model in Figure 20.8. The state pattern is used to provide flexibility as well as reduce complexity of the implementation of the test cases.

20.8 Draw and fill the entries of a table that summarizes the test methods presented in this chapter. The table has one row for each of the test methods: equivalence partition, boundary value analysis, cause-effect, basis path, data flow (intraprocedural only), use caseC based, ClassBench, web testing. The columns are:

1. Method, which shows the test methods presented in this chapter.
2. Test Model, which briefly describes what the test model is for the test method.
3. Test Case Generation, which briefly describes how the test cases are derived.



Test Case	Input	Expected Outcome	Actual Outcome	Pass/Fail
1	n1==null	null pointer exception		
2	n1!=null && head==null	a list containing only n1		
3	n1!=null && head!=null && n2==null	null pointer exception		
4	n1!=null && head!=null && n2!=null && n2==head	a list containing n1 followed by n2		
5	a list with at least two elements n0 and n2, and Node n1!=null	a list with n0, n1, and n2 in that order.		

Figure 20.7: Basis path testing for inserting into the linked list

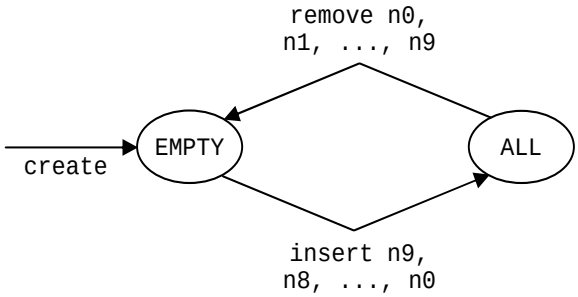


Figure 20.8: Singly-linked list test model

```
package classbench;
import junit.framework.*;
import sll.*;
public class State extends Assert {
  protected static SLL cut;
  public State() {
    super();
  }
  public State insertAll() {
    return this;
  }
  public State removeAll() {
    return this;
  }

package classbench;
import sll.*;
public class Empty
    extends State {
  public Empty(SLL cut) {
    super();
    State.cut=cut;
    check();
  }
  public void check() {
    assertEquals("Size of an empty list
        is not 0", 0, cut.size());
    for(int i=0; i<10; i++) {
      assertNull("Empty list is not
       empty:", cut.find("data",
       "Node "+i));
    }
  }
  public State insertAll() {
    Node n=new Node("Node 9");
    cut.insert(n, null);
    for(int i=8; i>=0; i--) {
      Node ni=new Node("Node "+i);
      cut.insert(ni, n);
      n=ni;
    }
    return new All();
  }
}

package classbench;
import sll.*;
public class All
    extends State {
  public All() {
    super();
    check();
  }
  public void check() {
    assertEquals("cut.size != 10", 10,
        cut.size());
    for(int i=0; i<10; i++) {
      assertNotNull("Node "+i+" not found
        in ALL-state.",
          cut.find("data", "Node "+i));
    }
  }
  public State removeAll() {
    for(int i=0; i<10; i++) {
      Node n=cut.find("data", "Node "+i);
      cut.remove(n);
    }
    return new Empty(cut);
  }
}

package classbench;

import junit.framework.*;
import sll.*;
import junit.swingui.TestRunner;

public class ClassBenchTest
    extends TestCase {
  protected SLL cut;
  private State state;
  public ClassBenchTest() {
    super();
  }
  public ClassBenchTest(String name) {
    super(name);
  }
  public void setUp() {
    cut=new SLL();
  }
  public void testPath() {
    state=new Empty(cut);
    state=state.insertAll();
    state=state.removeAll();
  }
  public static void main(String[] args){
    new TestRunner().start(new String[]
      {"classbench.ClassBenchTest"});
  }
}
```

Figure 20.9: ClassBench JUnit test cases for the singly-linked list

4. Test Coverage(s), which briefly specifies the applicable test coverage criteria.

Solution. Figure 20.10 shows the solution.

20.9 Prove that the three approaches to compute the cyclomatic complexity are equivalent. Hint: Use mathematical induction. In addition, proof by contradiction can be used.

Solution. Omitted.

	Method	Model	Test Generation	Coverage
equivalence partition	Black-box	Equivalence classes induced by application-dependent equivalence relations.	Randomly pick one test from each equivalence class.	• input domains • output domains • equivalence classes.
boundary value	Black-box	(Same as above)	Select tests at the boundaries of the equivalence classes.	(Same as above)
cause-effect	Black-box	Decision table	Generate tests from the rules of the decision table.	• conditions • rules of the decision table
basis path	White-box	Flow graph	Generate one test for paths of the flow graph.	• Statement • Edge • Basis path
data flow	White-box	Flow graph with nodes annotated by variable define/use expressions.	Tests are generated to exercise variable define-use paths.	• All defines • All uses • All define, some use • All define-use paths
use case-based	Black-box	• Expanded use cases describing actor-system interaction. Actor inputs are identified from nontrivial steps. • Table showing combinations of actor input values and expected outcomes.	Each row of the table generates a test.	• Use case • Nontrivial steps
ClassBench	Black-box	State diagram in which nodes represent states of the class under test (CUT), and edges represent execution of methods of the CUT.	A test case is a path from an initial node to some node.	• Node • Edge • Path
web testing	White-box	UML class diagram representing web documents and their relationships for the web application to be tested.	• Static analysis • Data flow testing • Request-response testing	Similar to data flow testing, and use case-based testing

Figure 20.10: Summary of test methods

Part VII

MAINTENANCE AND CONFIGURATION MANAGEMENT

Chapter 21

Software Maintenance

21.1 Collect more than five articles from refereed journals, magazines, and conference proceedings. The articles must be about industry lessons learned or case studies regarding reengineering, restructuring, or reorganizing a software system. Write a survey article about these project experiences. The survey article should discuss the factors that cause the change, types of maintenance, the change process, reverse engineering, reengineering, patterns or architectural styles applied, and tools used by the projects. Limit the article to no more than 10 pages or as instructed by the instructor.

Solution. There are many such articles. The solutions submitted by the students will differ depending on the articles used by the students. Grading of this homework should be relative. The grader could read the solutions and classify them into 3-5 categories according to their answers to this homework question. The solutions in each category are then examined more carefully and a score is given. Moving a solution from one category to another is common.

21.2 Collect more than five articles as in exercise 21.1. For this exercise, suggest and discuss your improvements to the reengineering projects described in the articles. For example, a project could have used a reverse-engineering or metrics tool, but it did not. A certain pattern could

be applied to improve the system, but the project did not. Write a report to describe your improvement suggestions. Limit the report to no more than five pages or as designated by the instructor.

Solution. See grading suggestion described for the previous exercise.

21.3 Describe how you would use a metrics tool to identify places that need improvement. Assume that you have the tool that can compute all the metrics and provide the capabilities you want.

Solution. Answers to this exercise can vary greatly depending on the metrics and tools selected by the students. See the grading suggestion presented for Exercise 21.1.

21.4 Assume that right after you graduate, you are hired by the maintenance team of a financial company. Moreover, you are assigned to maintain an online security (e.g., stocks, mutual funds, exchange-traded funds, etc.) trading software. Describe and explain the responsibilities of this position. State the assumptions you wish to make.

Solution. See grading suggestion presented for above exercises.

21.5 This exercise is a continuation of the previous exercise (i.e., exercise 21.4). For this exercise, describe how you would fulfill your responsibilities to optimize your job performance. Describe also how you would obtain job satisfaction from doing the work. Limit the article to no more than five pages or as set by the instructor.

Solution. See grading suggestion presented for above exercises.

21.6 Suppose you are hired by an IT consulting company to work on a major reengineering project or a large, complex object-oriented software system. For this exercise, you have the freedom to assume the application domain and specific application of the software system. Assume that your team has 20 members. The members may work in groups. Write a brief article to

Facade	Mediator
Similarities 1. Simplify client interface and interaction with a number of components. 2. The client interacts with the facade, which interacts with the components. 3. Decouple the client from the components. 4. Facilitate reuse of the components. 5. A useful pattern for software maintenance.	1. Simplify interfaces and interaction between interacting components. 2. Each component interacts only with the mediator, which interacts with the other components. 3. Decouple the components from each other. 4. Facilitate reuse of any of the components. 5. A useful pattern for software maintenance.
Differences 1. Simplify the "external" interface and interaction. 2. The client can still access to any of the components.	1. Simplify the "internal" interfaces and interaction. 2. It is not desirable for a component to interact with another component.

Figure 21.1: Similarities and differences between facade and mediator

do the following:

1. Describe how the project would proceed.
2. Describe which of the tasks you would like to be assigned to you, and why.
3. Assume that you are assigned the tasks you would like. Describe how you would carry out the tasks.

Solution. See grading suggestion for above exercises.

21.7 Discuss the similarities and differences between the patterns in each of the following pairs. Describe a unique situation in which one of the patterns should be applied and the other should not be applied.

1. Facade and mediator
2. Builder and facade
3. Observer and mediator
4. Facade and proxy.

Solution. Figures 21.1 through 21.4 summarize the similarities and differences.

Builder	Facade
Similarities	
• The construct() method of a concrete supervisor simplifies the client interface and interaction with a concrete builder.	• It provides an application-dependent client interface to simplifies the interaction with a set of components for the client.
Differences	
• Builder is classified as a creational pattern. • Changing the supervisor changes the process. • A concrete supervisor interacts with only one builder during a "construction" process.	• Facade is a structural pattern. • The sequence of interactions with the components is fixed. • The facade interacts with several components.

Figure 21.2: Similarities and differences between builder and facade

Observer	Mediator
Similarities	
• It is a behavioral pattern. • It defines a one-to-many relationship – one observable works with many observers. • The observable notifies the observers, which usually do not know, and do not work with, each other. • The observers may update the observable. • The observable and observers can be reused easily.	• It is a behavioral pattern. • It defines a one-to-many relationship – one mediator works with many components. • The mediator works with the components, which may not know, and should not work directly with, each other. • The components interact with the mediator. • It facilitates reuse of the components.
Differences	
• It decouples an event source from event handlers. • It simplifies event handling. • Observers can be added or removed dynamically, and without affecting the observable. • Inconsistent update to the observable may occur.	• It decouples interacting components. • It simplifies component interfacing and interaction. • Components cannot be added or removed dynamically without affecting the mediator. • The behavior of the mediator is often modeled by a state machine.

Figure 21.3: Similarities and differences between observer and mediator

Facade	Proxy
Similarities	
• It is a structural pattern. • In a less strict sense, the facade object "encapsulates" and "hides" a group of components from the client.	• It is a structural pattern. • The proxy object encapsulates and hides the real object.
Differences	
• It simplifies client interface and interaction with a group of components. • A client can still access to the components.	• It serves as a placeholder, which adds functionality to implement one of the four types of proxies. • No client can access to the real object.

Figure 21.4: Similarities and differences between facade and proxy

Chapter 22

Software Configuration Management

22.1 Describe, with examples, how to use the model in Figure 22.3 to represent the baselines and the associated baseline configuration items in Figure 22.1. Hint: For this exercise, you do not need to construct the complete model; it is enough to show just how to represent a couple of baselines and a few associated configuration items.

Solution. Figure 22.1 shows a partial model for the baselines and SCIs in Figure 22.1 of the textbook.

22.2 Suppose that you work on a project to develop a library information system using the conventional waterfall process. Define the baselines for the project, identify and specify the configuration items for the baselines.

Solution. The baselines and SCIs for each of the baselines are:

Software requirements baseline. Software requirements specification, domain model, use cases, preliminary user's manual, acceptance test plan, software project plan.

Design baseline. Architectural design specification, detailed design specification, integration test plan.

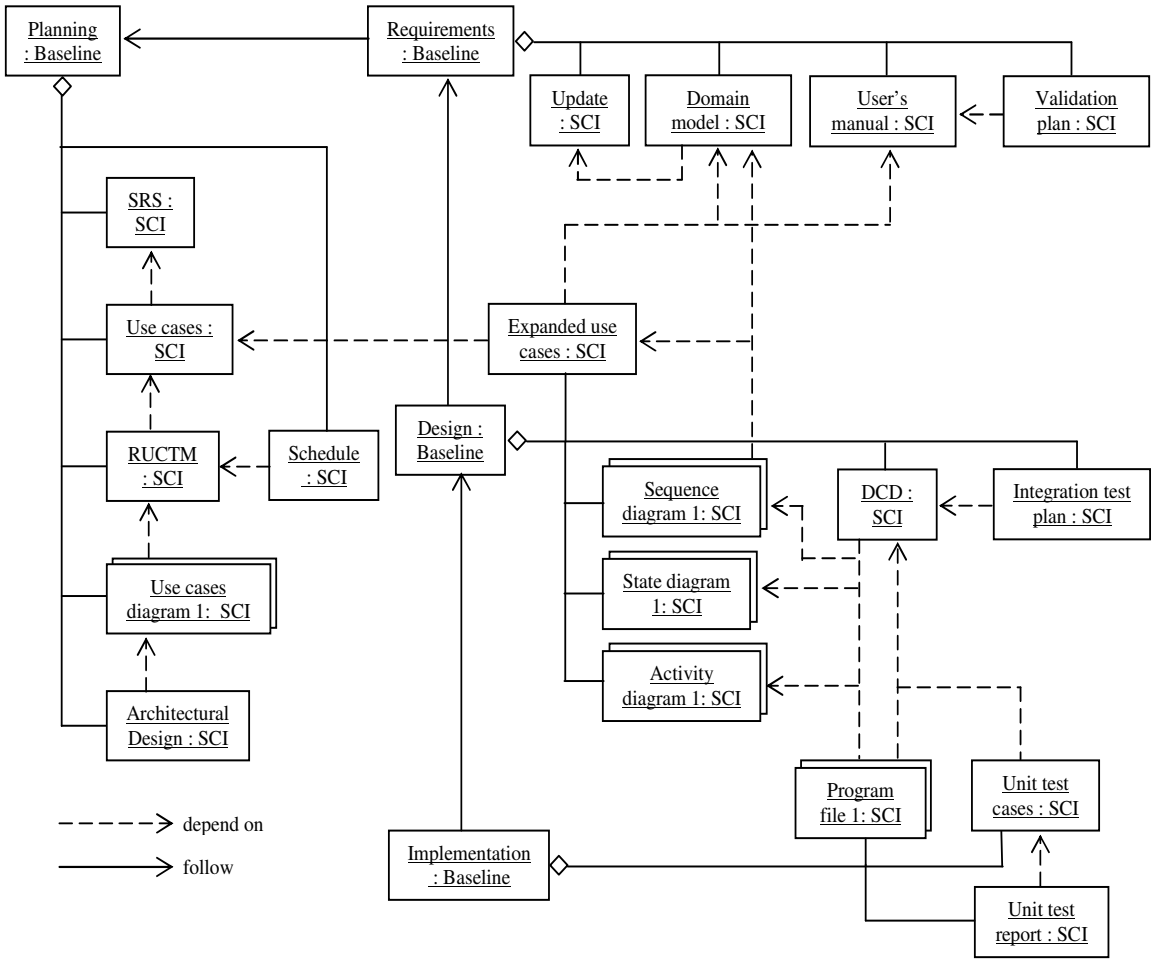


Figure 22.1: Some baselines and SCIs shown in Figure 22.1 of textbook

Implementation baseline. Program listing, unit test cases, test report.

Integration and testing baseline. Integration test cases, integration test report.

Acceptance testing baseline. Acceptance test cases, acceptance test report, as-built user's manual.

Deployment baseline. Installation manual, deployment report.

22.3 Suppose that during the implementation phase of the library information system project, a fatal design flaw is discovered in the architectural design. The architectural design must be modified. You are required to coordinate this activity. Therefore, you are required to write a proposal that addresses the following issues:

a Engineering change proposal (ECP). What is an ECP, who should be responsible for writing the ECP, and what should be included in the ECP for this change?

b ECP evaluation board. What is the name of the board or committee that will review and evaluate the ECP? What are the responsibilities of this organizational unit? Who are the possible members of this organizational unit? What are the possible outcomes of the review and evaluation process, and why?

c Post-evaluation. What are the possible actions to take place after the ECP is reviewed and evaluated, taking into account the possible outcomes of the ECP evaluation process?

Solution. The following answers the questions:

a **Engineering change proposal (ECP)**. The ECP is a report written by a technical staff who initiates the change to one or more SCIs. It describes the changes, why the changes are required, impact to other SCIs, and efforts and costs to implement the changes, as well as impact on the project schedule.

For the libray information system (LIS) case, the system analyst, or chief software architect, may be responsible for writing the ECP. Changes to the architectural design may affect detailed design specifications such as algorithm specifications, subsystems and components that are changed in the architectural design, program files that implement implement these subsystems and components, as well unit test cases and integration test plan. Unit test reports that have been produced may need to be modified accordingly. These need to be described in the ECP.

Effort, costs and time required to change the architectural design and modify the SCIs affected must be estimated. The system analyst or chief software architect may need to meet with developers of the SCIs that are affected by the change to obtain these estimates. He also needs to discuss with the project manager to assess impact and costs to project schedule. These efforts, costs and time must be itemized according to the changes to the architectural design and SCIs affected. These and impact to project schedule must be described in the ECP.

b **ECP evaluation board.** The name of the board is *Configuration Change Control Board (CCCB)*. The CCCB consists of representatives from various groups. In particular, it should include representatives from groups with SCIs affected by the proposed changes. The CCCB reviews the ECP and assess the cost-benefits as well as risk of the proposed changes, and approves or disapproves the proposed changes. The possible outcomes are that the ECP is accepted, rejected, returned for revision, and partially accepted (that is, some of the proposed changes are accepted and some are rejected).

c *Post-evaluation.* If the ECP is accepted or partially accepted, then the system analyst or chief software architect modifies the architectural design as described in the ECP and the decision by the CCCB. The affected SCIs are also modified accordingly by the

respective developers or staff members. If the ECP is returned for revision, then it is modified and resubmited. The ECP is archived for future reference if it is accepted, partially accepted, or rejected.

22.4 Do the same as in exercise 22.2, except that the project uses one of the agile methods presented in Chapter 2.

Solution. This solution chooses Feature Driven Development (FDD). The baselines and associated SCIs are:

Develop overall model baseline. The SCIs are: (1) overall system model, (2) subsystems and components, (3) group object models, (4) overall object model, (5) review reports.

Build a feature list baseline. The SCIs are: (1) List of business activities, (2) feature list, (3) review report.

Plan by feature baseline. The SCIs are: (1) assignment of features to chief programmers, (2) assignment of classes to developers, (3) project schedule.

Design by feature baseline. The SCIs are: (1) sequence diagrams, (2) class-diagram and updates, (3) design alternatives, (4) design inspection, and review reports.

Build by feature baseline. The SCIs are: (1) code listings, (2) code inspection reports, (3) unit test cases, (4) unit test reports.

Deploy by feature baseline. Possible SCIs are: (1) user's manual, (2) installation manual, (3) installation report.

22.5 Identify all possible software configuration items for each of the agile methods described in Chapter 2

Solution. Omitted.

Part VIII

PROJECT MANAGEMENT AND SOFTWARE SECURITY

Chapter 23

Software Project Management

23.1 Discuss in a brief article the pros and cons of the different team structures presented in this chapter. Discuss also their pros and cons with respect to agile projects.

Solution. Figure 23.1 summarizes the pros and cons of the three team structures. Agile projects value individual and interaction. Moreover, agile projects advocate that team members are empowered to make decisions. Therefore, the egoless team structure is the most appropriate for small, agile projects. For medium to large agile projects that require multiple teams to collaborate and work simultaneously, the hierarchical team structure is more appropriate. The chief-programmer team structure places more weight on the chief programmer, who is usually very technically competent, and familiar with the application domain. This makes the chief-programmer structure more suitable for agile projects that require predicatability.

23.2 Apply the COCOMO II Application Composition Model described in Section 23.2.2 to estimate the effort required to develop the Add Program use case for a Study Abroad web application for the Office of International Education (OIE) of a university. Assume that the object point productivity is nominal and no reuse of existing components is expected. The

	Pros	Cons
Egoless team	• Value individuals and interaction • Suitable for brainstorming • Effective for solving challenging problems • Suitable for small agile projects	• It may spend too much time on discussion; and hence, it may not be productive. • Its communication overhead could be high. • It may result in gridlock if the team is divided. • A good design or suggestion may not be adopted if it is not accepted by the majority of the team.
Chief-programmer team	• It reduces communication overhead between the team members; and hence, it could be more productive. • The other team members may be less experienced developers – this may reduce the project costs. • It may be more suitable for projects that require predictability. • If the chief programmer is really good, this team structure may work very well.	• The chief programmer must possess many skills; and hence, he could be hard to find and expensive to hire and keep. • The project may fail if the chief programmer is wrong, or he resigns in the middle of the project.
Hierarchical team	• Its communication overhead is moderate. • It tends to produce better designs because design decisions are made by experienced developers. • It avoids the single-failure point problem of the chief-programmer team structure. • It is useful for medium to large projects that require multiple teams to collaborate and work simultaneously.	• The senior engineers may compete for promotion, team members, and other resources. This may impact the project negatively. • A senior engineer may not be able to communicate, or work effectively with the project manager and the junior engineers. This may affect the overall project negatively.

Figure 23.1: Pros and cons of different team structures

Add Program use case allows an OIE staff to add an overseas exchange program, such as French Language and Culture. This use case needs to display three web pages and a message as follows:

a Initially, it displays a Welcome page, which consists of three frames. The main frame shows a description of the Study Abroad program. The top frame shows the OIE logo along with a row of buttons, such as OIE Home, About Study Abroad program, Contact Us, etc. The left frame shows a list of three menu items: Program Management, User Management, and System Settings. The Welcome page is displayed after an OIE staff member successfully logs into the system.

b To add a program, the OIE staff clicks the Program Management item, which expands the item and shows a list of four menu items under the expanded item — Add Program, Update Program, Delete Program, and Import Programs.

Step	#screens	#reports	#3GL Comp.	
1. Count the number of screens, reports, and 3GL components that will comprise the application.	4	0	0	
2. Determine the complexity levels of each of the screens and reports using Figure 23.2(a).	3 of the screens are simple, 1 is difficult.			
3. Look up the complexity weights for each of the screens, reports, and 3GL components from Figure 23.2(b).	Complexity weight=1 for a simple screen, it is 3 for a difficult screen.			
				Object point
4. Add the weighted counts of screens, reports and 3GL components to produce one number, called the Object Point (OP) count.	3*1 + 1*3			6
				New object point
5. Estimate the percentage of reuse to be achieved, and compute the New Object Points (NOPs) to be developed in the project: NOP = OP × (100 - Reuse)/100	reuse=0			6
				Productivity
6. Determine the object point productivity from Figure 23.2(c), that is, the average of two capabilities shown in Figure 23.2(c).	Nominal=13			13
7. Compute the person-month effort: Effort PM = NOP/PROD				0.46

Figure 23.2: Applying COCOMO Application Composition Model

c The OIE staff clicks the Add Program item. The system displays an Add Program Form in the main frame of the page. The Add Program Form lets the OIE staff enter program information into the various fields. The OIE staff clicks the Submit button to save the program.

d When the Submit button is clicked, the system saves the program into a database and displays a "program is successfully saved" message.

Solution. The Application Composition Model consists of seven steps. This exercise simply applies these steps to the Study Abroad Management System application. There are four web pages, which are counted as screens. Three of these are simple and the one that requires the OIE staff to enter program information is considered difficult. There is no report and 4GL components, and the productivity level is nominal. The spreadsheet shown in Figure 23.2 illustrates how the 0.46 person-month effort is computed.

23.3 Make proper assumptions for the missing information in the Add Program use case description

provided in the previous exercise and apply the function point method to estimate the required effort. Justify your assumptions and effort estimate.

Solution. Figure 23.3 show the assumptions and the function points. That is, the use case requires 67.64 function points.

23.4 Apply COCOMO II Early Design Model to estimate the effort required to develop the following use cases for the Study Abroad web application. Make necessary assumptions including the number of inputs, outputs, and the scale factors, cost drivers and the like. For example, you can assume average/nominal levels for all of the scale factors.

a UC01. Search for Programs (Actor: Web User, System: SAMS) This use case allows a user to search overseas exchange programs using a variety of search criteria such as subject, semester, year, country and region. The system searches the database and displays a list of programs, each of which includes a link to display the detail of the program.

b UC02. Display Program Detail (Actor: Web User, System: SAMS) This use case retrieves and displays the detailed description of a program, selected by clicking the link of the program displayed by the Search for Programs use case, or by giving the program ID or program name.

c UC03. Submit Online Application (Actor: Student, System: SAMS) This use case allows a student to apply to an overseas exchange program. The student fills in an application form and submits it. The system verifies the application, saves it in the database, and sets the status of the application to submitted. The system also sends e-mail to the two faculty members requested by the student to write recommendation letters, and the academic adviser to approve the course equivalency form.

1. Fill the counts for the five function categories in the FP worksheet in Figure 23.1.				Complexity			
2. Circle the complexity in Figure 23.1.		Function Category	Count	Simple	Average	Complex	Count **X** Complexity
3. Compute gross function points GFP = sum of count-i x complexity-i.	1	Number of user input	15	3	4	6	45
	2	Number of user output	1	4	5	7	4
	3	Number of user queries	0	3	4	6	0
	4	Number of data files and relational tables	1	7	10	15	7
	5	# of external interfaces	4	5	7	10	20
						GFP	76
4. Assign a processing complexity PCi.							
a. Does the system require reliable backup and recovery?	1						
b. Are data communications required?	0						
c. Are there distributed processing functions?	0						
d. Is performance critical?	0						
e. Will the system run in an existing, heavily utilized operational environment?	0						
f. Does the system require online data entry?	5						
g. Does the online data entry require the input transaction to be built over multiple screens or operations?	3						
h. Are the master files updated online?	5						
i. Are the inputs, outputs, files, or inquiries complex?	3						
j. Is the internal processing complex?	2						
k. Is the code designed to be reusable?	0						
l. Are conversion and installation included in the design?	0						
m. Is the system designed for multiple installations in different organizations?	0						
n. Is the application designed to facilitate change and ease of use by the user?	5						
5. Compute the processing complexity adjustment PCA = 0.65 + 0.01 * sum of PCi.	0.89						
6. Compute FP = GFP * PCA.	67.64						

Figure 23.3: Applying function point

d UC04. Login (Actor: Student, System: SAMS) This use case allows a registered student to login to the system.

e UC05. Logout (Actor: Student, System: SAMS) This use case allows a student to logout from the system.

f UC06. Edit Online Application (Actor: Student, System: SAMS) This use case allows the student to edit an application that is not yet submitted.

g UC07. Check Application Status (Actor: Student, System: SAMS) This use case allows a student to check the status of an application, such as in-preparation, submitted, under-review, accepted, and rejected.

h UC08. Submit Recommendation (Actor: Faculty, System: SAMS) This use case lets a faculty member submit a recommendation on behalf of a student. The system saves the recommendation in the database.

i UC09. Approve Course Equivalency Form (Actor: Advisor, System: SAMS) This use case lets an academic adviser review and approve course equivalency forms submitted by students when they submit their online applications. Each form specifies the overseas courses that the student plans to use to substitute for the courses of the academic department.

Solution. The first step to apply the Early Design Model is to determine size, which may be obtained directly, or from the function point estimates. Direct estimation is used here because this is simpler. Figure 23.4 shows the estimates of sizes in KLOC for the use cases. The following assumptions are used for determining the scale factors:

- **PREC: experience of the team for this type of project.** PREC=4.96 (low experience).

	KLOC
UC01. Search for Programs	1
UC02. Display Program Detail	0.5
UC03. Submit Online Application	1
UC04. Login	0.3
UC05. Logout	0.1
UC06. Edit Online Application	1
UC07. Check Application Status	0.4
UC08. Submit Recommendation	1
UC09. Approve Course Equivalency Form	1
	6.3

Figure 23.4: Size estimation of use cases

- **FLEX: flexibility given to the team.** FLEX=3.04 (nominal).
- **RESL: extent of design carried out, and risk elimination.** RESL=5.65 (low RESL).
- **TEAM: teamwork.** TEAM=3.29 (nominal).
- **PMAT: process maturity model.** PMAT=6.24 (low).

Next, compute constant b:

$b = 0.91 + 0.01 * (SF1 + ... + SF5) = 0.91 + 0.01 * (4.96 + 3.04 + 5.65 + 3.29 + 6.24) = 1.14$

Assume that all of the cost drivers are 1.00, the effort in person-months is:

Effort $PM = 2.94 * 6.3^{1.14} = 2.94 * 8.15 = 23.96$

23.5 Base on the estimates produced in the previous exercise, perform the following:

a Identify the dependencies among the nine use cases for the Study Abroad application.

b Assume that the team cannot work on a use case until the use cases that it depends on are completed. Assume that the completion of a use case marks a milestone. Produce a PERT chart schedule for developing the use cases.

c Compute the earliest start time and latest completion time for each of the milestones.

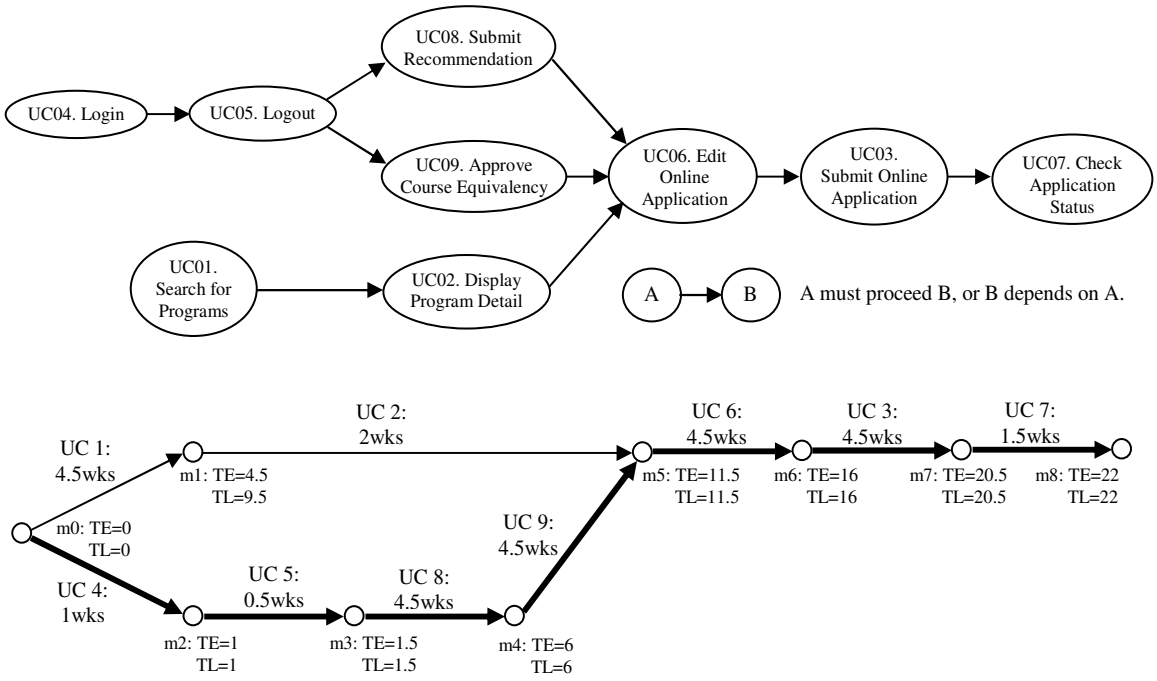


Figure 23.5: Use case dependencies and PERT chart

d Identify the critical path for the project.

e Select a calendar date for the project start and fill in the calendar dates for the milestones.

Solution. Figure 23.5 shows the dependencies between the nine use cases as well as a PERT chart with the earliest start times and latest completion times for the milestones. The darkened path is the critical path. The task durations are hypothetical.

23.6 Form a group of five students and practice agile estimation and planning for the 9 use cases of the Study Abroad application. Produce a brief report describing the process and the results obtained.

Solution. Omitted.

23.7 Identify top five possible risks for the SAMS project and develop risk resolution measures for them. Briefly justify your solution.

Solution. Figure 23.6 shows the result.

23.8 Suppose that a company is at level 1 of the CMMI. Propose a process improvement plan for the company to improve its process to level 5. Make necessary assumptions about the level-1 company.

Solution. Different solutions could be produced by the students, due to the assumptions made and the improvement plan proposed. Grading should be relative — that is, classify the solutions into three to five ranks accoring to various factors including validity of the assumptions, merits of the improvement plan, organization of the solution, readability, grammar, and spelling.

Risks (Ranked Top-Down)	Risk Resolution Measurements	Justification
1. Qualified personnel shortfall.	1. Staffing with top talent, job matching, team building, key personnel agreements, cross training. 2. Proactive recruiting. 3. Keeping a list of qualified expert consultants. 4. Regular developer training.	Shortfall of qualified technical personnel is always a challenge for many software development organizations. Proactive recruiting anticipates the problem of a potential personnel shortfall and starts the recruiting process before the problem actually occurs. A list of qualified expert consultants lets the project seek expert help in case a key technical staff leaves the project. Regular developer training keeps the team up to date. This may help qualified personnel shortfall to a certain extent.
2. Security attacks.	1. Security requirements. 2. Design for security. 3. Apply static analysis tools. 4. Code review and testing to detect security vulnerabilities. 5. Install firewall, and run effective virus scan software.	Security attacks include all kinds of attacks such as identity theft, leaking of private information, Trojan horse, etc. Damages are difficult and costly to repair. Security requirements are considered during the requirements phase – to identify security requirements that are critical to the Office of International Education. Design for security applies security architectural styles and security patterns to satisfy security requirements. Static analysis tools are applied to detect security vulnerabilities. Code review and testing look for security vulnerabilities as well as ensure that the implementation satisfies the security requirements.
3. Requirements misconception.	1. Agile development, and active involvement of users. 2. Prototyping, software demo, and user testing. 3. User survey to assess the extent of fulfillment of requirements.	Agile principles, practices and values are proven techniques for preventing requirements misconception. Seeing is believing – prototyping, software demo and user testing let the users see how the software would work. These, combine with user survey during or after demo and testing, could significantly reduce requirements misconception.
4. Schedule slippery.	1. Agile development, especially agile estimation and agile planning. 2. Daily stand-up meetings to keep track of status closely. 3. Requirements prioritization. 4. Capture requirements at a high level, visual, and lightweight. Good enough is enough.	Agile estimation and agile planning involve the team members in planning and scheduling. These, if practiced properly, could result in a more realistic schedule; and hence, they could reduce the likelihood of schedule slippery. Requirements prioritization lets the team understand the customer's business priorities, and develop and deploy high-priority use cases first. This allows the team to meet schedule by reducing functionalities of low-priority requirements if needed. Daily stand-up meetings let team members exchange status, and solve issues early. This helps meeting the schedule. Capture requirements at a high level, lightweight and visual reduces the time spent in the early stage of the project. This allows the team to focus on delivery of working software to meet schedule.
5. Cost overrun.	1. Agile estimation and agile planning. 2. Requirements prioritization. 3. Value working software over comprehensive documentation. 4. Agile modeling – barely enough modeling.	Agile estimation should produce more accurate effort estimates and hence cost estimates. This lets the management understand better the cost to construct and deliver the software. Agile planning should produce a schedule that is more realistic and match with the ability of the team. This avoid schedule compression, which is a cost factor. Requirements prioritization lets the team focus on high-priority requirements. It may help reduction of gold-plating features that the customer does not ask for. Value working software over comprehensive documentation and agile modeling let the team members focus on delivery of working software. This could effectively reduce project costs.

Figure 23.6: Top five risk items, resolution, and justification

Chapter 24

Software Security

24.1 Identify and formulate security requirements for the online car rental project described in Appendix D.1.

Solution. Some security requirements for the Car Rental System (CRS) are the following:

R 1. The CRS must identify and authenticate all employees and all customers who want to make a reservation of a vehicle.

R 2. The CRS must implement strong password rules.

R 3. The CRS must provide role-based access control as specified by the customer.

R 4. The CRS must provide mechanisms to prevent infections by malicious programs such as computer viruses, worms and Trojan horses to damage the CRS software and stored data.

R 5. The CRS must encrypt all sensitive data including passwords, customer information, and financial transaction information.

R 6. The CRS must ensure that all communications take place over a secured wired or wireless network.

R 7. The CRS must ensure the integrity of stored data including customer information, transaction records, and financial records.

R 8. The CRS must save, backup, and maintain all transaction information required to prevent any party from denying all or part of a transaction.

R 9. The CRS must reject all unauthorized access to customer information.

R 10. The CRS must keep track of all executions of all security functions for security auditing.

R 11. The CRS must be able to recover from a successful security attack.

24.2 Identify five of the most significant misuse cases for the online car rental project. Also specify the abstract, high-level, and expanded misuse cases for these five misuse cases. Hint: These are the same as abstract, high-level, and expanded use cases except that they have a hostile intent.

Solution. Different students may identify different top-five misuse cases. Their solutions do not need to be the same. Grading should be on the validity of the misuse cases, and whether they are significant. Whether they are really the top-five is not important.

The online car rental system (CRS) may have the following misuse cases or attacks. Two of these are refined in Figure 24.1.

a Steal Identity Information. This misuse case may describe an insider attack, in which the attacker has access to the CRS. If the CRS does not encrypt customer information, then the attacker can easily steal customers' identify information.

b Compromise a Default Account. Default administrator accounts and passwords tend to use informative user names and passowrds such as "admin," "Admin," "adminstrator," "Administrator," "admin1," "admin2" as user names and easy-to-remember passwords

MUC 1. Steal Identify Information.
TUCBW the internal attacker login to the CRS or database.
TUCEW the internal attacker successfully steals the identify information.

Actor: Internal Attacker	System: CRS
1. TUCBW the internal attacker login to the CRS or its database.	2. CRS or database indicates successful login to the attacker.
3. Attacker finds the identify information and copies it to his storage media.	4. CRS or database indicates that copying is completed.
5. TUCEW the internal attacker successfully steals the identify information.	

MUC 2. Compromise a Default Account.
Precondition: The external attacker uses an attack software.
TUCBW attacker starts the attack software.
TUCEW attacker sees the user name and password for a default account, or a fail message.

Actor: External Attacker	System: Attack Software
1. TUCBW attacker starts the attack software.	2. Attack software displays available options and requests the attacker to select.
3. Attacker selects "login via a default account," the user name and password files he wants to use, and clicks the OK button.	4. Attack software displays the user name and password of a default account, or a fail message.
5. TUCEW attacker sees the user name and password for a default account, or a fail message.	

Figure 24.1: Some misuse cases for the online car rental system

such as "admin123," "1234," etc. An attacker could try many combinations of such user names and passwords to compromise the CRS. The attacker could run an attack software, which automatically tries to login to the CRS using such combinations. It reports to the attacker if one of the attacks is successful. Implementation of strong password rules could prevent the attacker from success.

c Compromise an Account. This has been described in the textbook (see Example 24.3).

d Unauthorized access. Many unauthorized access scenarios could happen to the CRS. For example, if the role-based control security function is not designed and implemented properly, then a database administrator (DBA) of the CRS could access to the password file or database table. He could obtain customers' passwords if the passwords are not encrypted, or he also obtains the encryption key. The same could happen to a CRS system administrator. In addition, the CSR system administor could create a saleper-

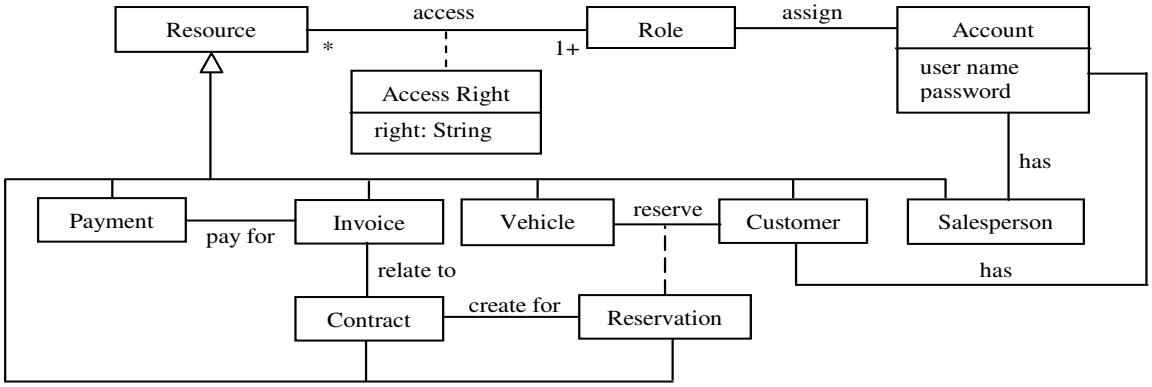


Figure 24.2: Security domain model for the online car rental system

son person account for himself. With this, he could access to customers' information, resulting in an unauthorized access.

e Query injection. Inproper input validation is a source for a query injection attack. The attacker may use an attack software to launch such attacks automatically, using known query injection attack patterns.

24.3 Produce a secure architectural design for the online car rental software. Indicate which secure software design principles and security patterns are applied.

Solution. Figure 24.3 of the textbook shows a secure architecture for the Study Abroad Management System (SAMS). It could be adapted for the online car rental system.

24.4 Produce a domain model for the online car rental application. Include security-related domain concepts, their properties and relationships.

Solution. Figure 24.2 shows a domain model that includes some of security related concepts, their properties and relationships.

24.5 Produce sequence diagrams for the three most useful use cases for the online car rental application. Include and indicate in the sequence diagrams security mechanisms to satisfy

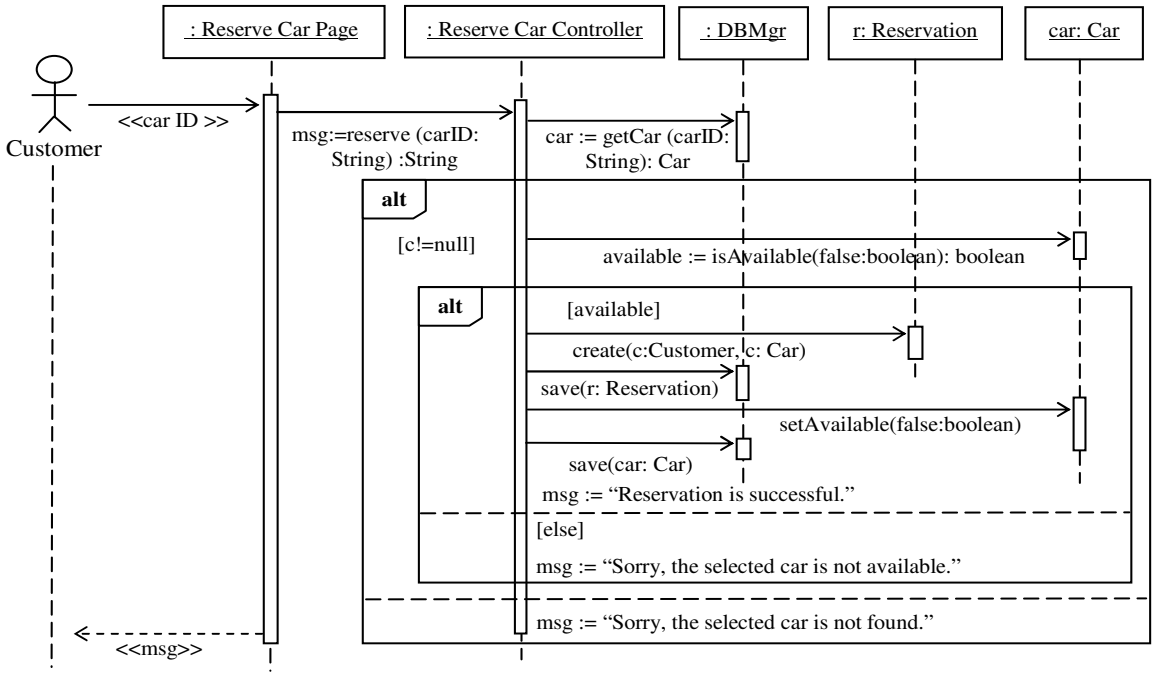


Figure 24.3: Reserve a car sequence diagram

the security requirements formulated previously.

Solution. The three most useful use cases are Login, Search for Cars, and Reserve a Car. To login, the customer must have created an account. For a Login sequence diagram, see Figure 9.3 of the textbook. A precondition for the Reserver a Car use case is that the customer must have logged into the system. Figure 24.3 shows the sequence diagram for the Reserve a Car use case.

24.6 Do the same exercises as above but use the Study Abroad Management System described in Appendix D.3.

Solution. Omitted.

24.7 Do the same exercises as above but use the National Trade Show Service (NTSS) system described in Appendix D.2.

Solution. Omitted.