Chapter 4

1. What are the main **components of the functional model**?
    - List and define each one.
        - Use Cases: A *use case* is a formal way of representing the way a business system interacts with its environment. Essentially, a use case is a high-level overview of the business processes in a business information system. From a practical perspective, use cases represent the entire basis for an object-oriented system.
            - Walkthrough / Role-Playing
        - Activity Diagrams: *Activity diagrams* are typically used to augment our understanding of the business pro- cesses and our use-case model. Technically, an activity diagram can be used for any type of process-modeling activity.
        - *Process models* depict how a business system operates.
    - How do they relate to each other?
        - Use-case diagrams provided a bird's-eye view of the basic functionality of the business pro- cesses contained in the evolving system. Activity diagrams, in a sense, open up the black box of each business process by providing a more-detailed graphical view of the underlying activities that support each business proces

2. What are the **elements of a use case**? (Appendix A)
    - The elements of a use-case diagram include actors, use cases, subject boundaries, and a set of relationships among actors, actors and use cases, and use cases:
    - **Actors** The stick figures on the diagram represent actors. An *actor* is not a specific user but instead is a role that a user can play while interacting with the system. An actor can also represent another system in which the current system interacts.
    - **Association** Use cases are connected to actors through association relationships; these rela- tionships show with which use cases the actors interact (see Figure 4-1). A line drawn from an actor to a use case depicts an association [can be two way [if no arrows] or one way (represented by arrows)].
        - The asterisk means multiciplicity (basically a many-to-many relationship)
    - **Use Case** A use case, depicted by an oval in the UML, is a major process that the system performs and that benefits an actor or actors in some; it is labeled using a descriptive verb–noun phrase.
    - **Subject Boundary** The use cases are enclosed within a *subject boundary,* which is a box that defines the scope of the system and clearly delineates what parts of the diagram are external or internal to it

3. What are the different types of relationships and what do they mean? (Appendix A)

- Association
- Include
- Extend
- Generalization

→ See Appendix A for definitions

4. What are the steps to follow in identifying the major use cases? (Appendix B)
    1. The first step is to review the requirements definition.
    2. The second step is to identify the subject's boundaries. This helps the analyst to identify the scope of the system.
    3. The third step is to identify the primary actors and their goals.
        a. The primary actors involved with the system come from a list of stakeholders and users.
        b. The goals represent the functionality that the system must provide the actor for the system to be a success.

    → Steps 2 and 3 are intertwined. As actors are identified and their goals are uncovered, the boundary of the system will change.
    4. The fourth step is to simply identify the business processes and major use cases. Rather than jumping into one use case and describing it completely at this point, we only want to identify the use cases.
    5. The fifth step is to carefully review the current set of use cases. It may be necessary to split some of them into multiple use cases or merge some of them into a single use case
    6. The trick is to select the right size so that you end up with three to nine use cases in each system

5. What are the different types of use cases?
    - An *overview use case* is used to enable the analyst and user to agree on a high-level over- view of the requirements.
    - A *detail use case* typically documents, as far as possible, all the information needed for the use case
    - An *essential use case* is one that describes only the minimum essential issues necessary to understand the required functionality.
    - A *real use case* goes farther and describes a specific set of steps.

6. What are **activity diagrams** used for?
    - Activity diagrams are used to model the behavior in a business process independent of objects. Activity diagrams can be used to model everything from a high-level business workflow that involves many different use cases, to the details of an individual use case, all the way down to the specific details of an individual method. In a nutshell, activity diagrams can be used to model any type of process.

7. How do they relate to use cases?
    - Use-case diagrams provided a bird's-eye view of the basic functionality of the business pro- cesses contained in the evolving system. Activity diagrams, in a sense, open up the black box of each business process by providing a more-detailed graphical view of the underlying activities that support each business process.

8. What are the elements of an activity diagram? (Appendix C)
    - **Actions and Activities** *Actions* and *activities* are performed for some specific business reason. Actions and activities can represent manual or computerized behavior.
    - The only difference between an action and an activity is that an activity can be decomposed further into a set of activities and/or actions, whereas an action represents a simple non-decomposable piece of the overall behavior being modeled.
    - **Object Nodes** Activities and actions typically modify or transform objects. *Object nodes* model these objects in an activity diagram. Object nodes are portrayed in an activity diagram as rectangles
    - **Control Flows and Object Flows** There are two different types of flows in activity diagrams: control and object.
    - *Control flows* model the paths of execution through a business process. A control flow is portrayed as a solid line with an arrowhead on it showing the direction of flow.
    - *Object flows* model the flow of objects through a business process
    - **Control Nodes** There are seven different types of *control nodes* in an activity diagram: initial, final-activity, final-flow, decision, merge, fork, and join (see Appendix C).
    - **Swimlanes.**

9. What are **use case descriptions used for**?
    - Use-case descriptions provide a means to more fully document the different aspects of each individual use case. Use-case descriptions contain all the information needed to document the functionality of the business processes.

10. How do use case descriptions relate to a use case diagram?
    - The use-case descriptions are based on the identified requirements, use-case diagram, and the activity diagram descriptions of the business processes.
    - Use cases provide a bird's-eye view of the business processes contained in the evolving system. The use-case diagram depicts the communication path between the actors and the system.

11. What are the elements of a use case description? (Appendix D)

- **Overview Information** The overview information identifies the use case and provides basic background information about the use case. The *use-case name* should be a verb–noun phrase (e.g., Make Old Patient Appt). The *use-case ID number* provides a unique way to find every use case and also enables the team to trace design decisions back to a specific requirement.
- **Relationships** Use-case relationships explain how the use case is related to other use cases and users. There are four basic types of *relationships:* association, extend, include, and generalization.
- **Flow of Events** Finally, the individual steps within the business process are described. Three different categories of steps, or *flows of events,* can be documented: normal flow of events, subflows, and alternative, or exceptional, flows:
  - The *normal flow of events* includes only steps that normally are executed in a use case.
  - In some cases, the normal flow of events should be decomposed into a set of *subflows* to keep the normal flow of events as simple as possible.
  - *Alternative or exceptional flows* are ones that do happen but are not considered to be the norm.
- **Optional Characteristics** Other characteristics of use cases can be documented by use-case descriptions.

12. What are **verifications and validations** for the functional model?
    - A *walkthrough* is essentially a peer review of a product.
      - In the case of the functional models, a walkthrough is a review of the different models and diagrams created during functional modeling.
    -
    -

13. Why do you do them?
    - The purpose of a walkthrough is to thoroughly *test* the fidelity of the functional models to the functional requirements and to ensure that the models are consistent

14. What does it involve?
    - *Walkthroughs*

15. When do you do them?
    -

Appendix A

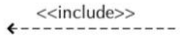
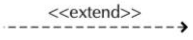
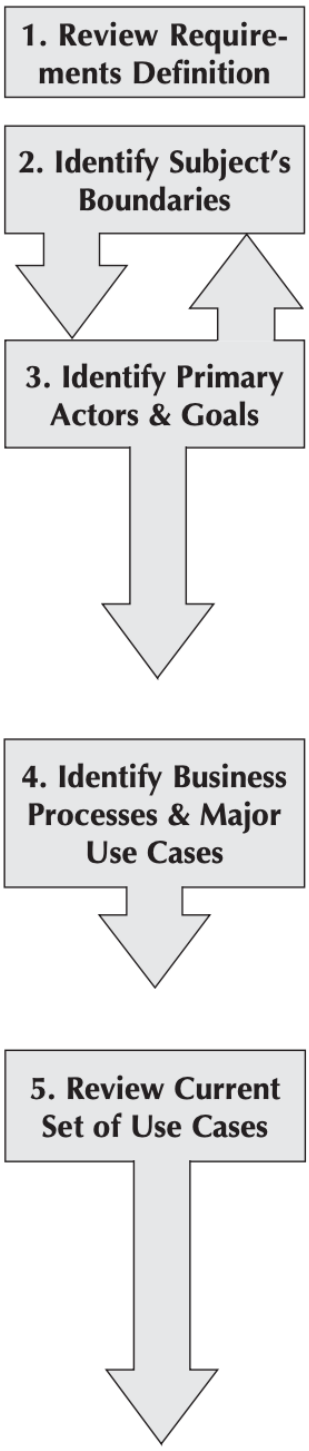
An actor: ■ Is a person or system that derives benefit from and is external to the subject. ■ Is depicted as either a stick figure (default) or, if a nonhuman actor is involved, a rectangle with <<actor>> in it (alternative). ■ Is labeled with its role. ■ Can be associated with other actors using a specialization/superclass association, denoted by an arrow with a hollow arrowhead. ■ Is placed outside the subject boundary.	 Actor/Role 
A use case: ■ Represents a major piece of system functionality. ■ Can extend another use case. ■ Can include another use case. ■ Is placed inside the system boundary. ■ Is labeled with a descriptive verb–noun phrase.	
A subject boundary: ■ Includes the name of the subject inside or on top. ■ Represents the scope of the subject, e.g., a system or an individual business process.	
An association relationship: ■ Links an actor with the use case(s) with which it interacts.	
An include relationship: ■ Represents the inclusion of the functionality of one use case within another. ■ Has an arrow drawn from the base use case to the used use case.	
An extend relationship: ■ Represents the extension of the use case to include optional behavior. ■ Has an arrow drawn from the extension use case to the base use case.	
A generalization relationship: ■ Represents a specialized use case to a more generalized one. ■ Has an arrow drawn from the specialized use case to the base use case.	

FIGURE 4-1 Syntax for Use-Case Diagram

Appendix B


1. Review Require-ments Definition
2. Identify Subject's Boundaries
3. Identify Primary Actors & Goals
4. Identify Business Processes & Major Use Cases
5. Review Current Set of Use Cases

Appendix C (Elements of an Activity Diagram)

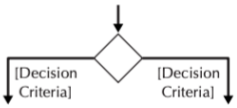
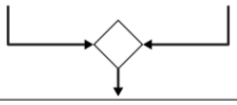
An action: ■ Is a simple, nondecomposable piece of behavior. ■ Is labeled by its name.	
An activity: ■ Is used to represent a set of actions. ■ Is labeled by its name.	
An object node: ■ Is used to represent an object that is connected to a set of object flows. ■ Is labeled by its class name.	
A control flow: ■ Shows the sequence of execution.	
An object flow: ■ Shows the flow of an object from one activity (or action) to another activity (or action).	
An initial node: ■ Portrays the beginning of a set of actions or activities.	
A final-activity node: ■ Is used to stop all control flows and object flows in an activity (or action).	
A final-flow node: ■ Is used to stop a specific control flow or object flow.	
A decision node: ■ Is used to represent a test condition to ensure that the control flow or object flow only goes down one path. ■ Is labeled with the decision criteria to continue down the specific path.	
A merge node: ■ Is used to bring back together different decision paths that were created using a decision node.	
A fork node: Is used to split behavior into a set of parallel or concurrent flows of activities (or actions)	
A join node: Is used to bring back together a set of parallel or concurrent flows of activities (or actions)	
A swimlane: Is used to break up an activity diagram into rows and columns to assign the individual activities (or actions) to the individuals or objects that are responsible for executing the activity (or action) Is labeled with the name of the individual or object responsible	

FIGURE 4-7 Syntax for an Activity Diagram

Appendix D

Use Case Name: Make Old Patient Appt	ID: 2	Importance Level: Low
Primary Actor: Old Patient	Use Case Type: Detail, Essential	

Stakeholders and Interests:
Old Patient – wants to make, change, or cancel an appointment
Doctor – wants to ensure patient's needs are met in a timely manner

Brief Description: This use case describes how we make an appointment as well as changing or canceling an appointment for a previously seen patient.

Trigger: Patient calls and asks for a new appointment or asks to cancel or change an existing appointment

Type: External

Overview

Relationships:
Association: Old Patient
Include:
Extend: Update Patient Information
Generalization: Manage Appointments

Relationships

Normal Flow of Events:
1. The Patient contacts the office regarding an appointment.
2. The Patient provides the Receptionist with his or her name and address.
3. If the Patient's information has changed
   Execute the Update Patient Information use case.
4. If the Patient's payment arrangements has changed
   Execute the Make Payments Arrangements use case.
5. The Receptionist asks Patient if he or she would like to make a new appointment, cancel an existing appointment, or change an existing appointment.
   If the patient wants to make a new appointment,
   the S-1: new appointment subflow is performed.
   If the patient wants to cancel an existing appointment,
   the S-2: cancel appointment subflow is performed.
   If the patient wants to change an existing appointment,
   the S-3: change appointment subflow is performed.
6. The Receptionist provides the results of the transaction to the Patient.

SubFlows:
S-1: New Appointment
1. The Receptionist asks the Patient for possible appointment times.
2. The Receptionist matches the Patient's desired appointment times with available dates and times and schedules the new appointment.

S-2: Cancel Appointment
1. The Receptionist asks the Patient for the old appointment time.
2. The Receptionist finds the current appointment in the appointment file and cancels it.

S-3: Change Appointment
1. The Receptionist performs the S-2: cancel appointment subflow.
2. The Receptionist performs the S-1: new appointment subflow.

Flow of Events

Alternate/Exceptional Flows:
S-1, 2a1: The Receptionist proposes some alternative appointment times based on what is available in the appointment schedule.
S-1, 2a2: The Patient chooses one of the proposed times or decides not to make an appointment.

FIGURE 4-13 Sample Use-Case Description

Chapter 5

This is a very important chapter

1. What are the components of the structural model?
    - List and explain each one
    - Typically, structural models are depicted using CRC cards, class diagrams, and, in some cases, object diagrams
    - Also include those which are not part of the UML diagrams but are mentioned in your book.
    - *Conceptual model*, which shows the logical organization of the objects without
    - Analysts evolve the conceptual structural model into a design model that reflects how the objects will be organized in databases and software

2. What are classes, attributes, and operations?
    - *Class* is a general template that we use to create specific instances, or *objects*, in the problem domain
        i. *Abstract classes* do not actually exist in the real world; they are simply useful abstractions.
        ii. *Concrete classes* are used to create objects.
    - An *attribute* of an analysis class represents a piece of information that is relevant to the description of the class within the application domain of the problem being investigated.
        i. Must be primitive/atomic types.
    - The behavior of an analysis class is defined in an *operation* or service. In later phases, the operations are converted to *methods.*

3. What is the difference between a class and an object?
    - Class is a general template; an object is a specific instance of that template.

4. What are the different types of relationships between classes?
    - **Generalization Relationships** The generalization abstraction enables the analyst to create classes that inherit attributes and operations of other classes
        i. The analyst creates a *super- class* that contains basic attributes and operations that will be used in several *subclasses.*
        ii. Generalization is represented with the *a-kind-of* relationship, so that we say that an employee is a-kind-of person.
    - **Aggregation Relationships** Generally speaking, all aggregation relationships relate *parts* to *wholes* or *assemblies.* For our purposes, we use the *a-part-of* or *has-parts* semantic relationship to represent the aggregation abstraction
        i. Aggregation relationships are bidirectional. The flip side of aggregation is *decomposition.* The analyst can use decomposition to uncover parts of a class that should be modeled separately.
    - **Association Relationships** There are other types of relationships that do not fit neatly into a generalization (a-kind-of) or aggregation (a-part-of) framework.

5. When would you use each type of relationship?
    - Generalization = 'a-kind-of' relationship (secretary and employee)
    - Aggregation = 'a-part-of' relationship (door and car)
    - Association = when the other two don't apply.

6. How do you identify objects?
    - The four most common approaches are textual analysis, brainstorming, common object lists, and patterns.
    - The analyst performs *textual analysis* by reviewing the use-case diagrams and examining the text in the use-case descriptions to identify potential objects, attributes, operations, and relationships.
    - *Brainstorming* is a process that a set of individuals sitting around a table suggest potential classes that could be useful for the problem under consideration.
        i. This approach does not use the functional models developed earlier.
    - A *common object list* is simply a list of objects common to the business domain of the system.
        i. Several categories of objects have been found to help the analyst in creating the list, such as physical or tangible things, incidents, roles, and interactions.
    - From our perspective, a *pattern* is simply a useful group of collaborating classes that provide a solution to a commonly occurring problem. Because patterns provide a solution to commonly occurring problems, they are reusable.

7. What are patterns?
    - From our perspective, a *pattern* is simply a useful group of collaborating classes that provide a solution to a commonly occurring problem. Because patterns provide a solution to commonly occurring problems, they are reusable.

8. How can you use patterns in your projects?
    - Project: Inventory Management (that analyzes transactions).
    - It is possible to put together commonly found object-oriented patterns to form elegant object-oriented information systems. For example, many business transactions involve the same types of objects and interactions. Virtually all transactions would require a transaction class, a transaction line item class, an item class, a location class, and a participant class. By reusing these existing patterns of classes, we can more quickly and more completely define the system than if we start with a blank piece of paper.
    - Using patterns from different sources in enables the development team to leverage knowledge beyond that of the immediate team members and allows the team to develop more complete and robust models of the problem domain

9. What are CRC cards? (Appendix A)
    - *CRC (Class–Responsibility–Collaboration) cards* are used to document the responsibilities and collaborations of a class.
    - *Responsibilities* of a class can be broken into two separate types: knowing and doing.

        i. *Knowing responsibilities* are those things that an instance of a class must be capable of knowing. An instance of a class typically knows the values of its attributes and its relationships.
        ii. *Doing responsibilities* are those things that an instance of a class must be capable of doing.
    - Classes form *collaborations*. Collaborations allow the analyst to think in terms of clients, servers, and contracts.
        i. A *client* object is an instance of a class that sends a request to an instance of another class for an operation to be executed.
        ii. A *server* object is the instance that receives the request from the client object.
        iii. A *contract* formalizes the interactions between the client and server objects.

10. Explain how you would use a CRC card.
    - An analyst can use the idea of class responsibilities and client–server–contract collaborations to help identify the classes, along with the attributes, operations, and relationships, involved with a use case.
    - One of the easiest ways to use CRC cards in developing a structural model is through anthropomorphism—pretending that the classes have human characteristics. Members of the development team can either ask questions of themselves or be asked questions by other members of the team. Typically, the questions asked are of the form:
        i. Who or what are you? // What do you know? // What can you do?
    - The answers to the questions are then used to add detail to the evolving CRC cards

11. What are the elements of a CRC card? (Appendix A)
    - Each CRC card captures and describes the essential elements of a class.
        i. The front of the card contains the class's name, ID, type, description, associated use cases, responsibilities, and collaborators.
        ii. The back of a CRC card contains the attributes and relationships of the class

12. What Is a class diagram?
    - A *class diagram* is a *static model* that shows the classes and the relationships among classes that remain constant in the system over time. The class diagram depicts classes, which include both behaviors and states, with the relationships between the classes.

13. What is it used for?
    - To show the classes and the relationships among classes that remain constant in the system over time

14. What are the elements of class diagram? (Appendix B)
    - **Class** The main building block of a class diagram is the class, which stores and manages information in the system

i. Attributes are properties of the class about which we want to capture information
    1. *Derived attributes*, which are attributes that can be calculated or derived
    2. A *public* attribute (+) is one that is not hidden from any other object. As such, other objects can modify its value.
    3. A *protected* attribute (#) is one that is hidden from all other classes except its immediate subclasses.
    4. A *private* attribute (-) is one that is hidden from all other classes. The default visibility for an attribute is normally private.

ii. *Operations* are actions or functions that a class can perform. The functions that are available to all classes are not explicitly shown within the class rectangle. Instead, only operations unique to the class are included.
    1. A *constructor operation* creates a new instance of a class. Typically, we do not see constructor methods explicitly on the class diagram (cause they're common to all classes).
    2. A *query operation* makes information about the state of an object available to other objects, but it does not alter the object in any way. Usually not shown because we assume that all objects have operations that produce the values of their attributes.
    3. An *update operation* changes the value of some or all the object's attributes, which may result in a change in the object's state.
    4. A *destructor operation* simply deletes or removes the object from the system. Deleting an object is one of the basic functions and therefore would not be included on the class diagram.

- **Relationships** A primary purpose of a class diagram is to show the relationships, or associations, that classes have with one another. These are depicted on the diagram by drawing lines between classes
    i. When multiple classes share a relationship (or a class shares a relationship with itself), a line is drawn and labeled with either the name of the relationship or the roles that the classes play in the relationship.
    ii. Relationships also have *multiplicity*, which documents how an instance of an object can be associated with other instances (Appendix C).
    iii. There are times when a relationship itself has associated properties, especially when its classes share a many-to-many relationship. In these cases, a class called an *association class* is formed, which has its own attributes and operations.
- **Generalization and Aggregation Associations**
    i. A *generalization association* shows that one class (subclass) inherits from another class (superclass), meaning that the properties and operations of the superclass are also valid for objects of the subclass (**Appendix D**)
    ii. An *aggregation association* is used when classes actually comprise other classes. (**Appendix E**)
        1. Composition is used to portray a physical part of relationships and is shown by a black diamond. *Physical* implies that the part can be associated with only a single whole.

15. What's the difference between a class diagram, and an object type?
    - An *object diagram* is essentially an instantiation of all or part of a class diagram. *Instantiation* means to create an instance of the class with a set of appropriate attribute values.
    - Object diagrams are essentially much more thorough and specific than class diagrams and are used to uncover more details of a class.

16. What is an object diagram? (**Appendix F**)
    - An *object diagram* is essentially an instantiation of all or part of a class diagram. *Instantiation* means to create an instance of the class with a set of appropriate attribute values.

17. When do you use an object diagram?
    - When you want to uncover even more details of a class.

18. How do you verify and validate the structural model?
    - We combine walkthroughs with the power of role-playing as a way to more completely verify and validate the structural model that will underlie the business processes and functional models.
    - In fact, all of the object identification approaches described in this chapter can be viewed as a way to test the fidelity of the structural model.
    - Steps:
    - In this case, the verification and validation of the structural model are accomplished during a formal review meeting using a walkthrough approach in which an analyst presents the model to a team of developers and users. The analyst walks through the model, explaining each part of the model and all the reasoning behind the decision to include each of the classes in the structural model.
    - Rules:
    - First, every CRC card should be associated with a class on the class diagram, and vice versa
    - Second, the responsibilities listed on the front of the CRC card must be included as operations in a class on a class diagram, and vice versa.
    - Third, collaborators on the front of the CRC card imply some type of relationship on the back of the CRC card and some type of association that is connected to the associated class on the class diagram
    - Fourth, attributes listed on the back of the CRC card must be included as attributes in a class on a class diagram, and vice versa.
    - Fifth, the object type of the attributes listed on the back of the CRC card and with the attributes in the attribute list of the class on a class diagram implies an association from the class to the class of the object type.
    - Sixth, the relationships included on the back of the CRC card must be portrayed using the appropriate notation on the class diagram.
    - Seventh, an association class should be created only if there is indeed some unique characteristic about the intersection of the connecting classes.

Appendix A

Front:

Class Name: Old Patient	**ID:** 3	**Type:** Concrete, Domain
Description: An individual who needs to receive or has received medical attention		**Associated Use Cases:** 2

Responsibilities	**Collaborators**
Make appointment	Appointment
Calculate last visit	
Change status	
Provide medical history	Medical history

Back:

Attributes:

Amount (double)	
Insurance carrier (text)	

Relationships:

Generalization (a-kind-of): Person

Aggregation (has-parts): Medical History

Other Associations: Appointment

FIGURE 5-6
Sample CRC Card

Appendix B

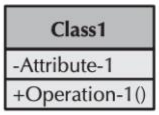
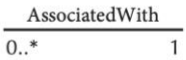
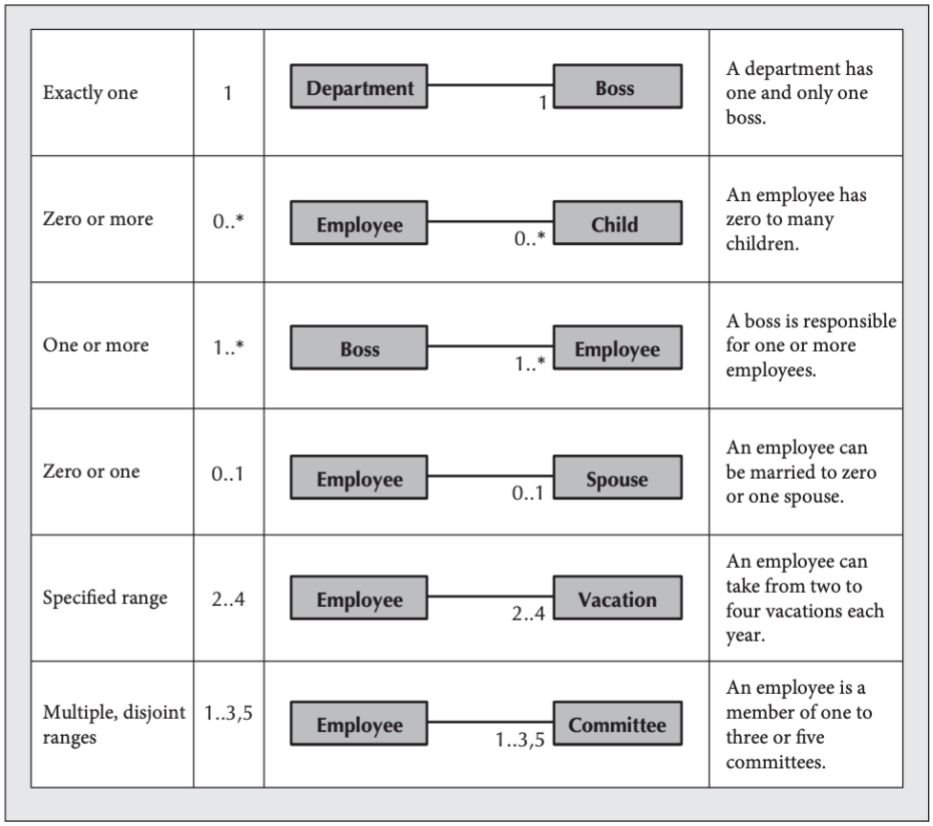
A class: • Represents a kind of person, place, or thing about which the system will need to capture and store information. • Has a name typed in bold and centered in its top compartment. • Has a list of attributes in its middle compartment. • Has a list of operations in its bottom compartment. • Does not explicitly show operations that are available to all classes.	
An attribute: • Represents properties that describe the state of an object. • Can be derived from other attributes, shown by placing a slash before the attribute's name.	attribute name /derived attribute name
An operation: • Represents the actions or functions that a class can perform. • Can be classified as a constructor, query, or update operation. • Includes parentheses that may contain parameters or information needed to perform the operation.	operation name ()
An association: • Represents a relationship between multiple classes or a class and itself. • Is labeled using a verb phrase or a role name, whichever better represents the relationship. • Can exist between one or more classes. • Contains multiplicity symbols, which represent the minimum and maximum times a class instance can be associated with the related class instance.	
A generalization: • Represents a-kind-of relationship between multiple classes.	
An aggregation: • Represents a logical a-part-of relationship between multiple classes or a class and itself. • Is a special form of an association.	
A composition: • Represents a physical a-part-of relationship between multiple classes or a class and itself • Is a special form of an association.	

FIGURE 5-8
Class Diagram Syntax

Appendix C



Exactly one	1	Department — 1 — Boss	A department has one and only one boss.
Zero or more	0..*	Employee — 0..* — Child	An employee has zero to many children.
One or more	1..*	Boss — 1..* — Employee	A boss is responsible for one or more employees.
Zero or one	0..1	Employee — 0..1 — Spouse	An employee can be married to zero or one spouse.
Specified range	2..4	Employee — 2..4 — Vacation	An employee can take from two to four vacations each year.
Multiple, disjoint ranges	1..3,5	Employee — 1..3,5 — Committee	An employee is a member of one to three or five committees.

FIGURE 5-10
Multiplicity

Appendix D

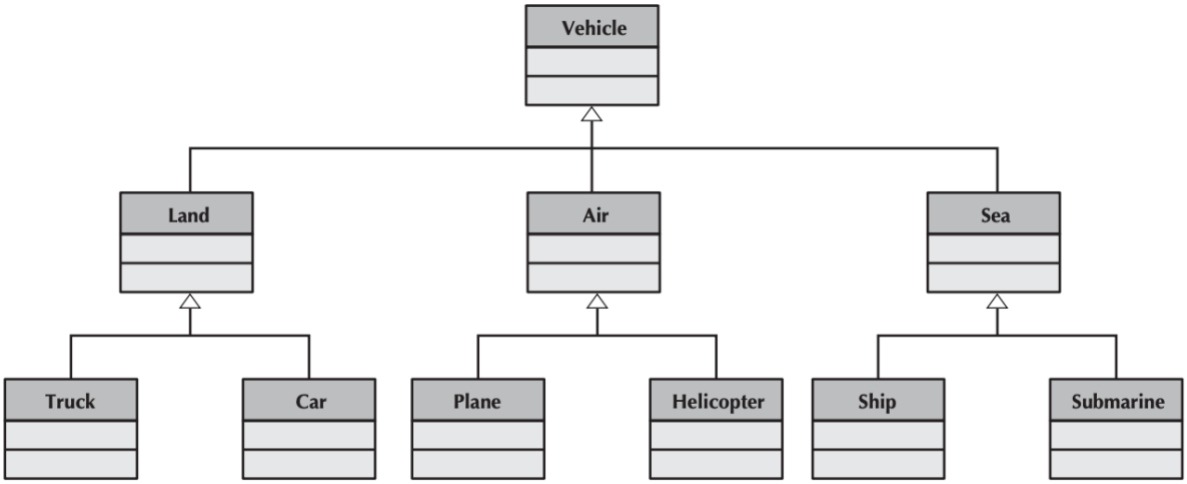


FIGURE 5-12 Sample Generalizations

Appendix E

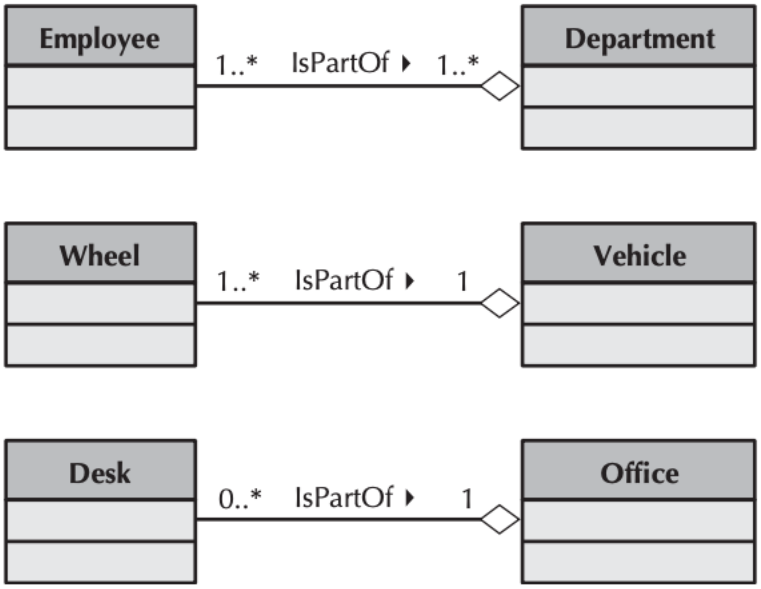


FIGURE 5-13
Sample Aggregation Associations

Appendix F

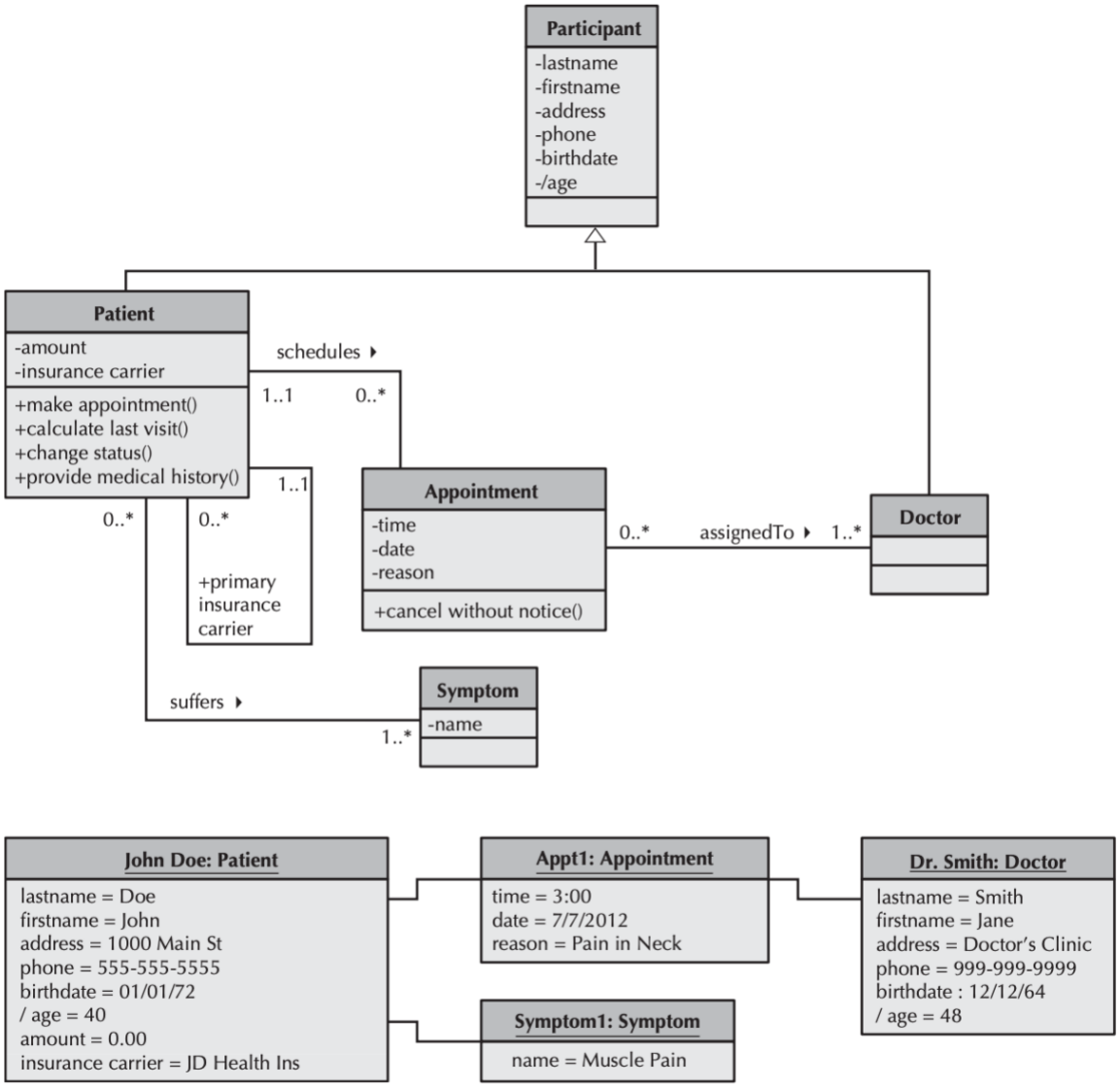


FIGURE 5-15 Sample Object Diagram

Chapter 6

This is a very important chapter

1. What are behavioral models?
    - Behavioral models describe the internal dynamic aspects of an information system that supports the business processes in an organization.
    - During analysis, behavioral models describe what the internal logic of the processes is without specifying how the processes are to be implemented. Later, in the design and implementation phases, the detailed design of the operations contained in the object is fully specified.

    - There are two types of behavioral models.
        - First, there are behavioral models used to represent the underlying details of a business process portrayed by a use-case model. (interaction diagram = sequence + communication diagram)
        - Second, a behavioral model is used to represent the changes that occur in the underlying data. (behavioral state machines)

2. What are the components of behavioral models?
    - 3 Unified Modeling Language (UML) diagrams that are used in behavioral modeling:
        - Sequence diagram // Communication diagram // Behavioral state machine
    - CRUDE (create, read, update, delete, execute) matrices?

3. What are interaction diagrams?
    - Interaction diagrams allow the analyst to model the distribution of the behavior of the system over the actors and objects in the system.
    - The process can be shown by the interaction that takes place between the objects that collaborate to support a use case through the use of interaction (sequence and communication) diagrams
    - They are made up of the sequence and communication diagrams.

4. How are objects, operations, and messages related to sequence diagram?
    - The process can be shown by the interaction that takes place between the objects that collaborate to support a use case through the use of interaction (sequence and communication) diagrams.
    - Sequence diagram is a *dynamic model* that shows the explicit sequence of messages that are passed between objects in a defined interaction
        - *Generic sequence diagram* that shows all possible scenarios for a use case
        - *Instance sequence diagrams*, each of which depicts a single *scenario* within the use case.

5. What are the elements of a sequence diagram? **(Appendix A)**
    - *Actors* and *objects* that participate in the sequence are placed across the top of the dia- gram using actor symbols from the use-case diagram and object symbols from the object diagram

- A dotted line runs vertically below each actor and object to denote the *lifeline* of the actors and objects over time.
- Sometimes an object creates a *temporary object;* in this case, an X is placed at the end of the lifeline at the point where the object is destroyed.
- A thin rectangular box, called the *execution occurrence,* is overlaid onto the lifeline to show when the classes are sending and receiving message.
    - However, in the case of using sequence diagrams to model use cases, two types of messages are typically used: operation call and return:
    - *Operation call messages* passed between objects are shown using solid lines connecting two objects with an arrow on the line showing which way the message is being passed.
    - A *return message* is depicted as a dashed line with an arrow on the end of the line portraying the direction of the return.
- At times a message is sent only if a *condition* is met. In those cases, the condition is placed between a set of brackets. The condition is placed in front of the message name
- An object can send a message to itself. This is known as *self-delegation.*
- The *frame*.

6. **How are sequence diagrams related to use case diagrams?**
    - Sequence Diagrams usually depict one use case inside of the use case diagrams (step 1)
    - Sequence Diagrams uses Use Case diagrams to identify actors and objects (step 2).
    - The second step is to identify the actors and objects that participate in the sequence being modeled—i.e., the actors and objects that interact with each other during the use-case scenario.

7. **How are sequence diagrams related to use case description?**
    - The Flow of Events are used to identify the number of instance-specific sequence diagram needed (step 1) and to find actors and objects (step 2).
    - Also, to know the actual messages sent between objects (step 4)
    - "Given that there are no Alternate/Exceptional Flows or any decisions being made in the Normal Flow of Events, nor are there any decisions in the activity diagram associated with the Add Apartment use case, there is only one scenario to be portrayed. Consequently, there is only one instance-specific sequence diagram to be created." → Used to find different use case scenarios?

8. **How are use case diagrams related to activity diagrams.**
    - Note: page, 214, and page 215. Indicate figure six seven where the activity diagram, as shown, corresponding to figure, six nine where the sequence diagram is shown and figure, six, eight, is the corresponding use case diagram.
    - "By looking through the Normal Flow of Events and the activity diagram, we see that the only object that seems to be relevant is the instance of the Apartment class that is being added." → Used to find relevant objects?

- Activity diagrams are used to know how many instance-specific sequence diagrams are needed (step 1).
- Also, to know the actual messages sent between objects (step 4).

9. What is the communication diagram?
    - Communication diagrams, like sequence diagrams, essentially provide a view of the dynamic aspects of an object-oriented system. They can show how the members of a set of objects collaborate to implement a use case or a use-case scenario. They can also be used to model all the interactions among a set of collaborating objects, in other words, a collaboration.
    - A communication diagram is essentially an object diagram that shows message-passing relationships instead of aggregation or generalization associations.
        - Very useful for process patterns.

10. What is it used for?
    - They can show how the members of a set of objects collaborate to implement a use case or a use-case scenario. They can also be used to model all the interactions among a set of collaborating objects, in other words, a collaboration.
    - To understand the flow of control over a set of collaborating objects or to understand which objects collaborate to support business processes, a communication diagram can be used.
    - For time ordering of the messages, a sequence diagram should be used. In some cases, both can be used to more fully understand the dynamic activity of the system.

11. What are the elements of the communication diagram? (**Appendix C**)
    - Actors and objects that collaborate to execute the use case are placed on the communication diagram in a manner to emphasize the message passing that takes place between them.
    - An *object* participates in a collaboration by sending and/or receiving messages.
    - An *association* is shown between actors and objects with an undirected line.
    - The *message* is shown as the label of the association.
    - A *condition*.
    - A *frame*.

12. How is it related to a sequence diagram?
    - Unlike the sequence diagram, the communication diagram does not have a means to explicitly show an object being deleted or created.
    - Another difference between the two interaction diagrams is that the communication diagram never shows returns from message sends, whereas the sequence diagram can optionally show them.
    - Like the sequence diagram, the communication diagram can represent conditional messages.
    - When comparing the communication diagrams to the sequence diagrams in these figures, you see that quite a bit of information is lost

13. How is it related to a use case?

- Use cases are used in order to set the context (step 1) for the creation of the communication diagram.
- Like a sequence diagram, the context of the diagram can be a system, a use case, or a scenario of a use case. The context of the diagram is depicted as a labeled frame around the diagram.
- Use cases (and their descriptions) are used in Step 4 to place the relevant association between actors and objects of the scenario, and to see their directionality and content (i.e., to write the messages).

14. What are behavioral state machines.
    - A behavioral state machine is a dynamic model that shows the different states through which a single object passes during its life in response to events, along with its responses and actions.
    - Typically, behavioral state machines are not used for all objects; rather, behavioral state machines are used with complex objects to further define them and to help simplify the design of algorithms for their methods.
    - Behavioral state machines should be used to help understand the dynamic aspects of a single class and how its instances evolve over time.

15. How do they relate to classes and objects?
    - Typically, behavioral state machines are not used for all objects; rather, behavioral state machines are used with complex objects to further define them and to help simplify the design of algorithms for their methods.
    - Behavioral state machines should be used to help understand the dynamic aspects of a single class and how its instances evolve over time.

16. How do they relate to actions and activities?
    - An object typically moves from one state to another based on the outcome of an action triggered by an event.
    - An *action* is an atomic, non-decomposable process that cannot be interrupted. From a practical perspective, actions take zero time, and they are associated with a transition.
        - A *transition* is a relationship that represents the movement of an object from one state to another state. Some transitions have a guard condition.
        - A *guard condition* is a Boolean expression that includes attribute values, which allows a transition to occur only if the condition is true.
    - In contrast, an *activity* is a non-atomic, decomposable process that can be interrupted. Activities take a long period of time to complete, and they can be started and stopped by an action.

    - Basically, behavioral state machines show the effects of an event's actions (and activities) on the state of an object.
        - The *state* of an object is defined by the value of its attributes and its relationships with other objects at a particular point in time
        - An *event* is something that takes place at a certain point in time and changes a value or values that describe an object, which, in turn, changes the object's state.

17. What are the elements of a behavioral state machine? (**Appendix D**)
    - A *state* is a set of values that describes an object at a specific point in time and represents a point in an object's life in which it satisfies some condition, performs some action, or waits for something to happen.
    - An *initial state* is shown using a small, filled-in circle (when object is created).
    - Object's *final state* is shown as a circle surrounding a small, filled-in circle (when object ceases to exist).
    - *Arrows* are used to connect the state symbols, representing the transitions between states. Each arrow is labeled with the appropriate event name and any parameters or conditions that may apply.

18. How do you know if you need to create a behavioral state machine diagram?
    - If you have an object that is being transformed, you probably want to make a behavioral state machine diagram.
    - Create a behavioral state machine for objects whose behavior changes based on the state of the object
    - You should examine your class diagram to identify which classes undergo a complex series of state changes and draw a behavioral state diagram for each of them.

19. Explain what CRUDE analysis is.
    - CRUDE analysis uses a *CRUDE matrix*, in which each interaction among objects is labeled with a letter for the type of interaction: C for create, R for read or reference, U for update, D for delete, and E for execute.
    - In an object-oriented approach, a class/actor-by-class/actor matrix is used. Each cell in the matrix represents the interaction between instances of the classes.

20. What is it used for?
    - To identify how the underlying objects in the problem domain work together to collaborate in support of the use cases.
    - CRUDE analysis can be used as a way to partially validate the interactions among the objects in an object-oriented system.
    - CRUDE analysis also can be used to identify complex objects (for behavioral state machines). The more (C)reate, (U)pdate, or (D)elete entries in the column associated with a class, the more likely the instances of the class have a complex life cycle.

21. How do you conduct a CRUDE analysis?
    - First, a class/actor-by-class/actor matrix is made. Each cell in the matrix represents the interaction between instances of the classes.
    - Then, you label each interaction with the letter for the type of interaction C for create, R for read or reference, U for update, D for delete, and E for execute.
    - OR

- The best way to create a CRUDE matrix is to conceptually merge the sequence and communication diagrams that model all of the scenarios of all of the use cases in a system.
    - The easiest way to accomplish this is simply to create an empty class/actor- by-class/actor matrix.
- Once this matrix has been laid out, role-playing the scenarios will show which actors and classes interact with each other.
- OR
- The first thing we need to do is to identify all of the actors and the classes that are involved in the campus housing service example
- Then label each one.

22. How does the behavioral state machine relate to the CRUDE matrix?
    - CRUDE analysis also can be used to identify complex objects (for behavioral state machines). The more (C)reate, (U)pdate, or (D)elete entries in the column associated with a class, the more likely the instances of the class have a complex life cycle.

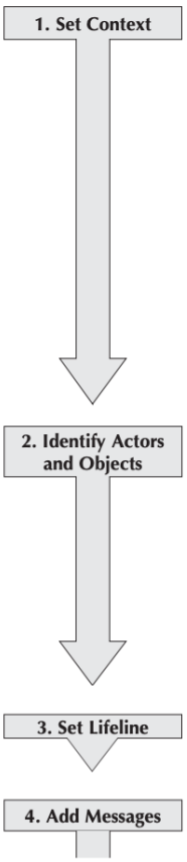
23. How do you verify and validate the behavioral model?
    - We combine walkthroughs with CRUDE matrices to more completely verify and validate the behavioral model
    - Rules:
    - First, every actor and object included on a sequence diagram must be included as an actor and an object on a communication diagram, and vice versa
    - Second, if there is a message on the sequence diagram, there must be an association on the communications diagram, and vice versa
    - Third, every message that is included on a sequence diagram must appear as a message on an association in the corresponding communication diagram, and vice versa.
    - Fourth, if a guard condition appears on a message in the sequence diagram, there must be an equivalent guard condition on the corresponding communication diagram, and vice versa.
    - Fifth, the sequence number included as part of a message label in a communications diagram implies the sequential order in which the message will be sent
    - Sixth, all transitions contained in a behavior state machine must be associated with a message being sent on a sequence and communication diagram, and it must be classified as a (C)reate, (U)pdate, or (D)elete message in a CRUDE matrix.
    - Seventh, all entries in a CRUDE matrix imply a message being sent from an actor or object to another actor or object.
    -

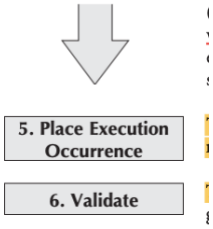
Appendix A

Term and Definition	Symbol
An actor: ■ Is a person or system that derives benefit from and is external to the system. ■ Participates in a sequence by sending and/or receiving messages. ■ Is placed across the top of the diagram. ■ Is depicted either as a stick figure (default) or, if a nonhuman actor is involved, as a rectangle with <<actor>> in it (alternative).	 anActor <<actor>> anActor
An object: ■ Participates in a sequence by sending and/or receiving messages. ■ Is placed across the top of the diagram.	anObject : aClass
A lifeline: ■ Denotes the life of an object during a sequence. ■ Contains an X at the point at which the class no longer interacts.	
An execution occurrence: ■ Is a long narrow rectangle placed atop a lifeline. ■ Denotes when an object is sending or receiving messages.	
A message: ■ Conveys information from one object to another one. ■ A operation call is labeled with the message being sent and a solid arrow, whereas a return is labeled with the value being returned and shown as a dashed arrow.	aMessage() ReturnValue
A guard condition: ■ Represents a test that must be met for the message to be sent.	[aGuardCondition]:aMessage()
For object destruction: ■ An X is placed at the end of an object's lifeline to show that it is going out of existence.	X
A frame: ■ Indicates the context of the sequence diagram.	Context

FIGURE 6-2 Sequence Diagram Syntax

Appendix B


1. Set Context
2. Identify Actors and Objects
3. Set Lifeline
4. Add Messages


5. Place Execution Occurrence
6. Validate

Appendix C

Term and Definition	Symbol
An actor: ■ Is a person or system that derives benefit from and is external to the system. ■ Participates in a collaboration by sending and/or receiving messages. ■ Is depicted either as a stick figure (default) or, if a nonhuman actor is involved, as a rectangle with <<actor>> in it (alternative).	 anActor <<actor>> anActor
An object: ■ Participates in a collaboration by sending and/or receiving messages.	anObject : aClass
An association: ■ Shows an association between actors and/or objects. ■ Is used to send messages.	
A message: ■ Conveys information from one object to another one. ■ Has direction shown using an arrowhead. ■ Has sequence shown by a sequence number.	SeqNumber: aMessage →
A guard condition: ■ Represents a test that must be met for the message to be sent.	SeqNumber: [aGuardCondition]: aMessage →
A frame: ■ Indicates the context of the communication diagram.	Context

FIGURE 6-11 Communication Diagram Syntax

Appendix D

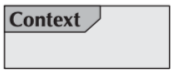
Term and Definition	Symbol
A state: ■ Is shown as a rectangle with rounded corners. ■ Has a name that represents the state of an object.	
An initial state: ■ Is shown as a small, filled-in circle. ■ Represents the point at which an object begins to exist.	
A final state: ■ Is shown as a circle surrounding a small, filled-in circle (bull's-eye). ■ Represents the completion of activity.	
An event: ■ Is a noteworthy occurrence that triggers a change in state. ■ Can be a designated condition becoming true, the receipt of an explicit signal from one object to another, or the passage of a designated period of time. ■ Is used to label a transition.	anEvent
A transition: ■ Indicates that an object in the first state will enter the second state. ■ Is triggered by the occurrence of the event labeling the transition. ■ Is shown as a solid arrow from one state to another, labeled by the event name.	
A frame: ■ Indicates the context of the behavioral state machine.	

FIGURE 6-17 Behavioral State Machine Diagram Syntax

Appendix E

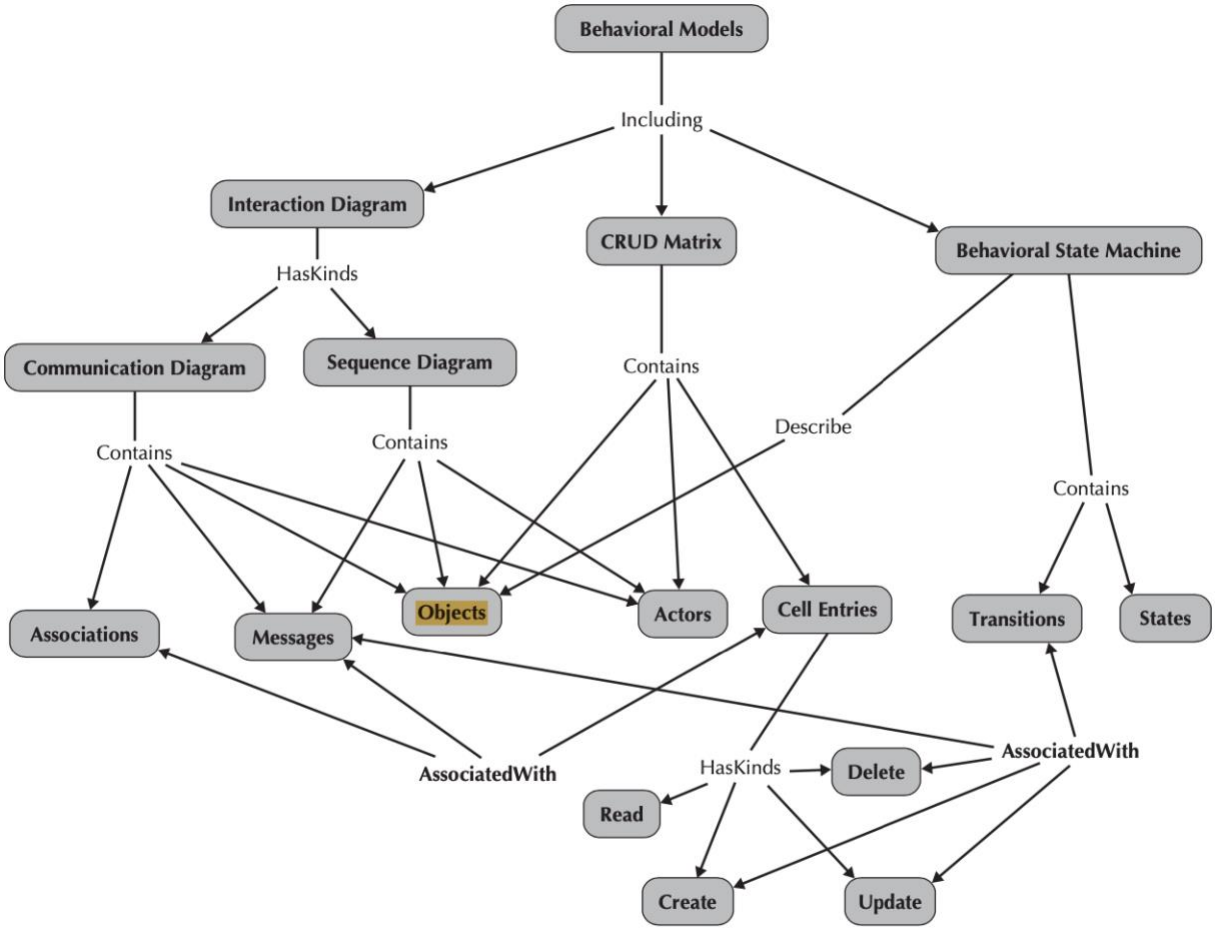


FIGURE 6-27 Interrelationships among Behavioral Models

Chapter 7

This is a very important chapter

Chapter Seven excludes the Design Strategies section (covered in BTM 481). Pages, 268 to 275 are excluded.

24. What are the analysis models?
    - The functional models:
        - Use Case diagrams
        - Activity diagrams
        - Use-Case descriptions
    - The structural models:
        - CRC cards
        - Class diagrams
        - Object diagrams
    - The behavioral models:
        - Sequence diagrams
        - Communication diagrams
        - Behavioral state machines
        - CRUDE matrices

25. How do you **verify and validate** the analysis model? (**Appendix A**)
    - This includes testing the fidelity of each model.
    - It also involves testing the fidelity between the models.
    - Basically, you have to balance all the models together.

    - **What does this mean to balance functional and structural model?**
    - Balancing the *Functional*/Structural models: (**Appendix B**)
    - To balance the functional and structural models, we must ensure that the two sets of models are consistent with each other. That is, the *activity diagrams, use-case descriptions, and use-case diagrams* must agree with the CRC cards and class diagrams that represent the evolving model of the problem domain.
    - First, every class on a class diagram and every CRC card must be associated with at least one use-case, and vice versa.
    - Second, every activity or action contained in an activity diagram and every event contained in a use-case description should be related to one or more responsibilities on a CRC card and one or more operations in a class on a class diagram and vice versa.
    - Third, every object node on an activity diagram must be associated with an instance of a class on a class diagram and a CRC card or an attribute contained in a class and on a CRC card.
    - Fourth, every attribute and association/aggregation relationships contained on a CRC card (and connected to a class on a class diagram) should be related to the subject or object of an event in a use-case description.

    - **What does it mean to balance functional and behavioral models?**
    - Balancing *Functional*/**Behavioral** models: (**Appendix C**)

- In this case, *the activity diagrams, use-case descriptions, and use-case diagrams* must agree with the **sequence diagrams, communication diagrams, behavioral state machines, and CRUDE matrix**.
- First, the sequence and communication diagrams must be associated with a use case on the use-case diagram and a use-case description
- Second, actors on sequence diagrams, communication diagrams, and/or CRUDE matrices must be associated with actors on the use-case diagram or referenced in the use- case description, and vice versa
- Third, messages on sequence and communication diagrams, transitions on behavioral state machines, and entries in a CRUDE matrix must be related to activities and actions on an activity diagram and events listed in a use-case description, and vice versa
- Fourth, all complex objects represented by an object node in an activity diagram must have a behavioral state machine that represents the object's lifecycle, and vice versa.

- **What does it mean to balance structural and behavioral models?**
- Balancing Structural/**Behavioral** models: (**Appendix D**)
- We connect CRC cards and class diagrams to **sequence and communication diagrams and the CRUDE matrix**.
- First, objects that appear in a CRUDE matrix must be associated with classes that are represented by CRC cards and appear on the class diagram, and vice versa
- Second, because behavioral state machines represent the life cycle of complex objects, they must be associated with instances (objects) of classes on a class diagram and with a CRC card that represents the class of the instance
- Third, communication and sequence diagrams contain objects that must be an instantiation of a class that is represented by a CRC card and is located on a class diagram.
- Fourth, messages contained on the sequence and communication diagrams, transitions on behavioral state machines, and cell entries on a CRUDE matrix must be associated with responsibilities and associations on CRC cards and operations in classes and associations connected to the classes on class diagrams.
- Fifth, the states in a behavioral state machine must be associated with different values of an attribute or set of attributes that describe an object.

2. How do you transition from analysis to design?
   - First, before we evolve our analysis representations into design representations, we need to verify and validate the current set of analysis models to ensure that they faithfully represent the problem domain under consideration.

   - From an object-oriented perspective, system design models simply refine the system analysis models by adding system environment (or solution domain) details to them and refining the problem domain information already contained in the analysis models.
   - When evolving the analysis model into the design model, you should first carefully review the use cases and the current set of classes (from a design lens).
     - We make modifications to the problem domain models that will enhance the efficiency and effectiveness of the evolving system.
   - We also use factoring // partitions // collaborations.

3. **Why is balancing important?**
    - Balancing is important because it allows us to make sure that all the analysis models are good before going into the design models, and to ensure consistency.
    - It's part of the iterative nature of object-oriented development.

4. What does **factoring** mean? Give an example.
    - *Factoring* is the process of separating out a *module* into a stand-alone module. The new module can be a new *class* or a new *method.*
    - For example, when reviewing a set of classes, it may be discovered that they have a similar set of attributes and methods. Thus, it might make sense to factor out the similarities into a separate class.
        - if the Employee class had not been identified, we could possibly identify it at this stage by factoring out the similar methods and attributes from the Nurse, Receptionist, and Doctor classes.
    - *Abstraction* and *refinement* are two processes closely related to factoring.
        - *Abstraction* deals with the creation of a higher-level idea from a set of ideas. In some cases, the abstraction process identifies *abstract classes,* whereas in other situations, it identifies additional *concrete classes.*
        - The *refinement* process is the opposite of the abstraction process (adding sub-classes to a high-level class).

5. What does **partitioning** mean? Give an example.
    - *Partioning* is to split the representation into a set of *partitions* (done because the system can appear huge after factoring).
    - A partition is the object-oriented equivalent of a subsystem, where a sub- system is a decomposition of a larger system into its component systems
        - Example: an accounting information system could be functionally decomposed into an accounts-payable system, an accounts-receivable system, a payroll system, etc.
    - From an object-oriented perspective, partitions are based on the pattern of activity (messages sent) among the objects in an object- oriented system.

6. Explain partition and collaboration.
    - A partition is the object-oriented equivalent of a subsystem, where a sub-system is a decomposition of a larger system into its component systems (e.g., an accounting information system could be functionally decomposed into an accounts-payable system).
    - From an object-oriented perspective, partitions are based on the pattern of activity (messages sent) among the objects in an object- oriented system.
    - A good place to look for potential partitions is the *collaborations* modeled in UML's communication diagrams.
        - In cases where classes are supporting multiple use cases, the collaborations should be merged.
        - Creating a diagram that combines the class diagram with the communication diagrams can be very useful to show to what degree the classes are coupled, or cluster analysis, or multiple dimensional scaling.
        - If collaborations are complex, they can be divided into partitions.

- The primary purpose of identifying collaborations and partitions is to determine which classes should be grouped together in design.

7. Explain abstraction and refinements and give an example.
    - *Abstraction* deals with the creation of a higher-level idea from a set of ideas. In some cases, the abstraction process identifies *abstract classes,* whereas in other situations, it identifies additional *concrete classes.*
        - Example: if the Employee class had not been identified, we could possibly identify it at this stage by factoring out the similar methods and attributes from the Nurse, Receptionist, and Doctor classes.
    - The *refinement* process is the opposite of the abstraction process (adding sub-classes to a high-level class).
        - Example: we could identify additional subclasses of the Employee class, such as Secretary and Bookkeeper.

8. **What do layers refer to?**
    - A *layer* represents an element of the software architecture of the evolving system.

9. **What kinds of layers do you have in your project?**
    - There should be a layer for each of the different elements of the system environment (e.g., data management, user interface, physical architecture).
    - We suggest the following layers on which to base software architecture: foundation, problem domain, data management, human–computer interaction, and physical architecture.
    - In our project, we have:
        - Foundation
        - Promblem Domain
        - Data Management
        - HCI
        - Physical Architecture

10. **Which is the most important layer that is most likely to be very different between one system, and another?**
    - The problem-domain layer.

11. What is a problem domain?
    - The problem domain is the problem that is being addressed by the team.

12. What classes do you have in your problem domain layer of your project.
    - Damn.

13. What are package diagrams used for? (example: **Appendix F**)
    - A *package* is a general construct that can be applied to any of the elements in UML models. A package serves the same purpose as a folder on your computer
    - *Package diagram*: a diagram composed only of packages. A package diagram is effectively a class diagram that only shows packages.
        - Partitions (i.e., clusters of classes) are modelled as packages.

14. What are the elements of a package diagram? (**Appendix E**)

- **A package:** A logical grouping of UML elements. It is used to simplify UML diagrams by grouping related elements into a single higher-level element.
- **A dependency relationship:** Represents a dependency between packages: If a package is changed, the dependent package also could have to be modified. It has an arrow drawn from the dependent package toward the package on which it is dependent.

Appendix A

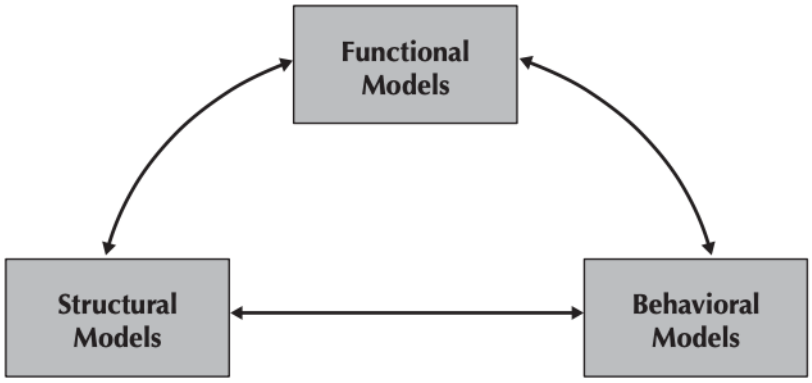


FIGURE 7-1
Object-Oriented Analysis Models

Appendix B

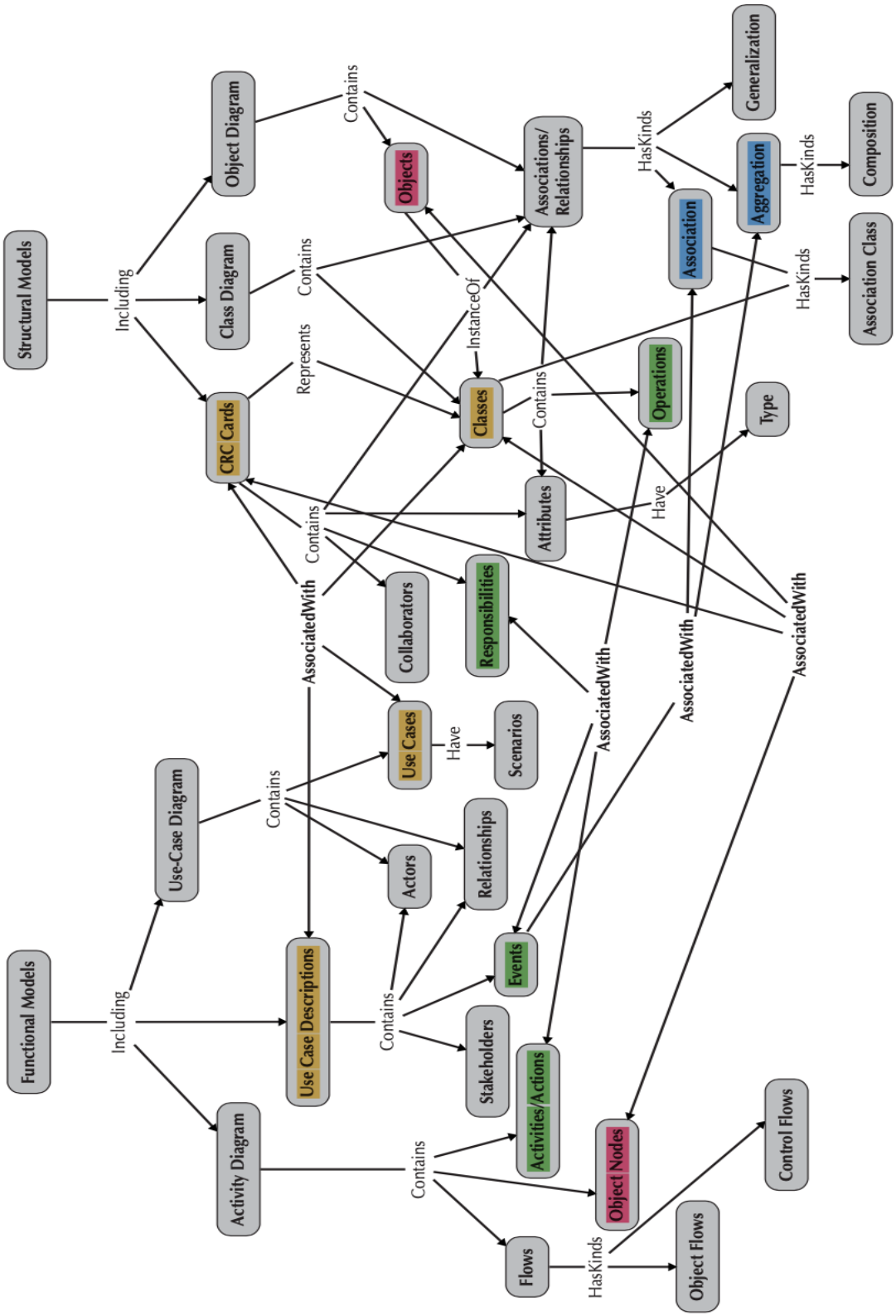


FIGURE 7-2 Relationships among Functional and Structural Models

Appendix C

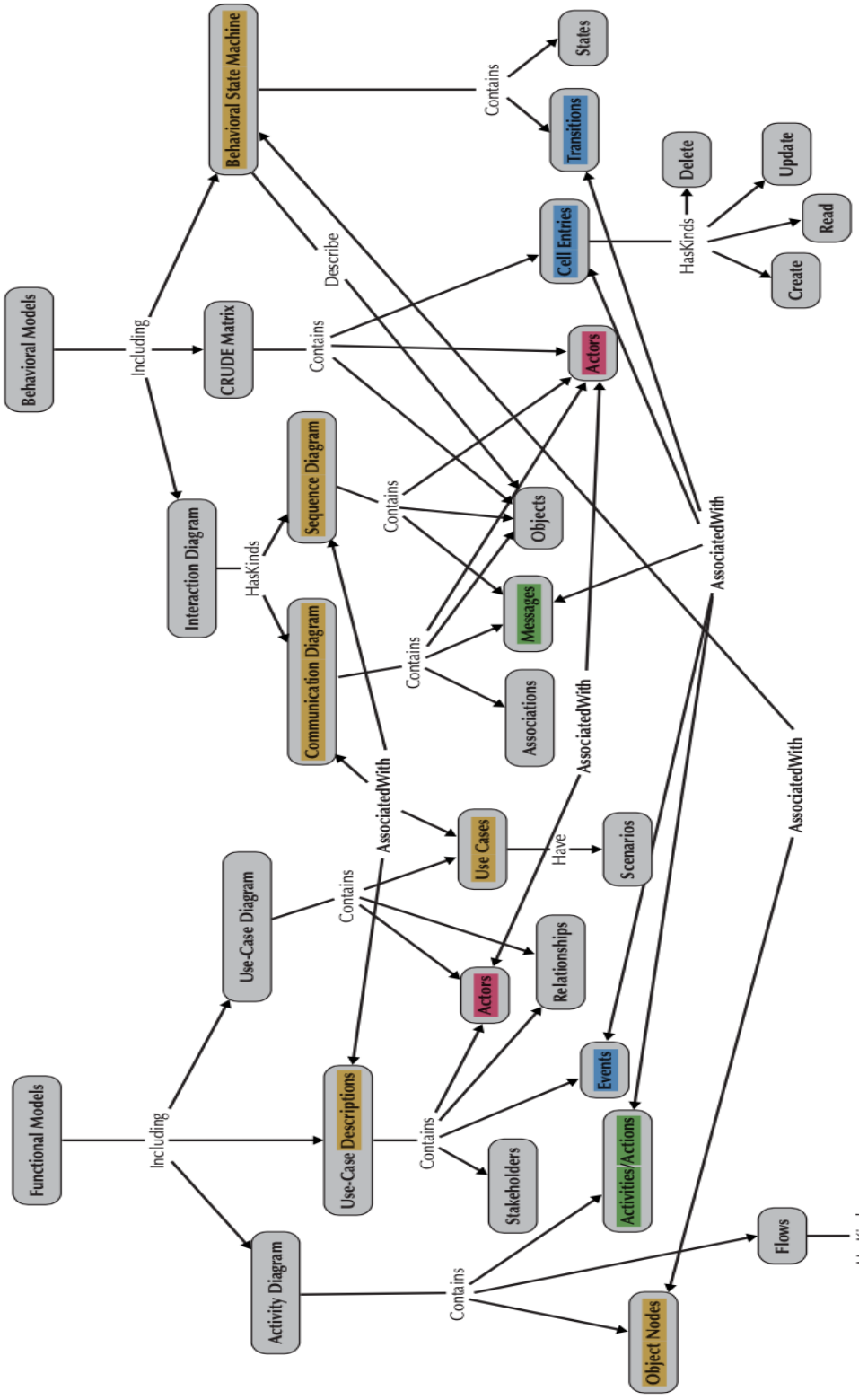


FIGURE 7-8 Relationships between Functional and Behavioral Models

Appendix D

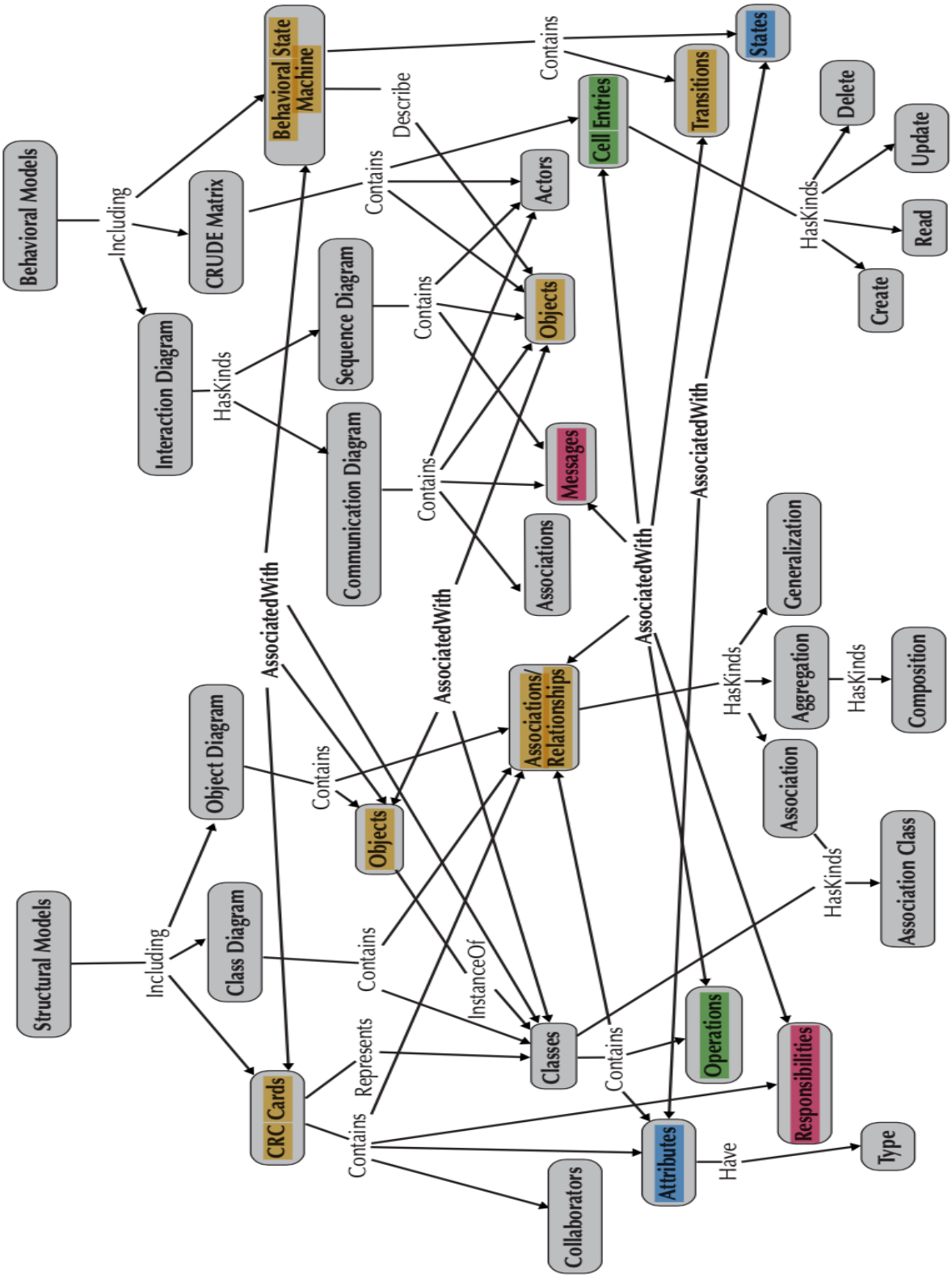


FIGURE 7-13 Relationships between Structural and Behavioral Models

Appendix E

A package: ■ Is a logical grouping of UML elements. ■ Is used to simplify UML diagrams by grouping related elements into a single higher-level element.	
A dependency relationship: ■ Represents a dependency between packages: If a package is changed, the dependent package also could have to be modified. ■ Has an arrow drawn from the dependent package toward the package on which it is dependent.	

FIGURE 7-18 Syntax for Package Diagram

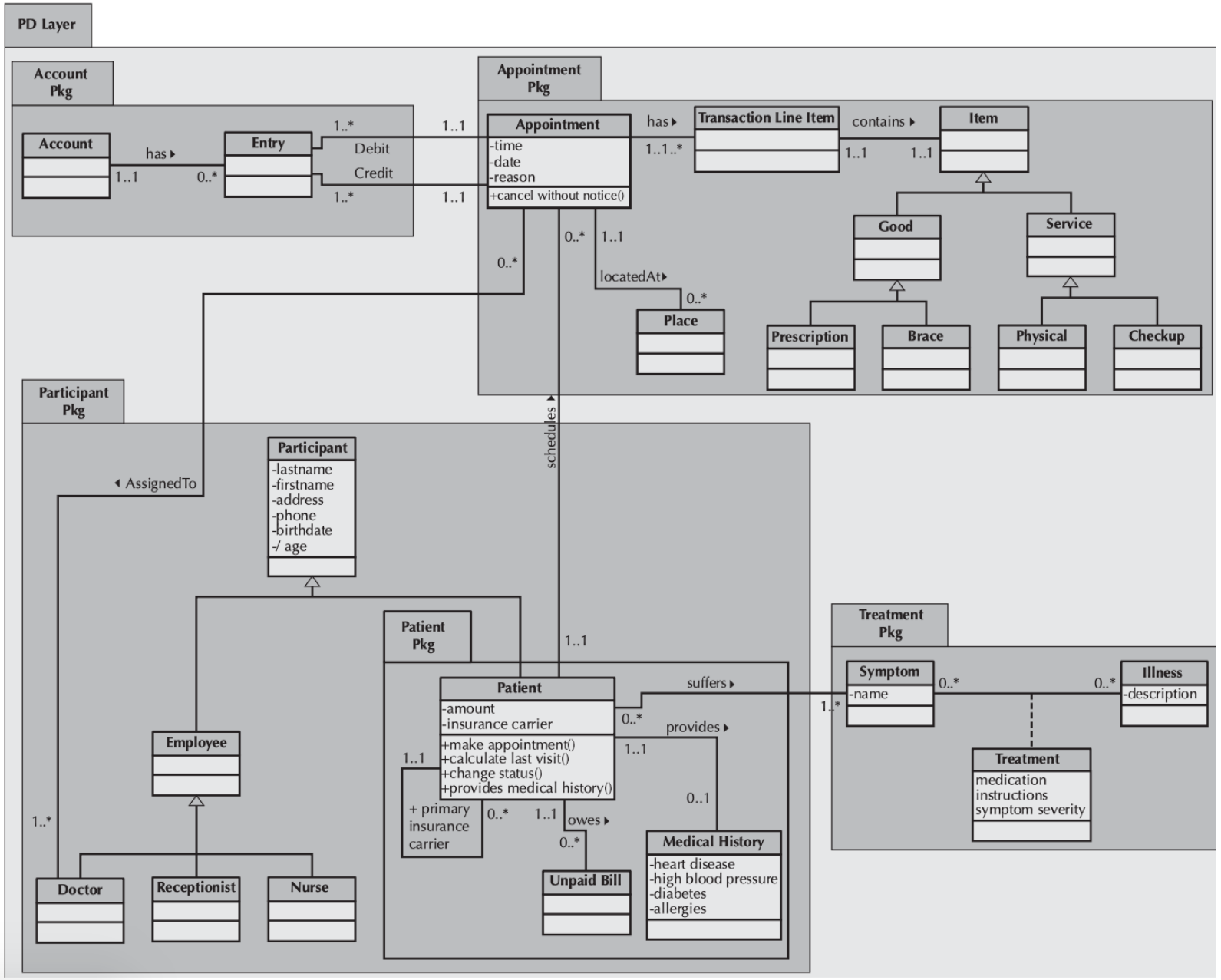


FIGURE 7.21 Package Diagram of the PD Layer of the Appointment Problem

Chapter 8

This is a very important chapter

19. What are the levels of **abstraction** in object-oriented system? (**Appendix A**)
    - These levels include variable, method, class/object, package, library, and/or application/ system levels.
    - The changes that take place at one level can affect other levels (e.g., changes to a class can affect the package level, which can affect both the system level and the library level, which in turn can cause changes back down at the class level). Finally, changes can occur at different levels at the same time.

20. What is the meaning of abstraction?

21. What are the basic characteristics of object orientation?
    - Object orientation has **classes, objects, methods, and messages.**
    - In object orientation, we **encapsulate processes into an object** and sometimes **hide information.** We separate methods from objects.
    - Objects can take many forms (**polymorphism**) depending on the information received.
    - **and Dynamic Binding**
    - **Inheritance**

22. Explain **classes, objects, methods and messages**.
    - The basic building block of the system is the *object*. Objects are *instances* of classes.
        i. Each object has *attributes* that describe data about the object. Objects have *state*, which is defined by the value of its attributes and its relationships with other objects at a particular point in time.
    - *Classes* are templates that we use to define both the data and processes that each object contains.
    - And each object has *methods*, which specify what processes the object can perform. From our perspective, methods are used to implement the *operations* that specified the *behavior* of the objects.
    - To get an object to perform a method (e.g., to delete itself), a *message* is sent to the object. A message is essentially a function or procedure call from one object to another object.

23. Explain **encapsulation** and **information hiding**.
    - *Encapsulation* is the mechanism that combines the processes and data into a single object.
    - *Information hiding* suggests that only the information required to use an object be available outside the object; that is, information hiding is related to the *visibility* of the methods and attribute.
    - The fact that we can use an object by sending a message that calls methods is the key to reusability because it shields the internal workings of the object from

changes in the outside system, and it keeps the system from being affected when changes are made to an object.

24. Explain **polymorphism** and dynamic binding.
    - *Polymorphism* means having the ability to take several forms. By supporting polymorphism, object-oriented systems can send the same message to a set of objects, which can be interpreted differently by different classes of objects.
    - *Dynamic binding* refers to the ability of object-oriented systems to defer the data typing of objects to run time.
    - The specific level of support for polymorphism and dynamic binding is language specific. Most object-oriented programming languages support dynamic binding of methods, and some support dynamic binding of attributes.

25. Give an example of polymorphism misuse.
    - Polymorphism can be a double-edged sword. Through the use of dynamic binding, there is no way to know before run time which specific object will be asked to execute its method. In effect, there is a decision made by the system that is not coded anywhere.
    - Because all these decisions are made at run time, it is possible to send a message to an object that it does not understand (i.e., the object does not have a corresponding method). This can cause a run-time error that, if the system is not programmed to handle it correctly, can cause the system to abort.
    - E.g., the 'Employee' object is made to execute 'giveCustomerInformation()' method.
    - OR
    - Two inherited attributes (or methods) of a SubClass have the same name but different semantics. (Basically, two superclasses have the same method name, but they do different things)

26. Explain **inheritance**.
    - *Inheritance* allows developers to define classes incrementally by reusing classes defined previously as the basis for new classes.
    - A superclass contains the data and methods needed by the subclasses and thus these classes inherit the properties of the superclass (above them).
        i. *Single inheritance* allows a subclass to have only a single parent class.
        ii. With *multiple inheritance*, a subclass may inherit from more than one superclass.

27. Give an example of inheritance misuse.
    -

28. Give an example of inheritance correct use.

29. Explain **inheritance conflict**.
    - Inheritance conflict is when a subclass inherits an attribute/method of a super-class that conflicts with one of its current attributes/methods.

- This may lead to an error at run-time, or the program executing in a way that was not intended, as well as giving information to an object in a way that is misrepresentative.

30. How are inheritance and polymorphism related?
    - When considering the interaction of inheritance with polymorphism and dynamic binding, object-oriented systems provide the developer with a very powerful, but dangerous, set of tools.
    - Depending on the object-oriented programming language used, this interaction can allow the same object to be associated with different classes at different times.

31. What are the **design criteria** you have to evaluate and follow?
    - A good design is one that balances trade-offs to minimize the total cost of the system over its entire lifetime. Criteria are:
        i. Coupling
        ii. Cohesion
        iii. Connascence

32. What does **coupling** refer to?
    - *Coupling* refers to how interdependent or interrelated the modules (classes, objects, and methods) are in a system. Basically, if modules are very interdependent (if they are coupled), then a small change is going to a big butterfly effect.
    - The higher the interdependency, the more likely changes in part of a design can cause changes to be required in other parts of the design. Two types:
        i. *Interaction coupling* deals with the coupling among methods and objects through message passing. Should be minimized. The one possible exception is that non–problem-domain classes must be coupled to their corresponding problem-domain classes. (**Appendix A**)
        ii. *Inheritance coupling*, as its name implies, deals with how tightly coupled the classes are in an inheritance hierarchy. Most authors tend to say simply that this type of coupling is desirable (but not too high).

33. Explain the **Law of Demeter**. Give an example of the Law of Demeter.
    - *Law of Demeter* is a guideline to minimize interaction coupling. Essentially, the law minimizes the number of objects that can receive messages from a given object.
    - The law states that an object should send messages only to one of the following:
        i. Itself. (For example, Object1 can send Message1 to itself. In other words, a method associated with Object1 can use other methods associated with Object1.13)
        ii. An object that is contained in an attribute of the object or one of its superclasses
        iii. An object that is passed as a parameter to the method
        iv. An object that is created by the method

v. An object that is stored in a global variable

34. Explain **cohesion**.
    - *Cohesion* refers to how single-minded a module (class, object, or method) is within a system. A class or object should represent only one thing, and a method should solve only a single task. Three types: method, class and generalization/specialization.
    - Should be maximized (usually).

35. What are the different types of cohesion? **Give an example of each.**
    - *Method cohesion* addresses the cohesion within an individual method (i.e., how single-minded a method is). Methods should do one and only one thing. Should be maximized. (**Appendix B**)
    - *Class cohesion* is the level of cohesion among the attributes and methods of a class (i.e., how single-minded a class is). A class should represent only one thing, such as an employee, a department, or an order. (**Appendix C**)
        i. All attributes and methods contained in a class should be required for the class to represent the thing (employee should have an attribute for name, but not for type of door).
    - *Generalization/specialization cohesion* addresses the sensibility of the inheritance hierarchy. Highly cohesive inheritance hierarchies should support only the semantics of generalization and specialization (a-kind-of) and the principle of substitutability.
        i. Are the classes related through a generalization/specialization (a-kind-of) semantics? (good)
        ii. Or, are they related via some association, aggregation, or membership type of relationship that was created for simple reuse purposes? (bad)

36. What is **connascence**? (types of connascence in **Appendix D**)
    - *Connascence* generalizes the ideas of cohesion and coupling, and it combines them with the arguments for encapsulation. 3 levels:
        i. Level-0 encapsulation refers to the amount of encapsulation realized in an individual line of code.
        ii. Level-1 encapsulation is the level of encapsulation attained by combining lines of code into a method
        iii. Level-2 encapsulation is achieved by creating classes that contain both methods and attributes.
    - Connascence literally means to be born together. From an object-oriented design perspective, it really means that two modules (classes or methods) are so intertwined that if you make a change in one, it is likely that a change in the other will be required (in a sense, similar to coupling).
    - We want to *minimize* overall connascence by eliminating any unnecessary connascence throughout the system; *minimize* connascence **across** any encapsulation boundaries (e.g., a subclass accessing a hidden attribute of a

superclass), such as method boundaries and class boundaries; and *maximize* connascence **within** any encapsulation boundary.

37. List, explain and give examples the object design activities.

- **Adding Specifications**: One object design activity is simply adding more detail to our current set of functional, structural and behavioral models. Steps:
- First, we should ensure that the classes on the problem-domain layer are both necessary and sufficient to solve the underlying problem.
    i. To do this, we need to be sure that there are no missing attributes or methods and no extra or unused attributes or methods in each class, or missing classes.
- Second, we need to finalize the visibility (hidden or visible) of the attributes and methods in each class
- Third, we need to decide on the signature of every method in every class.
    i. The *signature* of a method comprises three parts: the name of the method, the parameters or arguments that must be passed to the method, including their object type, and the type of value that the method will return to the calling method.
- Fourth, we need to define any constraints (and consequences of violations to these) that must be preserved by the objects (e.g., an attribute of an object that can have values only in a certain range).
    i. There are three different types of constraints: preconditions, postconditions, and invariants.
- **Identifying Opportunities for Reuse**: There are opportunities for using existing design patterns, frameworks, libraries, and components.
    i. Design patterns are simply useful grouping of collaborating classes that provide a solution to a commonly occurring problem.
    ii. A *framework* is composed of a set of implemented classes that can be used as a basis for implementing an application.
    iii. A *class library* is similar to a framework in that it typically has a set of implemented classes that were designed for reuse. However, frameworks tend to be more domain specific.
    iv. A *component* is a self-contained, encapsulated piece of software that can be plugged into a system to provide a specific set of required functionalities.
- **Restructuring the Design:** Once the individual classes and methods have been specified and the class libraries, frame- works, and components have been incorporated into the evolving design, we should use factoring to restructure the design, and we might also use normalization. Finally, all inheritance relationships should be challenged to ensure that they sup- port only a generalization/specialization (a-kind-of) semantics.
    i. *Factoring* (Chapter 7) is the process of separating out aspects of a method or class into a new method or class to simplify the overall design
- **Optimizing the Design**: Now, a good practical design manages the inevitable trade-offs between understandability and efficiency that must occur.

- **Mapping Problem-Domain Classes to Implementation Languages:** It is important to map the current design to the capabilities of the programming language used. For example, if we have used multiple inheritance in our design but we are implementing in a language that supports only single inheritance, then the multiple inheritance must be factored out of the design

38. What is a useful tool for this process?

- Design patterns

39. Available to incorporate as a class library?

- Frameworks? Or perhaps components?

40. What are design patterns?

- Design patterns are simply useful grouping of collaborating classes that provide a solution to a commonly occurring problem.
- The primary difference between analysis and design patterns is that design patterns are useful in solving "a general design problem in a particular context," whereas analysis patterns tended to aid in filling out a problem-domain representation.

41. What are frameworks?

- A *framework* is composed of a set of implemented classes that can be used as a basis for implementing an application.
- Most frameworks allow us to create subclasses to inherit from classes in the framework.
- When inheriting from classes in a framework, we are creating a dependency, which might cause a problem down the line.

42. What are libraries and components?

- A *class library* is similar to a framework in that it typically has a set of implemented classes that were designed for reuse. However, frameworks tend to be more domain specific.
    i. Instances of classes contained in the class library can be created, and in other cases, classes in the class library can be extended by creating subclasses based on them (but creating coupling and connascence).
- A *component* is a self-contained, encapsulated piece of software that can be plugged into a system to provide a specific set of required functionalities.
    i. A component has a well-defined *API* (application pro- gram interface). An API is essentially a set of method interfaces to the objects contained in the component.
    ii. Recompilation is typically not required.

43. How do you integrate these into your software design process?

- Frameworks are used mostly to aid in developing objects on the physical architecture, human–computer interaction, or data management layers.

- Components are used primarily to simplify the development of objects on the problem-domain and human– computer interaction layers.
- Class libraries are used to develop frameworks and components and to support the foundation layer.

44. What is factoring?
    - *Factoring* (Chapter 7) is the process of separating out aspects of a method or class into a new method or class to simplify the overall design.
    - For example, when reviewing a set of classes on a particular layer, we might discover that a subset of them shares a similar definition (could cause coupling or connascence).

45. What is normalization of a class diagram? (**Appendix E**)
    - Normalizing a class diagram means the act of making a class into a 'normal class' (meaning a class where all association and aggregation relationships are converted to attributes in the classes).
    - Steps:
    - First, convert association classes to a normal class (in the diagram).
    - Then, convert all associations to attributes that represent the relationships between the affected classes.

46. How do you optimize the design, what do you look for?
    - To optimize the design, we look for efficiency:
    - The first optimization to consider is to review the access paths between objects.
        i. If the path is long and the message is sent frequently, a redundant path should be considered.
        ii. Adding an attribute to the calling object that will store a direct connection to the object at the end of the path can accomplish this.
    - A second optimization is to review each attribute of each class.
        i. It should be determined which methods use the attributes and which objects use the methods.
        ii. If the only methods that use an attribute are read and update methods and only instances of a single class send messages to read and update the attribute, then the attribute may belong with the calling class instead of the called class.
        iii. Moving the attribute to the calling class will substantially speed up the system.
    - A third optimization is to review the direct and indirect fan-out of each method.
        i. *Fan-out* refers to the number of messages sent by a method.
        ii. If the fan-out of a method is high relative to the other methods in the system, the method should be optimized.
        iii. One way to do this is to consider adding an index to the attributes used to send the messages to the objects in the message tree.
    - A fourth optimization is to look at the execution order of the statements in often-used methods.
        i. It is possible to rearrange some of the statements to be more efficient.

- A fifth optimization is to avoid re-computation by creating a *derived attribute* (or *active value*) (e.g., a total that stores the value of the computation).
    i. This is also known as *caching computational results,* and it can be accomplished by adding a *trigger* to the attributes contained in the computation.
- A sixth optimization that should be considered deals with objects that participate in a one- to-one association; that is, they both must exist for either to exist. In this case, it might make sense, for efficiency purposes, to collapse the two defining classes into a single class.
    i. Alternatively, it could make more sense for the two classes to be combined on the problem-domain layer but kept separate on the data management layer (b/c the collapsed class could be too big for the data management).

47. What is the process of mapping problem domain classes to implementation languages?
    - Basically, *mapping problem domain classes to implementation languages* means adapting our design to the capabilities of the languages we are to use.
        i. For example, if we have used multiple inheritance in our design but we are implementing in a language that supports only single inheritance, then the multiple inheritance must be factored out of the design
    - Cases:
    - **Implementing Problem Domain Classes in a Single-Inheritance Language** The only issue associated with implementing problem-domain objects is the factoring out of any multiple inheritance—i.e., the use of more than one superclass—used in the evolving design.
        i. **RULE 1a:** Convert the additional inheritance relationships to association relationships. (**Appendix F**)
        ii. ***Or***
        iii. **RULE 1b:** Flatten the inheritance hierarchy by copying the attributes and methods of the additional superclass(es) down to all of the subclasses and remove the additional superclass from the design. (**Appendix F**)

    - **Implementing Problem Domain Objects in an Object-Based Language** If we are going to implement our solution in an *object-based language* (i.e., a language that supports the creation of objects but does not support implementation inheritance), we must factor out all uses of inheritance from the problem-domain class design. Basically, you can use **Rule 1a** or **Rule1b** (this could lead to inheritance conflict) again. (**Appendix G)**
    - **Implementing Problem-Domain Objects in a Traditional Language** From a practical perspective, we are much better off implementing an object-oriented design in an object-oriented programming language. The best advice that we can give about implementing an object-oriented design in a traditional programming language is to run away as fast and as far as possible from the project.

48. What is a contract?

- A contract formalizes the interactions between the client and server objects, where a *client* (*consumer*) object is an instance of a class that sends a message to a *server* (*supplier*) object that executes one of its methods in response to the request.
    i. Contracts are modeled on the legal notion of a contract, where both parties, client and server objects, have obligations and rights. Practically speaking, a contract is a set of constraints and guarantees

49. What the contract is for?
    - Contracts document the message passing that takes place between objects.
    - In practice, a contract is created for each method that can receive messages from other objects (i.e., one for each visible method).
    - A contract should thus contain the information necessary for a programmer to understand what a method is to do

50. What are the constraints of contracts?
    - Constraints are the limitations of contracts.
    - If the constraints are met, then the server object guarantees certain behavior. Constraints can be written in a natural language (e.g., English), a semiformal language (e.g., *Structured English*32), or a formal language (e.g., UML's Object Constraint Language, recommended).

51. What are the types of constraints, define and give an example?
    - Three different types of constraints are typically captured in object-oriented design: preconditions, postconditions, and invariants.
        i. A *precondition* is a constraint that must be met **for** a method to execute.
            1. For example, the parameters passed to a method must be valid for the method to execute.
        ii. A *postcondition* is a constraint that must be met **after** the method executes, or the effect of the method execution must be undone.
            1. For example, the method cannot make any of the attributes of the object take on an invalid value. In this case, an exception should be raised, and the effect of the method's execution should be undone.
        iii. *Invariants* model constraints that must always be true for all instances of a class (it's in the name; constraint can't vary).
            1. Examples of invariants include domains or types of attributes, multiplicity of attributes, and the valid values of attributes

52. What are the elements of a contract? (**Appendix H**)
    - A contract is composed of the information required for the developer of a client object to know what messages can be sent to the server objects and what the client can expect in return:
        i. Method Name, Class name, ID, Clients(Consumers), Associated Use Cases, Descriptions of Responsibilities, Arguments Received, Type of Value Returned, Pre-Conditions, Post-Conditions.

53. What are invariants?
    - *Invariants* model constraints that must always be true for all instances of a class (it's in the name; constraint can't vary).
    - Examples of invariants include domains or types of attributes, multiplicity of attributes, and the valid values of attributes.
    - **Order class invariants:**
        i. Cust ID = Customer.GetCustID()
        ii. State Name = Sate.GetState()
        iii. Sub Total = ProductOrder.sum(GetExtension())
        iv. Tax = State.GetTaxRate() *Sub Total1

54. How do you create a method specification?
    - *Method specifications* are written documents that include explicit instructions on how to write the code to implement the method. (**Appendix I**)
    - Typical method specification forms contain four components you need to create method specifications. The four components are:
        i. General Information: This information is used to help manage the programming effort.
        ii. Events: An *event* is a thing that happens or takes place.
        iii. Message Passing: *Message passing* means the messages the pass to and from the method, which are identified on the sequence and collaboration diagrams.
        iv. Algorithm Specification: The representation of the process that the method has to make, usually through the use of Structured English.
        v. Miscellaneous: Represents any extra notes that the programmer may need.

55. What are events?
    - An *event* is a thing that happens or takes place.
    - Clicking the mouse generates a mouse event, pressing a key generates a keystroke event—in fact, almost everything the user does generates an event.

56. How do you include them in your methods specification?
    - By including them in the 'Event' section of the method specification form.
    - Alternatively, you can use the behavioral state machines.

57. What does "message passing" mean?
    - *Message passing* means the messages the pass to and from the method, which are identified on the sequence and collaboration diagrams.
    - Programmers need to understand what arguments are being passed into, passed from, and returned by the method because the arguments ultimately translate into attributes and data structures within the actual method.

58. What is an algorithm specification?
    - *Algorithm specification* is the representation of the process that the method has to make, usually through the use of Structured English.

59. What syntax can you use for algorithm specifications?
    - *Structured English* (a formal way of writing instructions that describe the steps of a process.)
    - *Or*
    - Activity diagrams

60. When would you choose an algorithm specification and why?
    - What does this mean?

61. How do you verify and validate class and method design?
    - Virtually everything must be re-verified and re-validated.
    - First, we recommend that a walkthrough of all of the evolved problem domain representations be performed.
        i. That is, all functional models (Chapter 4) must be consistent; all structural models (Chapter 5) must be consistent; all behavioral models (Chapter 6) must be consistent; and the functional, structural, and behavioral models must be balanced (Chapter 7).
    - Second, all constraints, contracts, and method specifications must be tested.
        i. The best way to do this is to role-play the system using the different scenarios of the use cases.

Record: 43 questions!

Appendix A

Level	Type	Description
Good ↓	No Direct Coupling	The methods do not relate to one another; that is, they do not call one another.
	Data	The calling method passes a variable to the called method. If the variable is composite (i.e., an object), the entire object is used by the called method to perform its function.
	Stamp	The calling method passes a composite variable (i.e., an object) to the called method, but the called method only uses a portion of the object to perform its function.
	Control	The calling method passes a control variable whose value will control the execution of the called method.
	Common or Global	The methods refer to a "global data area" that is outside the individual objects.
Bad	Content or Pathological	A method of one object refers to the inside (hidden parts) of another object. This violates the principles of encapsulation and information hiding. However, C++ allows this to take place through the use of "friends."

Source: These types are based on material from Meilir Page-Jones, *The Practical Guide to Structured Systems Design,* 2nd Ed. (Englewood Cliffs, NJ: Yardon Press, 1988); Glenford Myers, *Composite/ Structured Design* (New York: Van Nostrand Reinhold, 1978).

FIGURE 8-7
Types of Interaction Coupling

Appendix B

Level	Type	Description
Good	Functional	A method performs a single problem-related task (e.g., calculate current GPA).
	Sequential	The method combines two functions in which the output from the first one is used as the input to the second one (e.g., format and validate current GPA).
	Communicational	The method combines two functions that use the same attributes to execute (e.g., calculate current and cumulative GPA).
	Procedural	The method supports multiple weakly related functions. For example, the method could calculate student GPA, print student record, calculate cumulative GPA, and print cumulative GPA.
	Temporal or Classical	The method supports multiple related functions in time (e.g., initialize all attributes).
	Logical	The method supports multiple related functions, but the choice of the specific function is chosen based on a control variable that is passed into the method. For example, the called method could open a checking account, open a savings account, or calculate a loan, depending on the message that is sent by its calling method.
Bad	Coincidental	The purpose of the method cannot be defined or it performs multiple functions that are unrelated to one another. For example, the method could update customer records, calculate loan payments, print exception reports, and analyze competitor pricing structure.

Source: These types are based on material from Page-Jones, *The Practical Guide to Structured Systems*; Myers, *Composite/Structured Design*; Edward Yourdon and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Englewood Cliffs, NJ: Prentice-Hall, 1979).

FIGURE 8-9
Types of Method Cohesion

Appendix C

Level	Type	Description
Good	Ideal	The class has none of the mixed cohesions.
	Mixed-Role	The class has one or more attributes that relate objects of the class to other objects on the same layer (e.g., the problem domain layer), but the attribute(s) has nothing to do with the underlying semantics of the class.
	Mixed-Domain	The class has one or more attributes that relate objects of the class to other objects on a different layer. As such, they have nothing to do with the underlying semantics of the thing that the class represents. In these cases, the offending attribute(s) belongs in another class located on one of the other layers. For example, a port attribute located in a problem domain class should be in a system architecture class that is related to the problem domain class.
Worse	Mixed-Instance	The class represents two different types of objects. The class should be decomposed into two separate classes. Typically, different instances only use a portion of the full definition of the class.

Based upon material from Page-Jones, *Fundamentals of Object-Oriented Design in UML*.

FIGURE 8-10
Types of Class Cohesion

Appendix D

Type	Description
Name	If a method refers to an attribute, it is tied to the name of the attribute. If the attribute's name changes, the content of the method will have to change.
Type or Class	If a class has an attribute of type A, it is tied to the type of the attribute. If the type of the attribute changes, the attribute declaration will have to change.
Convention	A class has an attribute in which a range of values has a semantic meaning (e.g., account numbers whose values range from 1000 to 1999 are assets). If the range would change, then every method that used the attribute would have to be modified.
Algorithm	Two different methods of a class are dependent on the same algorithm to execute correctly (e.g., insert an element into an array and find an element in the same array). If the underlying algorithm would change, then the insert and find methods would also have to change.
Position	The order of the code in a method or the order of the arguments to a method is critical for the method to execute correctly. If either is wrong, then the method will, at least, not function correctly.

Based upon material from Meilir Page-Jones, "Comparing Techniques by Means of Encapsulation and Connascence" and Meilir Page-Jones, *Fundamentals of Object-Oriented Design in UML*.

FIGURE 8-12
Types of Connascence

Appendix E

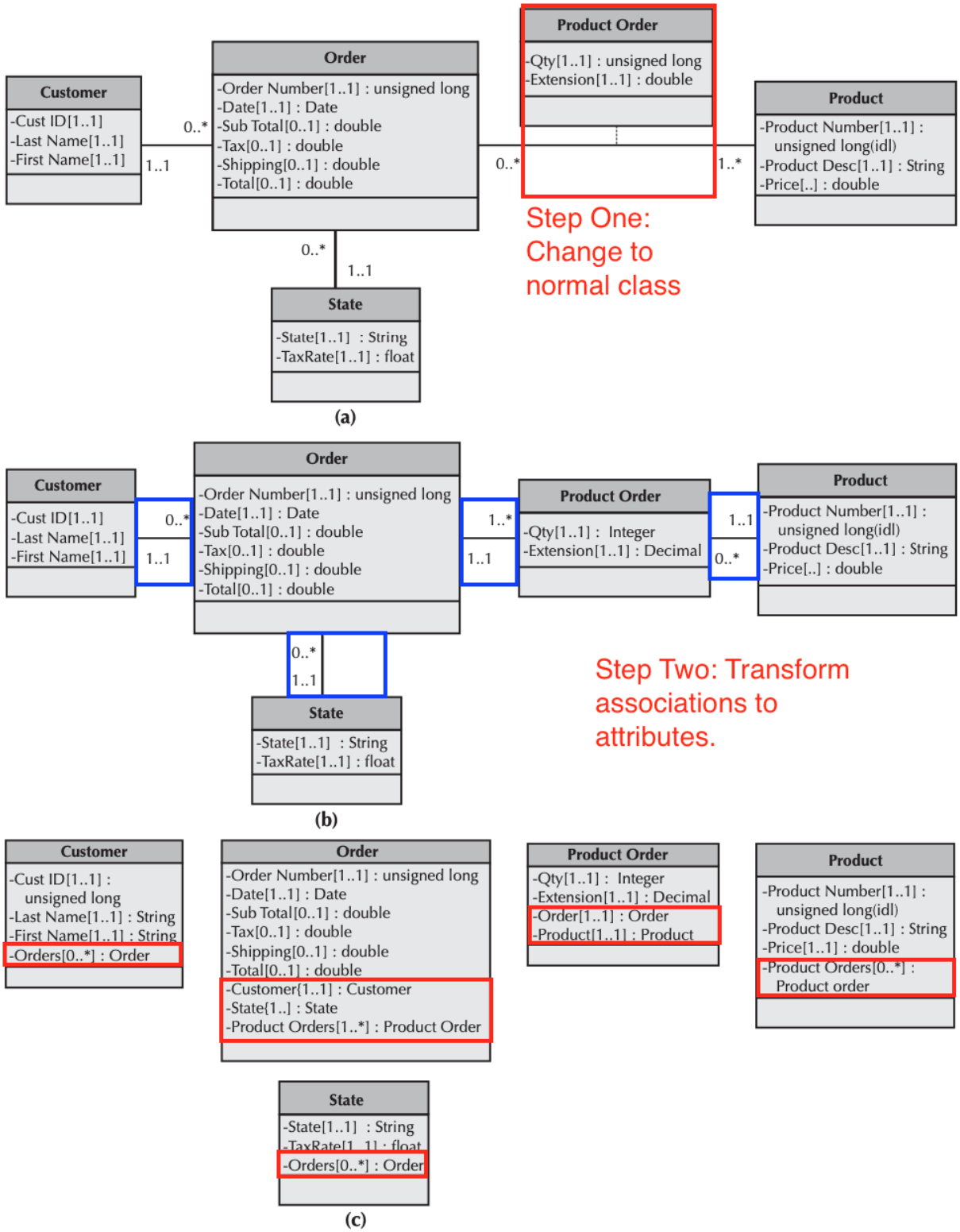


FIGURE 8-15 Converting Associations to Attributes

Appendix F

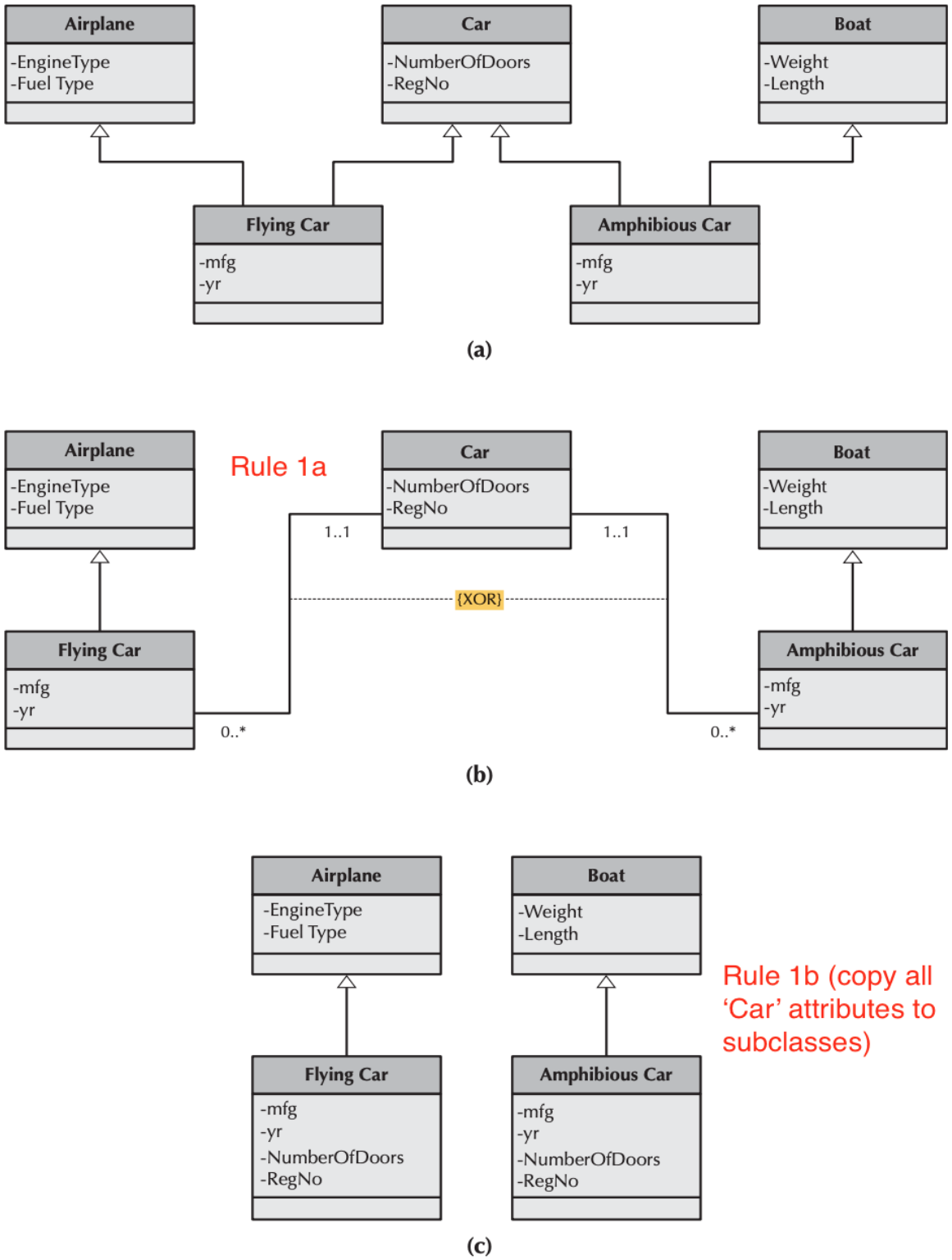


FIGURE 8-16 Factoring Out Multiple-Inheritance Effect for a Single-Inheritance Language

Appendix G

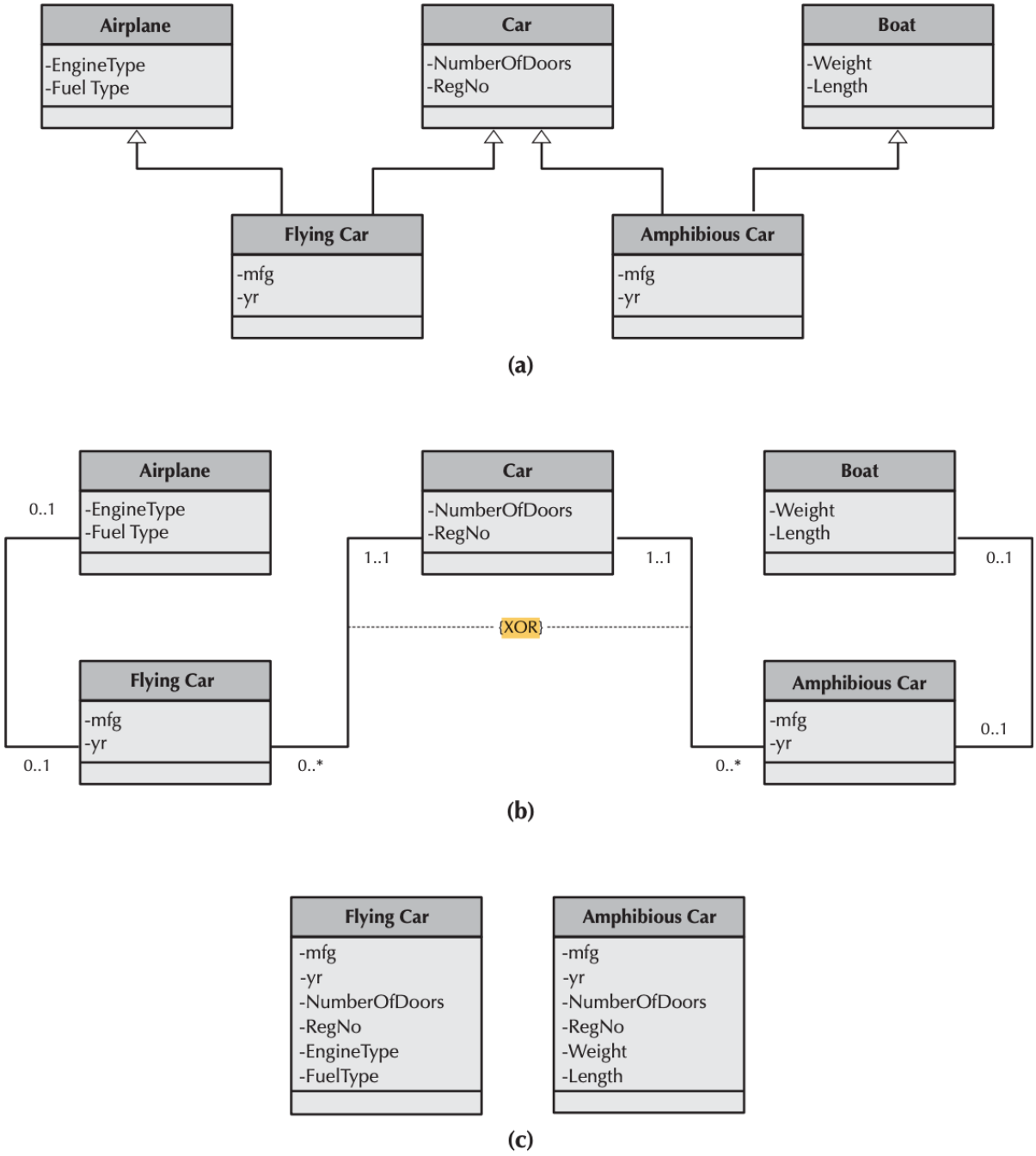


FIGURE 8-17 Factoring Out Multiple Inheritance Effect for an Object-Based Language

Appendix H

Method Name:	Class Name:	ID:
Clients (Consumers):		
Associated Use Cases:		
Description of Responsibilities:		
Arguments Received:		
Type of Value Returned:		
Pre-Conditions:		
Post-Conditions:		

FIGURE 8-22
Sample Contract Form

Appendix I

i. General Information

Method Name:	Class Name:	ID:
Contract ID:	Programmer:	Date Due:
Programming Language: ❑ Visual Basic ❑ Smalltalk ❑ C++ ❑ Java		

ii. Events

Triggers/Events:

iii. Message Passing

Arguments Received: Data Type:	Notes:

Messages Sent & Arguments Passed: ClassName.MethodName:	Data Type:	Notes:

Arguments Returned: Data Type:	Notes:

iv. Algorithm Spec.

Algorithm Specification:

v. Misc.

Misc. Notes:

FIGURE 8-26 Method Specification Form

Chapter 9

1. Based on the **chapter nine lecture**: which are the two most promising object persistence formats? (**Appendix A**)
    - Object relational databases
    - NoSQL data stores
    - ~~Multimodal DBMS? (support relational and non-relational)~~

2. What does "*mapping the problem domain object to object persistence format*" mean?
    - *Mapping the problem domain object to object persistence format* is when objects must be converted so that they can be stored in a table. Basically, it's when we have to map our problem domain layer to our database management layer.
    - Tips:
        i. Supporting primary keys and foreign keys is important, so add them to your problem domain classes at this point.
        ii. We also recommend that data management functionality specifics, such as retrieval and updating of data from the object storage, be included only in classes contained in the data management layer.

    - It has the following steps (for Relational DataBase Management System):
        i. Create normalized RDBMS Schema
        ii. Create mapping between Problem Domain classes and RDBMS Schema (see guidelines in book)
        iii. Create data access and manipulation (DAM) Classes

    - Rules are in **Appendix B**:
        i. The first four rules are basically the same set of rules used to map problem domain objects to ORDBMS-based data management objects

3. What is the main way of optimizing storage efficiency?
    - *Normalization* is a process whereby a series of rules are applied to the RDBMS tables to assess the efficiency of the tables. These rules help analysts identify tables that are not represented correctly. Forms:
    - 0 Normal Form, which is an unnormalized model before the normalization rules have been applied.
    - A model is in *first normal form (1NF)* if it does not lead to multivalued fields, fields that allow a set of values to be stored, or repeating fields, which are fields that repeat within a table to capture multiple values.
        i. The rule for 1NF says that all tables must contain the same number of columns (i.e., fields) and that all the columns must contain a single value.
    - *Second normal form (2NF)* requires first that the data model is in 1NF and second that the data model leads to tables containing fields that depend on a *whole* primary key.

    i. This means that the primary key value for each record can determine the value for all the other fields in the record
  - *Third normal form (3NF)* occurs when a model is in both 1NF and 2NF and, in the resulting tables, none of the fields depend on nonprimary key fields (i.e., *transitive dependency*).

4. What are some ways of data access speed?
  - **Denormalization** After the object storage is optimized, the project team may decide that increased data retrieval speed is more important than storage efficiency or data update speed and elect to denormalize or add redundancy back into the design. *Denormalization* reduces the number of joins that need to be performed in a query, thus speeding up access.
    i. There are three cases in which you may rely upon denormalization to reduce joins and improve performance:
    ii. First, denormalization can be applied in the case of look-up tables, which are tables that contain descriptions of values
    iii. Second, one-to-one relationships are good candidates for denormalization.
    iv. Third, at times it is more efficient to include a parent entity's attributes in its child entity on the physical data mode.
  - **Clustering** One way to improve access speed is to reduce the number of times that the storage medium needs to be accessed during a transaction. One method is to *cluster* records together physically so that similar records are stored close together.
    i. With ***intrafile*** *clustering,* like records in the table are stored together in some way, such as in order by primary key or, in the case of a grocery store, by item type.
    ii. ***Interfile*** *clustering* combines records from more than one table that typically are retrieved together.
  - **Indexing** A familiar time saver is an index located in the back of a textbook, which points directly to the page or pages that contain a topic of interest.
    i. An *index* in data storage is like an index in the back of a textbook; it is a minitable that contains values from one or more columns in a table and the location of the values within the table.
    ii. Indexes are one of the most important ways to improve database performance.
    iii. Guidelines:
      1. Use indexes sparingly for transaction systems.
      2. Use many indexes to increase response times in decision support systems.
      3. For each table, create a unique index that is based on the primary key.
      4. For each table, create an index that is based on the foreign key to improve the performance of joins.

5. Create an index for fields that are used frequently for grouping, sorting, or criteria.

iv. Note that these slow down the update process.

5. How do you estimate data Storage size? Please refer to the book on **page 356**. (example on **Appendix C**)
    - Through *volumetrics,* which means estimating the amount of data that the hardware will need to support.
    - The size of the database is based on the amount of *raw data* in the tables and the *overhead* requirements of the DBMS. To estimate size, you will need to have a good understanding of the initial size of your database as well as its expected growth rate over time.
    - *Raw data* refers to all the data that are stored within the tables of the database, and it is calculated based on a bottom-up approach:
        i. First, write down the estimated average width (as in the avg value) for each column (field) in the table and sum the values for a total record size
        ii. Next, calculate the overhead for the table as a percentage of each record
        iii. Finally, record the number of initial records that will be loaded into the table, as well as the expected growth per month.

6. What do data access and manipulation classes do?
    - *Data access and manipulation classes* act as a translator between the object persistence and the problem domain objects.
    - Thus, they should always be capable of at least reading and writing both the object persistence and problem domain objects.
    - A simple rule to follow is that there should be one data access and manipulation class for each **concrete** problem domain class.
    - One specific approach that has been suggested to support the implementation of data access and manipulation classes is to use an object-relational mapping library such as Hibernate.

7. What are the non-functional requirements for the data management layer design?
    - The *operational requirements* for the data management layer include issues that deal with the technology being used to support object persistence (hardware and OS).
    - The primary *performance requirements* that affect the data management layer are speed and capacity.
    - The *security requirements* deal primarily with access controls, encryption, and backup.
    - There are few *political and cultural requirements* that can affect the data management layer. These include issues related to the expected number of

characters that should be allocated for a data field, the format of a data field, and the issues related to security.

i. Finally, there could be a corporate IT bias toward different hardware and software platforms.

8. How do you verify and validate the data management layer?
    - Verifying and validating the design of the data management layer falls into three basic groups:
        i. First, we recommend verifying and validating any changes made to the problem domain by performing walkthroughs of the modified functional models (Chapter 4), structural models (Chapter 5), and behavioral models (Chapter 6) .
            1. Furthermore, all of the models must be consistent and balanced (Chapter 7).
            2. And, if any problem domain class was modified that was associated with a use-case scenario, that scenario should be tested again through role-playing.
        ii. Dependency of the object persistence instances on the problem domain must be enforced.
        iii. Third, the design of the data access and manipulation classes need to be tested to ensure that they are dependent on the problem domain classes and the object persistence format, not the other way around

Appendix A

	Sequential and Random Access Files	Relational DBMS	Object Relational DBMS	Object-Oriented DBMS	NoSQL data store
Major Strengths	Usually part of an object-oriented programming language Files can be designed for fast performance Good for short-term data storage	Leader in the database market Can handle diverse data needs	Based on established, proven technology, e.g., SQL Able to handle complex data	Able to handle complex data Direct support for object orientation	Able to handle complex data
Major Weaknesses	Redundant data Data must be updated using programs, i.e., no manipulation or query language No access control	Cannot handle complex data No support for object orientation Impedance mismatch between tables and objects	Limited support for object orientation Impedance mismatch between tables and objects	Technology is still maturing Skills are hard to find	Technology is still maturing Skills are hard to find
Data Types Supported	Simple and Complex	Simple	Simple and Complex	Simple and Complex	Simple and Complex
Types of Application Systems Supported	Transaction processing	Transaction processing and decision making	Transaction processing and decision making	Transaction processing and decision making	Primarily decision making
Existing Storage Formats	Organization dependent	Organization dependent	Organization dependent	Organization dependent	Organization dependent
Future Needs	Poor future prospects	Good future prospects	Good future prospects	At End of life (Gartner)	Good future prospects

Appendix B

Rule 1: Map all concrete-problem domain classes to the RDBMS tables. Also, if an abstract Problem Domain class has multiple direct subclasses, map the abstract class to a RDBMS table.

Rule 2: Map single-valued attributes to columns of the tables.

Rule 3: Map methods to stored procedures or to program modules.

Rule 4: Map single-valued aggregation and association relationships to a column that can store the key of the related table, i.e., add a foreign key to the table. Do this for both sides of the relationship.

Rule 5: Map multivalued attributes and repeating groups to new tables and create a one-to-many association from the original table to the new ones.

Rule 6: Map multivalued aggregation and association relationships to a new associative table that relates the two original tables together. Copy the primary key from both original tables to the new associative table, i.e., add foreign keys to the table.

Rule 7: For aggregation and association relationships of mixed type, copy the primary key from the single-valued side (1..1 or 0..1) of the relationship to a new column in the table on the multivalued side (1..* or 0..*) of the relationship that can store the key of the related table, i.e., add a foreign key to the table on the multivalued side of the relationship.

For generalization/inheritance relationships:

Rule 8a: Ensure that the primary key of the subclass instance is the same as the primary key of the superclass. The multiplicity of this new association from the subclass to the "superclass" should be 1..1. If the superclasses are concrete, that is, they can be instantiated themselves, then the multiplicity from the superclass to the subclass is 0..1, otherwise, it is 1..1. Furthermore, an exclusive-or (XOR) constraint must be added between the associations. Do this for each superclass.

or

Rule 8b: Flatten the inheritance hierarchy by copying the superclass attributes down to all of the subclasses and remove the superclass from the design.*

* It is also a good idea to document this modification in the design so that in the future, modifications to the design can be maintained easily.

FIGURE 9-9 Schema for Mapping Problem Domain Objects to RDBMS

Appendix C

Step 1: Calculate average width.

Step 2: Calculate the overhead and new record size

Step 3: Calculate initial table size (estimate) and apply growth rate.

Field	Average Size
Order Number	8
Date	7
Cust ID	4
Last Name	13
First Name	9
State	2
Amount	4
Tax Rate	2
Record Size	49
Overhead	30%
Total Record Size	63.7
Initial Table Size	50,000
Initial Table Volume	3,185,000
Growth Rate/Month	1,000
Table Volume @ 3 years	5,478,200

FIGURE 9-20
Calculating Volumetrics