Module 8

Object-Oriented Software Development

Lesson 19

Domain Modeling

Specific Instructional Objectives

At the end of this lesson the student will be able to:

- Explain what is meant by domain modeling.
- Identify the three types of objects identified during domain analysis.
- Explain the purpose of different types of objects identified during domain analysis. Explain how these objects interact among each other.
- Explain at least three approaches for identifying objects in the context of object-oriented design methodology.
- Identify two goals of interaction modeling.
- Explain the CRC cards technique.
- Develop sequence diagram for any given use case.
- Identify how sequence diagrams are useful in developing the class diagram.
- Identify five important criteria for judging the goodness of an object-oriented design.

Explain what is meant by domain modeling.

Domain modeling is known as conceptual modeling. A domain model is a representation of the concepts or objects appearing in the problem domain. It also captures the obvious relationships among these objects. Examples of such conceptual objects are the Book, BookRegister, MemeberRegister, LibraryMember, etc. The recommended strategy is to quickly create a rough conceptual model where the emphasis is in finding the obvious concepts expressed in the requirements while deferring a detailed investigation. Later during the development process, the conceptual model is incrementally refined and extended.

Identify the three types of objects identified during domain analysis.

The objects identified during domain analysis can be classified into three types:

- Boundary objects
- Controller objects
- Entity objects

The boundary and controller objects can be systematically identified from the use case diagram whereas identification of entity objects requires practice. So, the crux of the domain modeling activity is to identify the entity models.

Explain the purpose of different types of objects identified during domain analysis. Explain how these objects interact among each other.

The different kinds of objects identified during domain analysis and their relationships are as follows:

Boundary objects: The boundary objects are those with which the actors interact. These include screens, menus, forms, dialogs, etc. The boundary objects are mainly responsible for user interaction. Therefore, they normally do not include any processing logic. However, they may be responsible for validating inputs, formatting, outputs, etc. The boundary objects were earlier being called as the interface objects. However, the term interface class is being used for Java, COM/DCOM, and UML with different meaning. A recommendation for the initial identification of the boundary classes is to define one boundary class per actor/use case pair.

Entity objects: These normally hold information such as data tables and files that need to outlive use case execution, e.g. Book, BookRegister, LibraryMember, etc. Many of the entity objects are "dumb servers". They are normally responsible for storing data, fetching data, and doing some fundamental kinds of operation that do not change often.

Controller objects: The controller objects coordinate the activities of a set of entity objects and interface with the boundary objects to provide the overall behavior of the system. The responsibilities assigned to a controller object are closely related to the realization of a specific use case. The controller objects effectively decouple the boundary and entity objects from one another making the system tolerant to changes of the user interface and processing logic. The controller objects embody most of the logic involved with the use case realization (this logic may change time to time). A typical interaction of a controller object with boundary and entity objects is shown in fig. 8.2. Normally, each use case is realized using one controller object. However, some use cases can be realized without using any controller object, i.e. through boundary and entity objects only. This is often true for use cases that achieve only some simple manipulation of the stored information.

For example, let's consider the "query book availability" use case of the Library Information System (LIS). Realization of the use case involves only matching the given book name against the books available in the catalog. More complex use cases may require more than one controller object to realize the use case. A complex use case can have several controller objects such as transaction manager, resource coordinator, and error handler. There is another situation where a use case can have more than one controller object. Sometimes the use cases require the controller object to transit through a number of states.

In such cases, one controller object might have to be created for each execution of the use case.

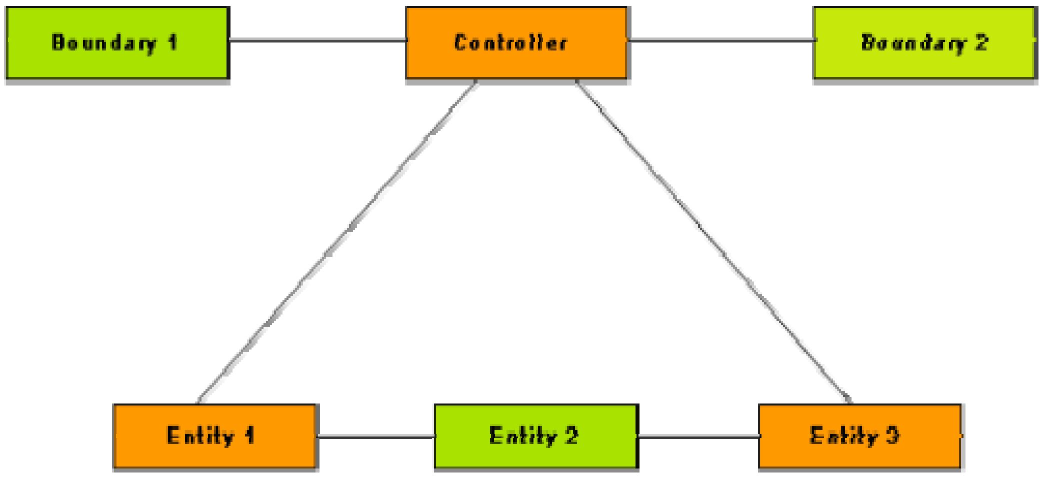


Fig. 8.2: A typical realization of a use case through the collaboration of boundary, controller, and entity objects

Explain at least three approaches for identifying objects in the context of object-oriented design methodology.

One of the most important steps in any object-oriented design methodology is the identification of objects. In fact, the quality of the final design depends to a great extent on the appropriateness of the objects identified. However, to date no formal methodology exists for identification of objects. Several semi-formal and informal approaches have been proposed for object identification. These can be classified into the following broad classes**:**

- Grammatical analysis of the problem description.
- Derivation from data flow.
- Derivation from the entity relationship (E-R) diagram.

A widely accepted object identification approach is the grammatical analysis approach. Grady Booch originated the grammatical analysis approach [1991]. In Booch's approach, the nouns occurring in the extended problem description statement (processing narrative) are mapped to objects and the verbs are mapped to methods. The identification approaches based on derivation from the data flow diagram and the entity-relationship model are still evolving and therefore will not be discussed in this text.

Booch's Object Identification Method

Booch's object identification approach requires a processing narrative of the given problem to be first developed. The processing narrative describes the problem and discusses how it can be solved. The objects are identified by noting down the nouns in the processing narrative. Synonym of a noun must be eliminated. If an object is required to implement a solution, then it is said to be part of the solution space. Otherwise, if an object is necessary only to describe the problem, then it is said to be a part of the problem space. However, several of the nouns may not be objects. An imperative procedure name, i.e., noun form of a verb actually represents an action and should not be considered as an object. A potential object found after lexical analysis is usually considered legitimate, only if it satisfies the following criteria**:**

Retained information. Some information about the object should be remembered for the system to function. If an object does not contain any private data, it can not be expected to play any important role in the system.

Multiple attributes. Usually objects have multiple attributes and support multiple methods. It is very rare to find useful objects which store only a single data element or support only a single method, because an object having only a single data element or method is usually implemented as a part of another object.

Common operations. A set of operations can be defined for potential objects. If these operations apply to all occurrences of the object, then a class can be defined. An attribute or operation defined for a class must apply to each instance of the class. If some of the attributes or operations apply only to some specific instances of the class, then one or more subclasses can be needed for these special objects.

Normally, the actors themselves and the interactions among themselves should be excluded from the entity identification exercise. However, some times there is a need to maintain information about an actor within the system. This is not the same as modeling the actor. These classes are sometimes called surrogates. For example, in the Library Information System (LIS) we would need to store information about each library member. This is independent of the fact that the library member also plays the role of an actor of the system.

Although the grammatical approach is simple and intuitively appealing, yet through a naive use of the approach, it is very difficult to achieve high quality results. In particular, it is very difficult to come up with useful abstractions simply by doing grammatical analysis of the problem description. Useful abstractions usually result from clever factoring of the problem description into independent and intuitively correct elements.

Example: Tic-tac-toe

Let us identify the entity objects of the following Tic-tac-toe software:

Tic-tac-toe is a computer game in which a human player and the computer **make** alternative moves on a 3 X 3 square. A move consists of **marking** a previously unmarked square. A player who first **places** three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square **wins**. As soon as either the human player or the computer **wins**, a message congratulating the winner should be **displayed**. If neither player manages to **get** three consecutive marks along a straight line, and all the squares on the board are filled up, then the game is **drawn**. The computer always **tries to win** a game.

By performing a grammatical analysis of this problem statement, it can be seen that nouns that have been underlined in the problem description and the actions as the italicized verbs. However, on closer examination synonyms can be eliminated from the identified nouns. The list of nouns after eliminating the synonyms are the following: Tic-tac-toe, computer game, human player, move, square, mark, straight line, board, row, column, and diagonal.

From this list of possible objects, nouns can be eliminated like human player as it does not belong to the problem domain. Also, the nouns square, game, computer, Tic-tac-toe, straight line, row, column, and diagonal can be eliminated, as any data and methods can not be associated with them. The noun **move** can also be eliminated from the list of potential objects since it is an imperative verb and actually represents an action. Thus, there is only one object left – board.

After experienced in object identification, it is not normally necessary to really identify all nouns in the problem description by underlining them or actually listing them down, and systematically eliminate the non-objects to arrive at the final set of objects.

Identify two goals of interaction modeling.

The primary goal of interaction modeling are the following**:**

- To allocate the responsibility of a use case realization among the boundary, entity, and controller objects. The responsibilities for each class is reflected as an operation to be supported by that class.

- To show the detailed interaction that occur over time among the objects associated with each use case.

Explain the CRC cards technique.

The interactions diagrams for only simple use cases that involve collaboration among a limited number of classes can be drawn from an inspection of the use case description. More complex use cases require the use of CRC cards where a number of team members participate to determine the responsibility of the classes involved in the use case realization.

CRC (Class-Responsibility-Collaborator) technology was pioneered by Ward Cunningham and Kent Becka at the research laboratory of Tektronix at Portland, Oregon, USA. CRC cards are index cards that are prepared one per each class. On each of these cards, the responsibility of each class is written briefly. The objects with which this object needs to collaborate its responsibility are also written.

CRC cards are usually developed in small group sessions where people role play being various classes. Each person holds the CRC card of the classes he is playing the role of. The cards are deliberately made small (4 inch ´ 6 inch) so that each class can have only limited number of responsibilities. A responsibility is the high level description of the part that a class needs to play in the realization of a use case. An example CRC card for the BookRegister class of the Library Automation System is shown in fig. 8.3.

After assigning the responsibility to classes using CRC cards, it is easier to develop the interaction diagrams by flipping through the CRC cards.

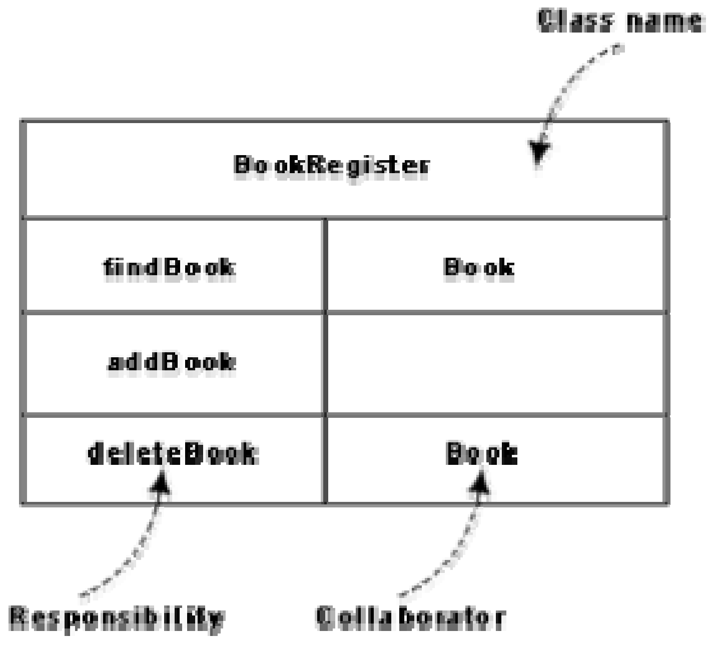
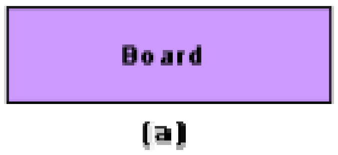


Fig. 8.3: CRC card for the BookRegister class

Develop sequence diagram for any given use case.

Consider the Tic-tac-toe computer game discussed earlier. The step-by-step workout of this example is as follows:

- The use case model is shown in fig. 7.2.
- The initial domain model is shown in fig. 8.4(a).
- The domain model after adding the boundary and control classes is shown in fig. 8.4(b).
- Sequence diagram for the play move use case is shown in fig. 8.5.



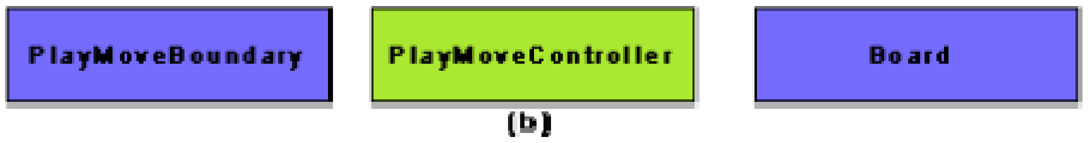


Fig. 8.4: (a) Initial domain model (b) Refined domain model

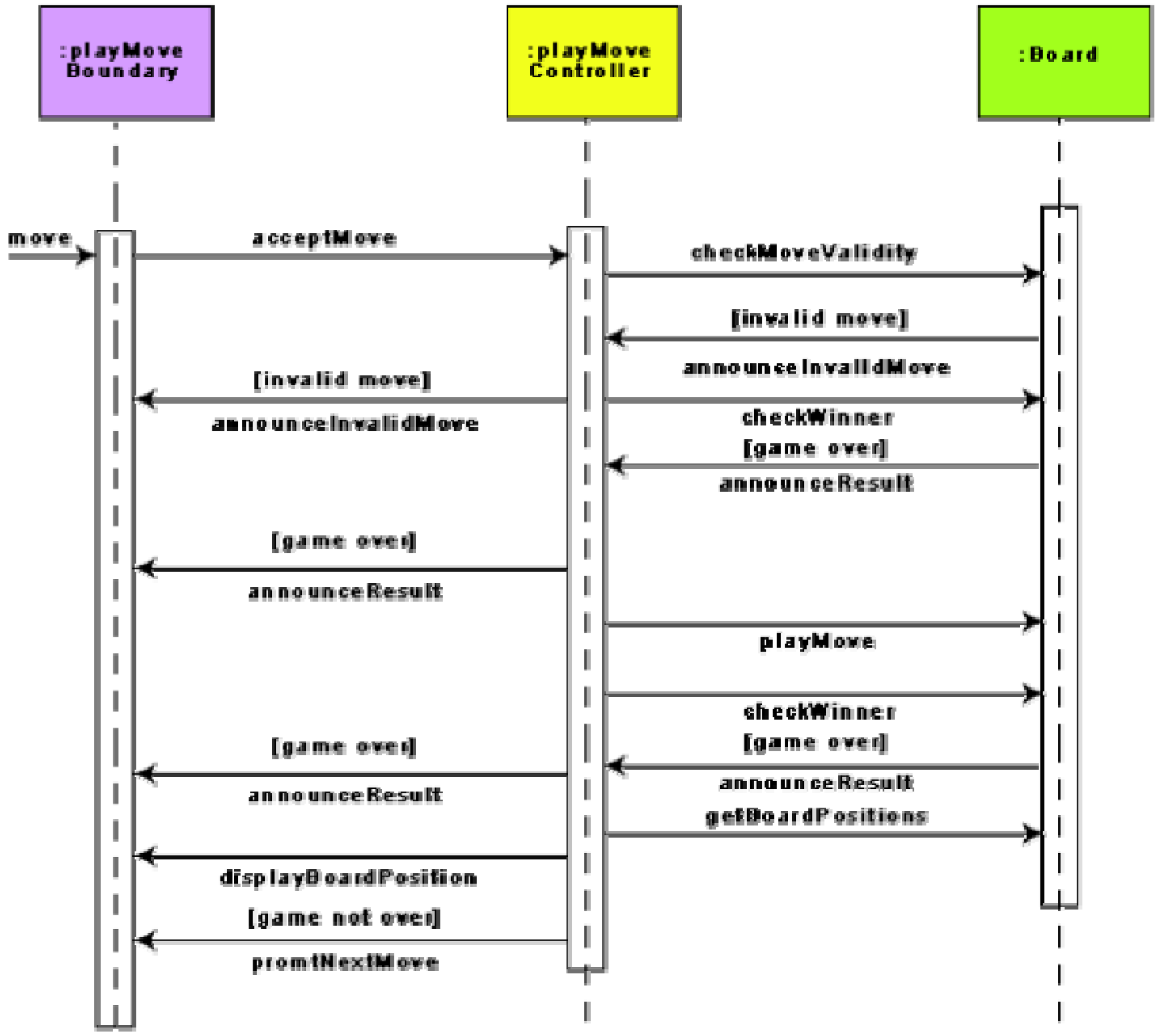


Fig. 8.5: Sequence diagram for the play move use case

Identify how sequence diagrams are useful in developing the class diagram.

Consider the Supermarket prizes scheme software discussed earlier. The step-by-step analysis and design workout of this problem is as follows:

- The use case model is shown in fig. 8.6.
- The initial domain model is shown in fig. 8.7(a).
- The domain model after adding the boundary and control classes is shown in fig. 8.7(b).

- Sequence diagram for the select winner list use case is shown in fig. 8.8.
- Sequence diagram for the register customer use case is shown in fig. 8.9.
- Sequence diagram for the register sales use case is shown in fig. 8.10. In this use case, since the responsibility of the RegisterSalesController is trivial, the controller class can be deleted and the sequence diagram can be redrawn as in fig. 8.11 after incorporating this change.
- Class diagram is shown in fig. 8.12. The messages of the sequence diagrams of the different use cases have been populated as the methods of the corresponding classes.

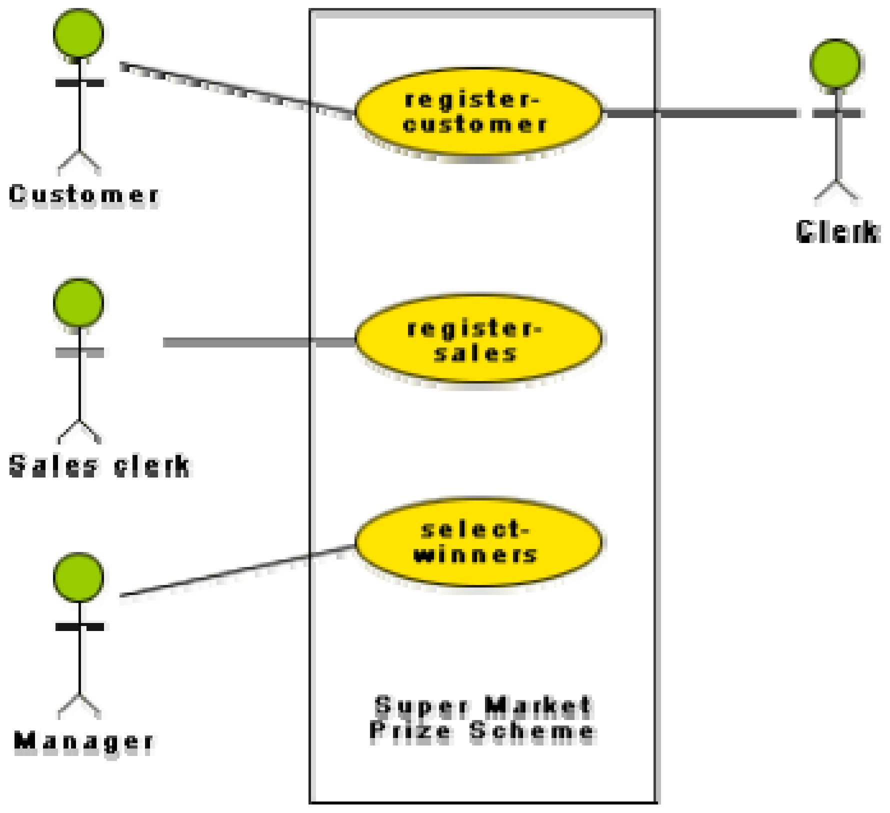
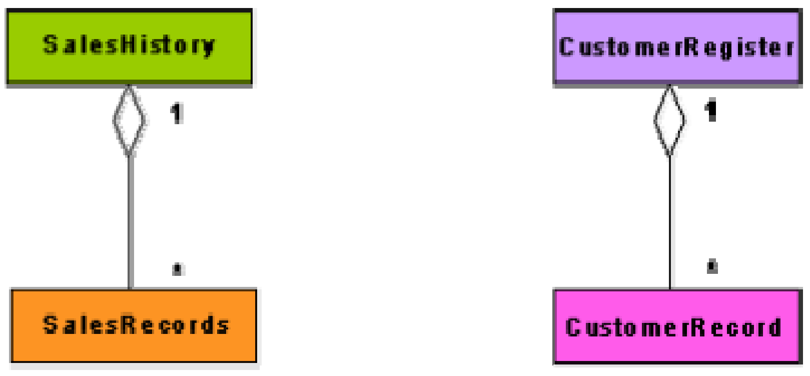
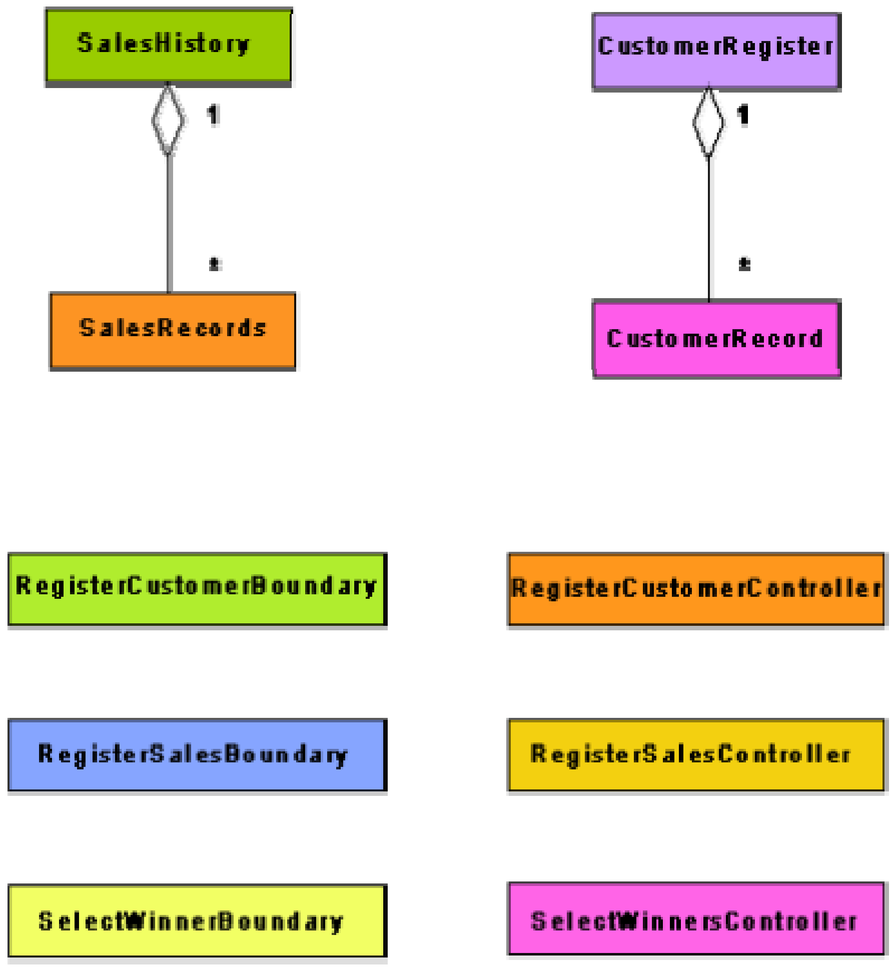


Fig. 8.6: Use case model for Super Market Prize Scheme



(a)



(b)

Fig. 8.7: (a) Initial domain model (b) Refined domain model

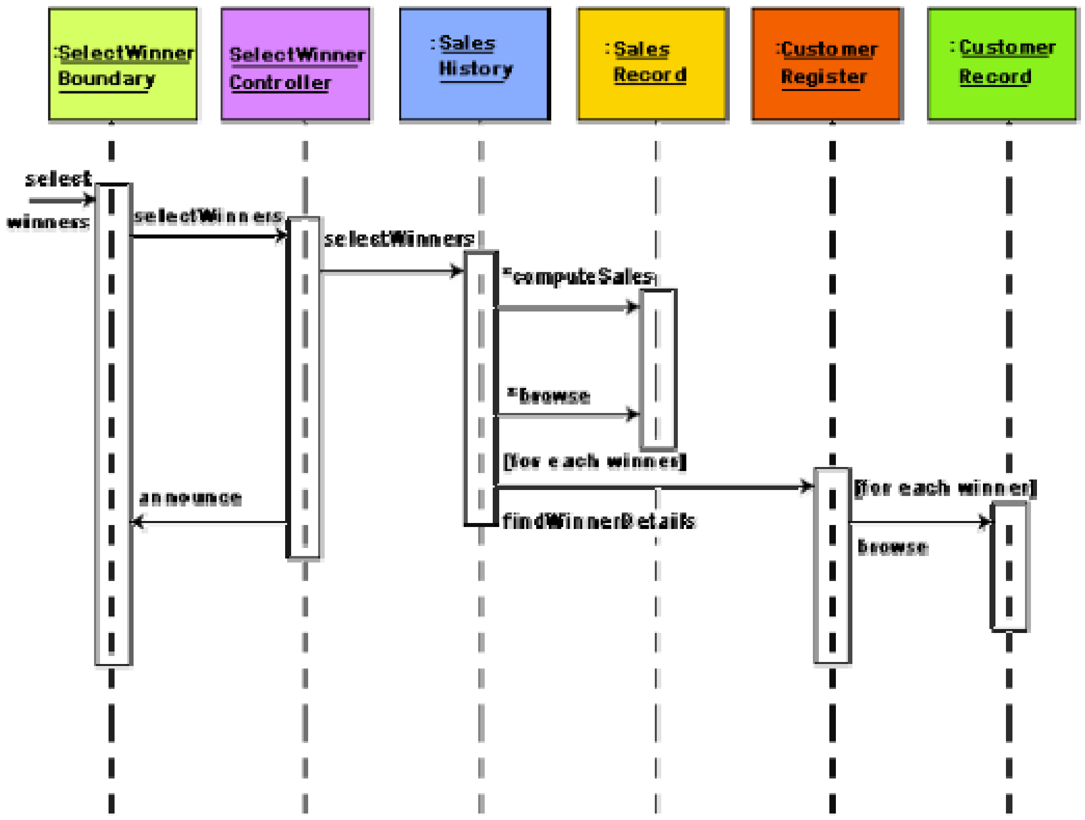


Fig. 8.8: Sequence diagram for the select winner list use case

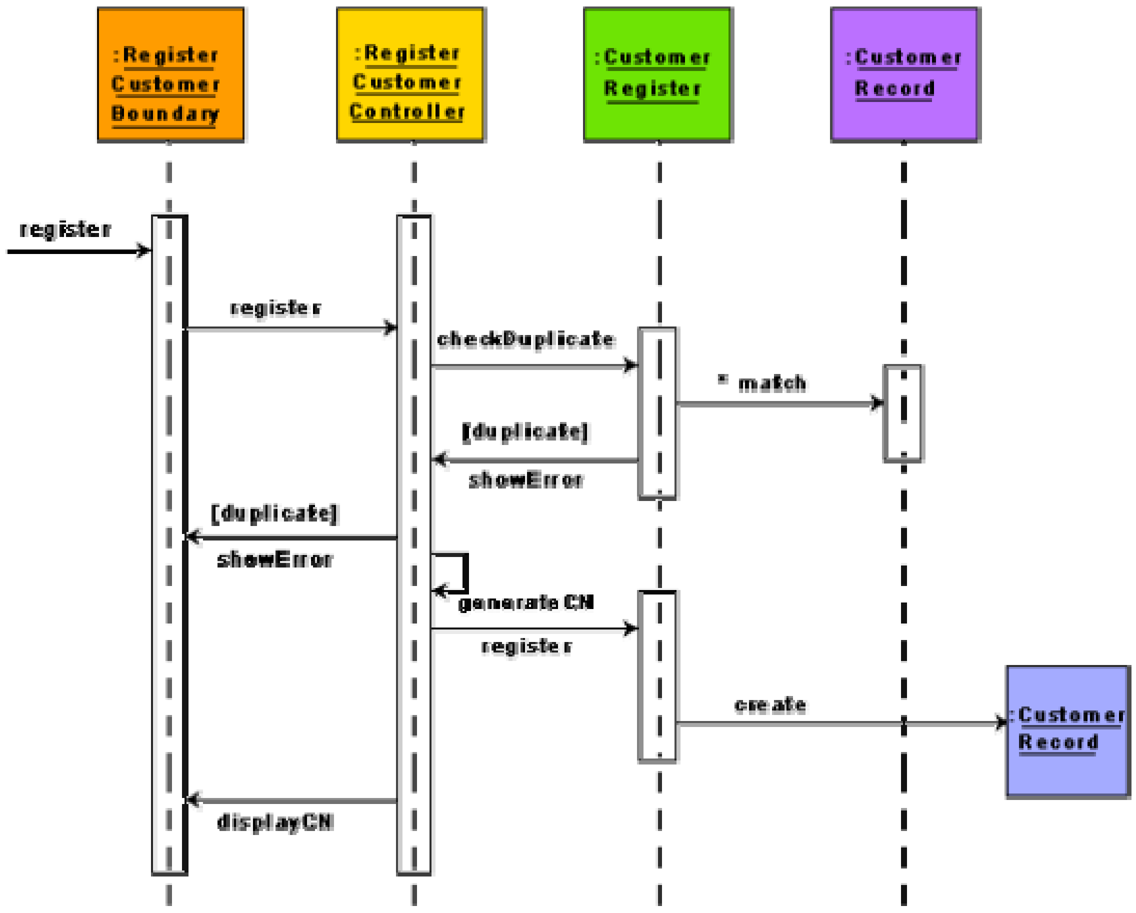


Fig. 8.9: Sequence diagram for the register customer use case

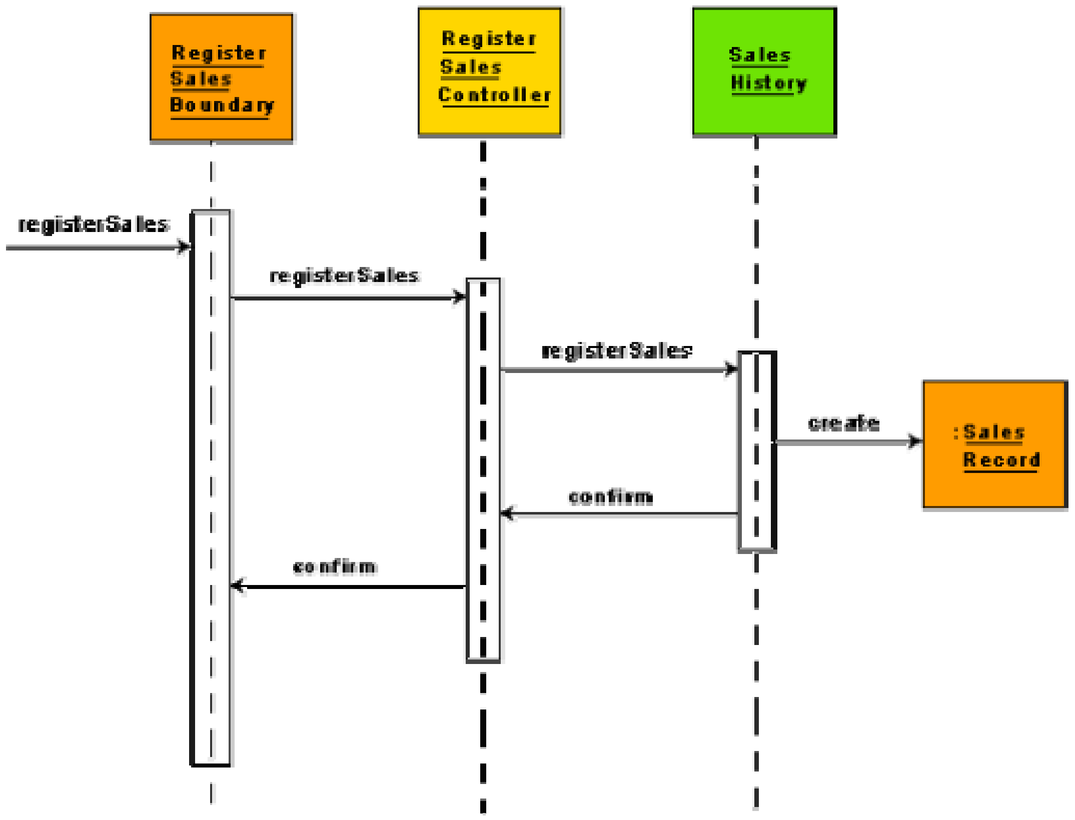


Fig. 8.10: Sequence diagram for the register sales use case

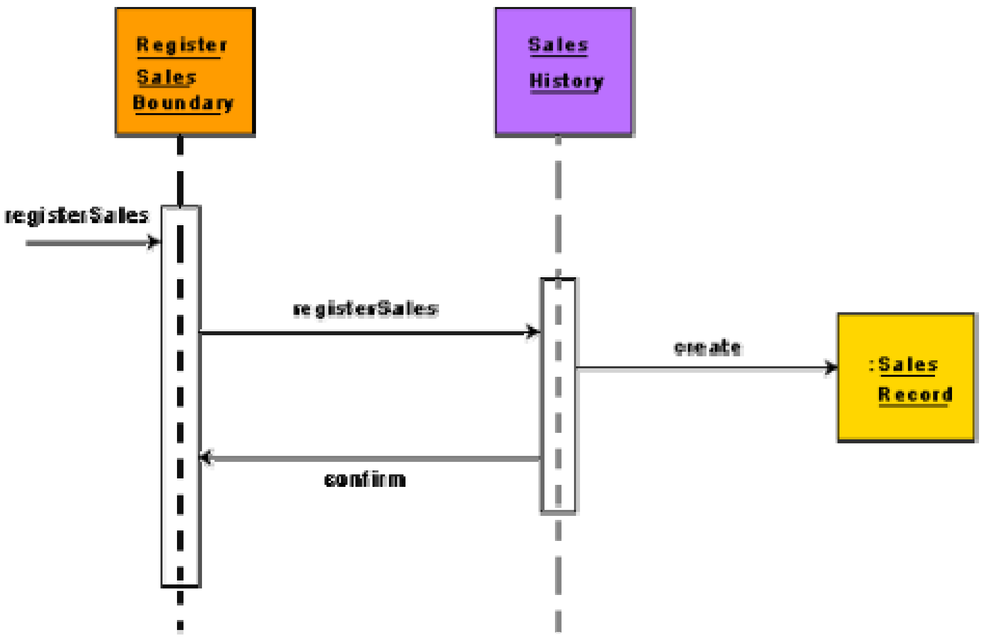


Fig. 8.11: Refined sequence diagram for the register sales use case

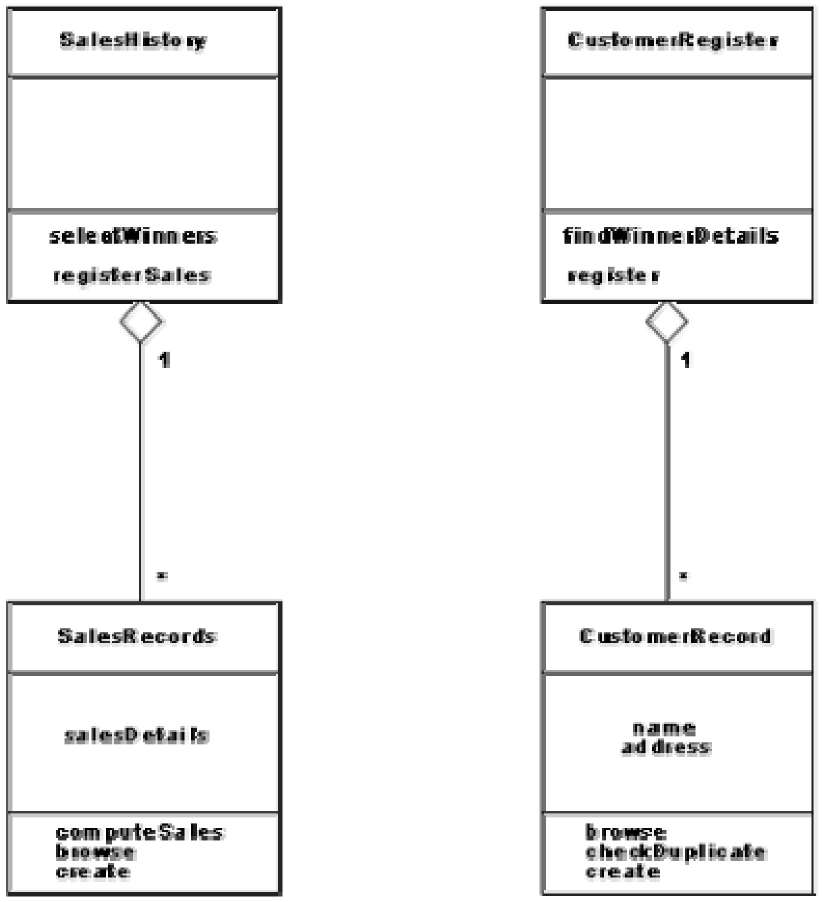


Fig. 8.12: Class diagram

Identify five important criteria for judging the goodness of an object-oriented design.

It is quite obvious that there are several subjective judgments involved in arriving at a good object-oriented design. Therefore, several alternative design solutions to the same problem are possible. In order to be able to determine which of any two designs is better, some criteria for judging the goodness of a design must be identified. The following are some of the accepted criteria for judging the goodness of a design.

- **Coupling guidelines.** The number of messages between two objects or among a group of objects should be minimum. Excessive coupling between objects is determined to modular design and prevents reuse.

- **Cohesion guideline.** In OOD, cohesion is about three levels:

  # **Cohesiveness of the individual methods.** Cohesiveness of each of

the individual method is desirable, since it assumes that each method does only a well-defined function.

# **Cohesiveness of the data and methods within a class.** This is desirable since it assures that the methods of an object do actions for which the object is naturally responsible, i.e. it assures that no action has been improperly mapped to an object.

# **Cohesiveness of an entire class hierarchy.** Cohesiveness of methods within a class is desirable since it promotes encapsulation of the objects.

- **Hierarchy and factoring guidelines.** A base class should not have too many subclasses. If too many subclasses are derived from a single base class, then it becomes difficult to understand the design. In fact, there should approximately be no more than 7±2 classes derived from a base class at any level.

- **Keeping message protocols simple.** Complex message protocols are an indication of excessive coupling among objects. If a message requires more than 3 parameters, then it is an indication of bad design.

- **Number of Methods.** Objects with a large number of methods are likely to be more application-specific and also difficult to comprehend – limiting the possibility of their reuse. Therefore, objects should not have too many methods. This is a measure of the complexity of a class. It is likely that the classes having more than about seven methods would have problems.

- **Depth of the inheritance tree.** The deeper a class is in the class inheritance hierarchy, the greater is the number of methods it is likely to inherit, making it more complex. Therefore, the height of the inheritance tree should not be very large.

- **Number of messages per use case.** If methods of a large number of objects are invoked in a chain action in response to a single message, testing and debugging of the objects becomes complicated. Therefore, a single message should not result in excessive message generation and transmission in a system.

- **Response for a class.** This is a measure of the maximum number of methods that an instance of this class would call. If the same method is called more than once, then it is counted only once. A class which calls more than about seven different methods is susceptible to errors.

The following questions have been designed to test the objectives identified for this module:

1. Write down basic differences between object-oriented analysis (OOA) and object-oriented design (OOD) technique.

Ans.: - The term object-oriented analysis (OOA) refers to a method of developing an initial model of the software from the requirements specification. The analysis model is refined into a design model. The design model can be implemented using a programming language. The term object-oriented programming refers to the implementation of programs using object-oriented concepts.

In contrast, object-oriented design (OOD) paradigm suggests that the natural objects (i.e. the entities) occurring in a problem should be identified first and then implemented. Object-oriented design (OOD) techniques not only identify objects but also identify the internal details of these identified objects. Also, the relationships existing among different objects are identified and represented in such a way that the objects can be easily implemented using a programming language.

2. What is meant by design patterns?

Ans.: - Design patterns are very useful in creating good software design solutions. In addition to providing the model of a good solution, design patterns include a clear specification of the problem, and also explain the circumstances in which the solution would and would not work. Thus, a design pattern has four important parts:

- The problem.
- The context in which the problem occurs.
- The solution.
- The context within which the solution works.

3. What are the advantages of using design patterns?

Ans.: - Design patterns are reusable solutions to problems that recur in many applications. A pattern serves as a guide for creating a “good” design. Patterns are based on sound common sense and the application of fundamental design principles. These are created by people who spot repeating themes across designs. The pattern solutions are typically described in terms of class and interaction diagrams. Examples of design patterns are expert pattern, creator pattern, controller pattern etc.

4. Write down some popular design patterns and their necessities.

Ans.: - Some popular design patterns are as follows:

Expert Pattern:

Problem: Which class should be responsible for doing certain things?

Solution: Assign responsibility to the information expert – the class that has the information necessary to fulfill the required responsibility. The expert pattern expresses the common intuition that objects do things related to the information they have. The class diagram and collaboration diagrams for this solution to the problem of which class should compute the total sales is shown in the fig. 8.1.

Creator Pattern:

Problem: Which class should be responsible for creating a new instance of some class?

Solution: Assign a class C1 the responsibility to create an instance of class C2, if one or more of the following are true**:**

- C1 is an aggregation of objects of type C2.
- C1 contains objects of type C2.
- C1 closely uses objects of type C2.
- C1 has the data that would be required to initialize the objects of type C2, when they are created.

Controller Pattern:

Problem: Who should be responsible for handling the actor requests?

Solution: For every use case, there should be a separate controller object which would be responsible for handling requests from the actor. Also, the same controller should be used for all the actor requests pertaining to one use case so that it becomes possible to maintain the necessary information about the state of the use case. The state information maintained by a controller can be used to identify the out-of-sequence actor requests, e.g. whether voucher request is received before arrange payment request.

Façade Pattern:

Problem: How should the services be requested from a service package?

Context in which the problem occurs: A package as already discussed is a cohesive set of classes – the classes have strongly related responsibilities. For example, an RDBMS interface package may contain classes that allow one to perform various operations on the RDBMS.

Solution: A class (such as DBfacade) can be created which provides a common interface to the services of the package.

5. Outline an object-oriented development process.

Ans.: - A generalized object-oriented analysis and design process is schematically shown in the following figure.

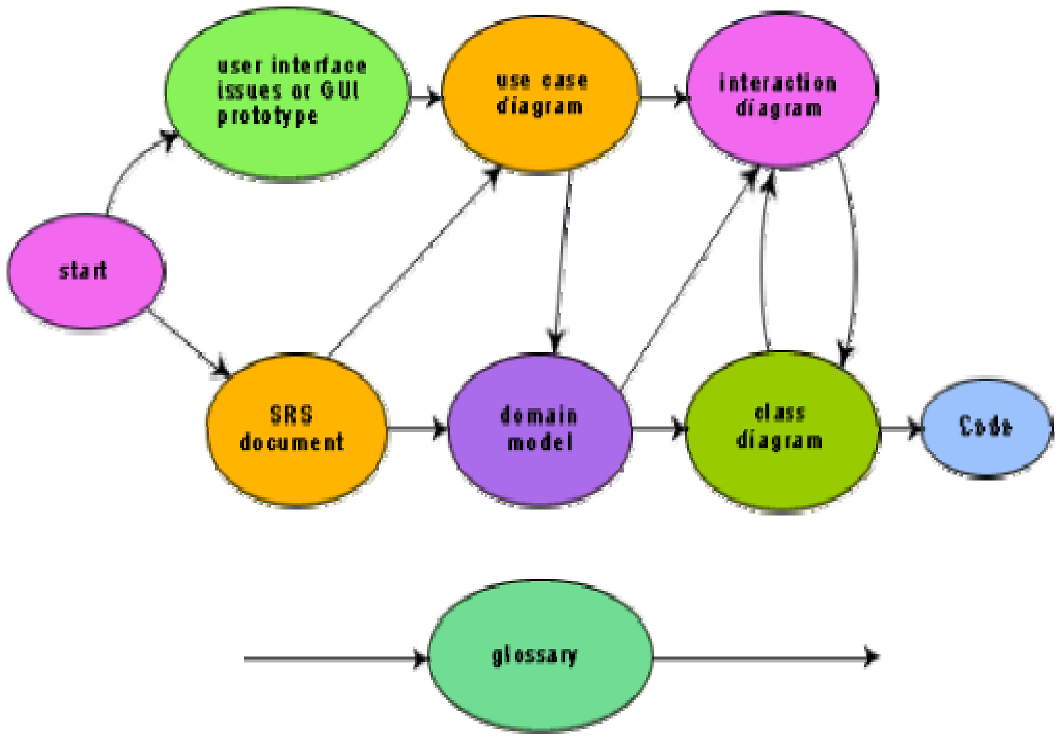


Fig. 8.13: An object-oriented analysis and design process

The use case model is developed first. In any user-centric development process, all models must conform to the use case model. As shown in fig. 8.13, the domain model is constructed next by examining the use case model and the SRS document. The domain model is refined into the class diagram through a serious of iterations through the interaction diagram.

Through out the analysis and design process, a glossary is continuously and consciously prepared. A glossary is a dictionary of terms which can help in understanding the various terms (or concepts) used in the model. The terms listed in the glossary are essentially concept names. The glossary or model dictionary lists and defines all the terms that require explanation in order to improve communication and to reduce the risk of misunderstanding. Maintaining the glossary is an ongoing activity through out the project as shown in the fig. 8.13.

6. What is meant by domain modeling?

Ans.: - Domain modeling is known as conceptual modeling. A domain model is a representation of the concepts or objects appearing in the problem domain. It also captures the obvious relationships among these objects. Examples of such conceptual objects are the Book, BookRegister, MemeberRegister, LibraryMember, etc. The recommended strategy is to quickly create a rough conceptual model where the emphasis is in finding the obvious concepts expressed in the requirements while deferring a detailed investigation. Later during the development process, the conceptual model is incrementally refined and extended.

7. Differentiate among different types of objects that are identified during domain analysis and also explain the relationships among those identified objects.

Ans.: - The different kinds of objects identified during domain analysis and their relationships are as follows:

Boundary objects: The boundary objects are those with which the actors interact. These include screens, menus, forms, dialogs, etc. The boundary objects are mainly responsible for user interaction. Therefore, they normally do not include any processing logic. However, they may be responsible for validating inputs, formatting, outputs, etc. The boundary objects were earlier being called as the interface objects. However, the term interface class is being used for Java, COM/DCOM, and UML with different meaning. A recommendation for the initial identification of the boundary classes is to define one boundary class per actor/use case pair.

Entity objects: These normally hold information such as data tables and files that need to outlive use case execution, e.g. Book, BookRegister, LibraryMember, etc. Many of the entity objects are “dumb servers”. They are normally responsible for storing data, fetching data, and doing some fundamental kinds of operation that do not change often.

Controller objects: The controller objects coordinate the activities of a set of entity objects and interface with the boundary objects to provide the overall behavior of the system. The responsibilities assigned to a controller object are closely related to the realization of a specific use case. The controller objects effectively decouple the boundary and entity objects from one another making the system tolerant to changes of the user interface and processing logic. The controller objects embody most of the logic involved with the use case realization (this logic may change time to time). A typical interaction of a controller object with boundary and entity objects is shown in fig. 8.2. Normally, each use case is realized using one controller object. However, some use cases can be realized without using any controller object, i.e. through boundary and entity objects only. This is often true for use cases that achieve only some simple manipulation of the stored information.

For example, let's consider the "query book availability" use case of the Library Information System (LIS). Realization of the use case involves only matching the given book name against the books available in the catalog. More complex use cases may require more than one controller object to realize the use case. A complex use case can have several controller objects such as transaction manager, resource coordinator, and error handler. There is another situation where a use case can have more than one controller object. Sometimes the use cases require the controller object to transit through a number of states. In such cases, one controller object might have to be created for each execution of the use case.

8. Explain at least three approaches for identifying objects in the context of object-oriented design methodology.

Ans.: - One of the most important steps in any object-oriented design methodology is the identification of objects. In fact, the quality of the final design depends to a great extent on the appropriateness of the objects identified. However, to date no formal methodology exists for identification of objects. Several semi-formal and informal approaches have been proposed for object identification. These can be classified into the following broad classes:

- Grammatical analysis of the problem description.
- Derivation from data flow.
- Derivation from the entity relationship (E-R) diagram.

A widely accepted object identification approach is the grammatical analysis approach. Grady Booch originated the grammatical analysis approach [1991]. In Booch's approach, the nouns occurring in the extended problem description statement (processing narrative) are mapped to objects and the verbs are mapped to methods. The identification approaches based on derivation from the data flow diagram and the entity-relationship model are still evolving and therefore will not be discussed in this text.

Booch's Object Identification Method

Booch's object identification approach requires a processing narrative of the given problem to be first developed. The processing narrative describes the problem and discusses how it can be solved. The objects are identified by noting down the nouns in the processing narrative. Synonym of a noun must be eliminated. If an object is required to implement a solution, then it is said to be part of the solution space. Otherwise, if an object is necessary only to describe the problem, then it is said to be a part of the problem space. However, several of the nouns may not be objects. An imperative procedure name, i.e., noun form of a verb actually represents an action and should not be considered as an object. A potential object found after lexical analysis is usually considered legitimate, only if it satisfies the following criteria**:**

Retained information: Some information about the object should be remembered for the system to function. If an object does not contain any private data, it can not be expected to play any important role in the system.

Multiple attributes: Usually objects have multiple attributes and support multiple methods. It is very rare to find useful objects which store only a single data element or support only a single method, because an object having only a single data element or method is usually implemented as a part of another object.

Common operations: A set of operations can be defined for potential objects. If these operations apply to all occurrences of the object, then a class can be defined. An attribute or operation defined for a class must apply to each instance of the class. If some of the attributes or operations apply only to some specific instances of the class, then one or more subclasses can be needed for these special objects.

Normally, the actors themselves and the interactions among themselves should be excluded from the entity identification exercise. However, some times there is a need to maintain information about an actor within the system. This is not the same as modeling the actor. These classes are sometimes called surrogates. For example, in the Library Information System (LIS) we would need to store information about each library member. This is independent of the fact that the library member also plays the role of an actor of the system.

Although the grammatical approach is simple and intuitively appealing, yet through a naive use of the approach, it is very difficult to achieve high quality results. In particular, it is very difficult to come up with useful abstractions simply by doing grammatical analysis of the problem description. Useful abstractions usually result from clever factoring of the problem description into independent and intuitively correct elements.

Example: Tic-tac-toe

Let us identify the entity objects of the following Tic-tac-toe software:

Tic-tac-toe is a computer game in which a human player and the computer **make** alternative moves on a 3 X 3 square. A move consists of **marking** a previously unmarked square. A player who first **places** three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square **wins**. As soon as either the human player or the computer **wins**, a message congratulating the winner should be **displayed**. If neither player manages to **get** three consecutive marks along a straight line, and all the squares on the board are filled up, then the game is **drawn**. The computer always **tries to win** a game.

By performing a grammatical analysis of this problem statement, it can be seen that nouns that have been underlined in the problem description and the actions as the italicized verbs. However, on closer examination synonyms can be eliminated from the identified nouns. The list of nouns after eliminating the synonyms are the following: Tic-tac-toe, computer game, human player, move, square, mark, straight line, board, row, column, and diagonal.

From this list of possible objects, nouns can be eliminated like human player as it does not belong to the problem domain. Also, the nouns square, game, computer, Tic-tac-toe, straight line, row, column, and diagonal can be eliminated, as any data and methods can not be associated with them. The noun **move** can also be eliminated from the list of potential objects since it is an imperative verb and actually represents an action. Thus, there is only one object left – board.

After experienced in object identification, it is not normally necessary to really identify all nouns in the problem description by underlining them or actually listing them down, and systematically eliminate the non-objects to arrive at the final set of objects.

9. Identify the primary goal of interaction modeling in the context of object-oriented design.

Ans.: - The primary goal of interaction modeling are the following**:**

- To allocate the responsibility of a use case realization among the boundary, entity, and controller objects. The responsibilities for each class is reflected as an operation to be supported by that class.

- To show the detailed interaction that occur over time among the objects associated with each use case.

10. Identify the necessity of CRC (Class-Responsibility-Collaborator) cards in the context of object-oriented design.

Ans.: - The interactions diagrams for only simple use cases that involve collaboration among a limited number of classes can be drawn from an inspection of the use case description. More complex use cases require the use of CRC cards where a number of team members participate to determine the responsibility of the classes involved in the use case realization.

CRC (Class-Responsibility-Collaborator) technology was pioneered by Ward Cunningham and Kent Becka at the research laboratory of Tektronix at Portland, Oregon, USA. CRC cards are index cards that are prepared one per each class. On each of these cards, the responsibility of each class is written briefly. The objects with which this object needs to collaborate its responsibility are also written.

CRC cards are usually developed in small group sessions where people role play being various classes. Each person holds the CRC card of the classes he is playing the role of. The cards are deliberately made small (4 inch ´ 6 inch) so that each class can have only limited number of responsibilities. A responsibility is the high level description of the part that a class needs to play in the realization of a use case. An example CRC card for the BookRegister class of the Library Automation System is shown in fig. 8.3.

After assigning the responsibility to classes using CRC cards, it is easier to develop the interaction diagrams by flipping through the CRC cards.

11. Define the term cohesion in the context of object-oriented design.

Ans.: - Cohesion is a measure of functional strength of a module. In OOD, cohesion is about three levels**:**

# **Cohesiveness of the individual methods -** Cohesiveness of each of the individual method is desirable, since it assumes that each method does only a well-defined function.

# **Cohesiveness of the data and methods within a class -**This is desirable since it assures that the methods of an object do actions for which the object is naturally responsible, i.e. it assures that no action has been improperly mapped to an object.

# **Cohesiveness of an entire class hierarchy -** Cohesiveness of

methods within a class is desirable since it promotes encapsulation of the objects.

12. Identify at least five important features that characterize a good object-oriented design.

Ans.: - It is quite obvious that there are several subjective judgments involved in arriving at a good object-oriented design. Therefore, several alternative design solutions to the same problem are possible. In order to be able to determine which of any two designs is better, some criteria for judging the goodness of a design must be identified. The following are some of the accepted criteria for judging the goodness of a design.

- **Coupling guidelines.** The number of messages between two objects or among a group of objects should be minimum. Excessive coupling between objects is determined to modular design and prevents reuse.

- **Cohesion guideline.** In OOD, cohesion is about three levels**:**

  # **Cohesiveness of the individual methods.** Cohesiveness of each of the individual method is desirable, since it assumes that each method does only a well-defined function.

  # **Cohesiveness of the data and methods within a class.** This is desirable since it assures that the methods of an object do actions for which the object is naturally responsible, i.e. it assures that no action has been improperly mapped to an object.

  # **Cohesiveness of an entire class hierarchy.** Cohesiveness of methods within a class is desirable since it promotes encapsulation of the objects.

- **Hierarchy and factoring guidelines.** A base class should not have too many subclasses. If too many subclasses are derived from a single base class, then it becomes difficult to understand the design. In fact, there should approximately be no more than 7±2 classes derived from a base class at any level.

- **Keeping message protocols simple.** Complex message protocols are an indication of excessive coupling among objects. If a message requires more than 3 parameters, then it is an indication of bad design.

- **Number of Methods.** Objects with a large number of methods are likely to be more application-specific and also difficult to comprehend – limiting the possibility of their reuse. Therefore, objects should not have too many methods. This is a measure of the complexity of a class. It is likely that the classes having more than about seven methods would have problems.

- **Depth of the inheritance tree.** The deeper a class is in the class inheritance hierarchy, the greater is the number of methods it is likely to inherit, making it more complex. Therefore, the height of the inheritance tree should not be very large.

- **Number of messages per use case.** If methods of a large number of objects are invoked in a chain action in response to a single message, testing and debugging of the objects becomes complicated. Therefore, a single message should not result in excessive message generation and transmission in a system.

- **Response for a class.** This is a measure of the maximum number of methods that an instance of this class would call. If the same method is called more than once, then it is counted only once. A class which calls more than about seven different methods is susceptible to errors.

Mark all options which are true.

1. The design pattern solutions are typically described in terms of
   □ class diagrams
   □ object diagrams
   □ interaction diagrams
   □ both class and interaction diagrams √

2. The class that should be responsible for doing certain things for which it has the necessary information – is the solution proposed by
   □ creator pattern
   □ controller pattern
   □ expert pattern √
   □ façade pattern

3. The class that should be responsible for creating a new instance of some class – is the solution proposed by
   □ creator pattern √
   □ controller pattern
   □ expert pattern
   □ façade pattern

4. The class that should be responsible for handling the actor requests – is the solution proposed by
   □ creator pattern

□ controller pattern √
□ expert pattern
□ façade pattern

5. The objects identified during domain analysis can be classified into
□ boundary objects
□ controller objects
□ entity objects
□ all of the above √

6. The objects with which the actors interact are known as
□ controller objects
□ boundary objects √
□ entity objects
□ all of the above

7. The most critical part of the domain modeling activity is to identify
□ controller objects
□ boundary objects
□ entity objects √
□ none of the above

8. The objects which are responsible for storing data, fetching data, and doing some fundamental kinds of operation that do not change often are known as
□ controller objects
□ boundary objects
□ entity objects √
□ none of the above

9. The objects which effectively decouple the boundary and entity objects from one another making the system tolerant to changes of the user interface and processing logic are
□ controller objects √
□ boundary objects
□ entity objects
□ none of the above

Mark the following as either True or False. Justify your answer.

1. Façade pattern tells the way that non-GUI classes should communicate with the GUI classes.

Ans.: - False.

Explanation: - A façade pattern tells how should the services be requested from a service package? On the other hand, model view separation model tells the way that non-GUI classes should communicate with the GUI classes.

2. **The use cases should be tightly tied to the GUI.**

   Ans.: - False.

   Explanation: - The use cases should not be too tightly tied to the GUI. For example, the use cases should not make any reference to the type of the GUI element appearing on the screen, e.g. radioButton, pushButton, etc. This is necessary because, the type of the user interface component used may change frequently. However, the functionalities do not change so often.

3. **The responsibilities assigned to a controller object are closely related to the realization of a specific use case.**

   Ans.: - True.

   Explanation: - Normally, each use case is realized using one controller object. More complex use cases may require more than one controller object to realize the use case. A complex use case can have several controller objects such as transaction manager, resource coordinator, and error handler.

4. **Entity objects are responsible for implementing the business logic.**

   Ans.: - False.

   Explanation: - Entity objects normally hold information such as data tables and files that need to outlive use case execution, e.g. Book, BookRegister, LibraryMember, etc. Many of the entity objects are "dumb servers". They are normally responsible for storing data, fetching data, and doing some fundamental kinds of operation that do not change often. The controller objects are responsible for implementing the business logic.

5. **CRC card technique is useful in identifying the different classes necessary to solve a problem.**

   Ans.: - False.

Explanation: - CRC (Class-Responsibility-Collaborator) cards are index cards that are prepared one per each class. On each of these cards, the responsibility of each class is written briefly. The objects with which this object needs to collaborate its responsibility are also written. Once we assign the responsibility to classes using CRC cards, then we can develop the interaction diagrams by flipping through the CRC cards. The CRC cards help determining the methods to be supported by different classes and the interaction among the classes.

6. **There is a one-to-one correspondence between the classes of the domain model and the final class diagram.**

   Ans.: - False.

   Explanation: - The domain model undergoes refinements such as combining classes if the roles of some classes are trivial and splitting classes which have too much of responsibility and even adding new classes when required, etc.

7. **Large number of message exchanges between objects indicates good delegation and is a sure sign of a design well-done.**

   Ans.: - False.

   Explanation: - The number of messages between two objects or among a group of objects should be kept to the minimum. Excessive coupling between objects is detrimental to modular design and prevents reuse. It also makes testing of the classes difficult.

8. **Deep class hierarchies are the hallmark of any good OOD.**

   Ans.: - False.

   Explanation: - The deeper a class is in the class inheritance hierarchy, the greater is the number of methods it is likely to inherit, making the design more complex. Therefore, the height of the inheritance tree should not be very large.

9. **Cohesiveness of the data and methods within a class is a sign of good OOD.**

   Ans.: - True.

   Explanation: - Cohesiveness of the data and the methods within a class is desirable since it assures that the methods of an object do actions for which the object is naturally responsible, i.e. it assures that no action has been improperly mapped to an object.

10. **Keeping the message protocols complex is an indication of a good object-oriented design.**

    Ans.: - False.

    Explanation: - Complex message protocols are an indication of excessive coupling among objects. We also know that excessive coupling among objects are not desirable for a good OOD. If a message requires more than 3 parameters, then it is an indication of bad design.