



Unit 3

Software Design



Software Design

- The design activity begins with the requirements document for the software to be developed is available and the architecture has been designed.
- During design we further refine the architecture.
- During design we determine what modules should the system have and which have to be developed.
- The design of a system is a blueprint or a plan for a solution for the system.
- Here we consider a system to be a set of modules with clearly defined behavior which interact with each other in a defined manner to produce some behavior or services for its environment.

Software Design

- The design process for software systems often has two levels. ***system design*** or ***top-level design*** and ***detailed design*** or ***logic design***.
- At first level the focus is on deciding which modules are needed for the system, the specifications of these modules, and how the modules should be interconnected. This is called system design or top-level design.
- In second level, the internal design of the modules, or how the specifications of the module can be satisfied, is decided. This design level is called detailed design or logic design.
- A ***design methodology*** is a systematic approach to creating a design by applying of a set of techniques and guidelines.



Design Principles

- The design of a system is *correct* if a system built precisely according to the design satisfies the requirements of that system.
- But, correctness is not the sole criterion during the design phase, as there can be many correct designs.
- The goal of the design process is to find the best possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate.
- A design should clearly be verifiable, complete and traceable.

Design Principles

- The two most important properties that concern designers are ***efficiency*** and ***simplicity***.
- Efficiency of any system is concerned with the proper use of resources by the system.
- Simplicity is the most important quality criteria for software system. The design of a system is one of the most important factors affecting the maintainability of a system.
- These two criteria are not independent, an increasing one may have an unfavorable effect on another. (tricks used to increase efficiency of a system result in making the system more complex.)
- Creating a simple and efficient design of a large system is extremely complex task.



Design Principles

- The principles that can be used in design are the same as those used in problem analysis, but there are some fundamental differences.
- In problem analysis, we are constructing a model of the problem domain, while in design we are constructing a model for the solution domain.
- In problem analysis, the analyst has limited degrees of freedom in selecting the models as the problem is given, and modeling has to represent it. In design, the designer has a great deal of freedom in deciding the models.
- In design, the system depends on the model, while in problem analysis the model depends on the system.
- The basic aim of modeling in problem analysis is to understand, while the basic aim of modeling in design is to optimize.



Problem Partitioning and Hierarchy

- For solving larger problems, the basic principle is “divide and conquer”.
- For software design, the goal is to divide the problem into manageably small pieces that can be solved separately.
- The different pieces have to cooperate and communicate to solve the larger problem. This communication adds complexity, which arises due to partitioning and may not have existed in the original problem.
- The designer has to make the judgment about when to stop partitioning.



Problem Partitioning and Hierarchy

- If a piece can be modified separately, we call it independent of other pieces.
- Total independence of modules of one system is not possible, but the design process should support as much independence as possible between modules.
- Dependence between modules in a software system is one of the reasons for high maintenance costs.
- The design produced by using problem partitioning can be represented as hierarchy of components.
- The relationship between the elements in this hierarchy can vary depending on the method used. (The most common is the “whole-part of” relationship. In this, the system consists of some parts, each part consist of subparts, and so on.)



Abstraction

- Abstraction permits a designer to consider a component at an abstract level without worrying about the details of the implementation of the component.
- It describes the external behavior of the component without bothering with the internal details that produce the behavior.
- Abstraction is essential for problem partitioning.
- To decide how a component interacts with other components, the designer has to know, at the very least, the external behavior of other components.



Abstraction

- Abstraction plays an important role in the maintenance phase. It helps to determine how modifying a component affects the system.
- There are two common abstraction mechanisms for software systems: ***functional abstraction*** and ***data abstraction***.
- In functional abstraction, a module is specified by the function it performs.
- Functional abstraction is the basis of partitioning in function-oriented approaches.
- The second unit for abstraction is data abstraction. Data abstraction forms the basis for object-oriented design.



Modularity

- A system is considered ***modular*** if it consists of discrete components so that each component can be implemented separately, and a change to one component has minimal impact on other components.
- Modularity helps in system debugging-isolating the system problem to a component is easier if the system is modular.
- A software system cannot be made modular by simply chopping it into a set of modules. For modularity, each module needs to support a well-defined abstraction and have a clear interface through which it can interact with other modules.
- Modularity is where abstraction and partitioning come together.



Top-Down and Bottom-Up Strategies

- A system is a hierarchy of components. The highest-level component correspond to the total system. To design such a hierarchy there are two possible approaches: top-down and bottom-up.



Top Down Strategy

- A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved.
- Top down design methods often result in some form of *stepwise refinement*.
- Starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly.
- Most design methodologies are based on the top-down approach.



Bottom Up Strategy

- A bottom up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower level components.
- Bottom up methods work with layers of abstraction.
- Starting from the very bottom, operations that provide a layer of abstraction are implemented. The operations of this layer are then used to implement more powerful operations and a still higher layer of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.
- A top down approach is suitable only if the specifications of the system are clearly known and the system development is from scratch.
- If a system is to be built from an existing system, a bottom up approach is more suitable, as it starts from some existing components.



Module-Level Concepts

- A module is a logically separable part of a program. It is a program unit that is discreet and identifiable with respect to compiling and loading.
- In systems using functional abstraction, a module is usually a procedure or function or a collection of these.



Coupling

- Two modules are considered independent if one can function completely without the presence of other.
- However, all the modules in a system cannot be independent of each other, as they must interact so that together they produce the desired external behavior of the system.
- coupling between modules is the strength of interconnections between modules or a measure of interdependence among modules.
- “Highly coupled” modules are joined by strong interconnections, while “loosely coupled” modules have weak interconnections.



Factors affecting coupling

- Coupling increases with the complexity and obscurity of the interface between modules.
- To keep coupling low we would like to minimize the number of interfaces per module and the complexity of each interface.
- Complexity of the interface is another factor affecting coupling. The more complex each interface is, the higher will be the degree of coupling.
- The type of information flow along the interfaces is the third major factor affecting coupling.



Factors affecting coupling

- There are two kinds of information that can flow along an interface : data or control.
- Passing or receiving control information means that the action of the module will depend on this control information.
- Transfer of data information means that a module passes as input some data to another module and gets in return some data as output.
- In general, interfaces with only data communication result in the lowest degree of coupling, followed by interfaces that only transfer control data.
- Coupling is considered highest if the data is hybrid.



Cohesion

- Cohesion is the intra-module concept.
- Cohesion determines how closely the elements of a module are related to each other.
- Cohesion of a module represents how tightly bound the internal elements of the module to one another.
- Usually, the greater the cohesion of each module in the system the lower the coupling between modules is.



Levels of cohesion

- Coincidental
- Logical
- Temporal
- Procedural
- Communicational
- Sequential
- Functional

Levels of cohesion

- Coincidental – coincidental cohesion occurs when there is no meaningful relationship among the elements of a module. If a module is created to save duplicate code by combining some part of code that occurs at many different places, that module is likely to have coincidental cohesion.
- Logical – a module has logical cohesion if there is some logical relationship between the elements of a module, and the elements perform functions that fall in the same logical class. For example, the module that performs all the inputs or all the outputs.
- Temporal – is same as logical cohesion, except that the elements are also related in time and are executed together. For example, modules that perform activities like “clean-up”, “termination”. Temporal cohesion is higher than logical cohesion, because the elements are all executed together.



Levels of cohesion

- Procedural – a procedurally cohesive module contains elements that belong to a common procedural unit. For example, a loop or a sequence of decision statements in a module may be combined to form a separate module. A module with only procedural cohesion may contain only part of a complete function or parts of several functions.
- Communicational – a module with communicational cohesion has elements that are related by a reference to the same input or output data. In this, the elements of the modules are together because they operate on the same input or output data. For example, a module to “print and punch record”.



Levels of cohesion

- Sequential – when the elements are together in a module because the output of one forms the input to another, we get sequential cohesion.
- Functional – in a functionally bound module, all the elements of the module are related to perform a single function.

Determination of the cohesion level

- A useful technique for determining if a module has functional cohesion is to write a sentence that describes fully and accurately, the function or purpose of the module. Then , the following tests can be made.
- 1. if the sentence must be a compound sentence, if it contains a comma, or it has more than one verb, the module is probably performing more than one function, and it probably has sequential or communicational cohesion.
- 2. if the sentence contains words relating to time, like “first”, “when” the module probably has sequential or temporal cohesion.

Determination of the cohesion level

- 3. if the predicate of the sentence does not contain a single specific object following the verb such as “edit all data” the module probably has logical cohesion.
- 4. words like “initialize” and “clean up” imply temporal cohesion.



Structured design methodology

- Structured design methodology (SDM) views every software system as having some inputs that are converted into the desired outputs by the software system.
- The software is viewed as transformation function that transforms the given inputs into the desired outputs, and the central problem of designing software systems is considered to be properly designing this transformation function.
- Factoring is the process of decomposing a module so that the bulk of its work is done by its subordinates.
- A system is said to be completely factored if all the actual processing is accomplished by bottom-level atomic modules and if non-atomic modules largely perform the jobs of control and coordination.



Structured design methodology

- The overall strategy is to identify the input and output streams and the primary transformations that have to be produce the output.
- High level modules are then created to perform these major activities, which are later refined.
- There are four major steps in this strategy:
 1. Restate the problem as a data flow diagram.
 2. Identify the input and output data elements.
 3. First-level factoring.
 4. Factoring of input, output, and transform branches.



Design Heuristics

- Module size is often considered an indication of module complexity. However, the decision to split a module or combine different modules should not be based on size alone.
- Cohesion and coupling of modules should be the primary guiding factors.
- A module should be split into separate modules only if the cohesion of the original module was low, the resulting modules have higher degree of cohesion, and the coupling between modules does not increase.
- Similarly, two or more modules should be combined only if the resulting module has a higher degree of cohesion and the coupling of the resulting module is not greater than the coupling of the sub modules.



Design Heuristics

- Another parameter that can be considered while “fine-tuning” the structure is the fan-in and fan-out of the modules.
- Fan-in of a module is the number of arrows coming in the module, indicating the number of super ordinates of a module.
- Fan-out of a module is the number of arrows going out of that module, indicating the number of subordinates of the module.
- In general the fan-out should not be increased above five or six.



Design Heuristics

- Next important factor that should be considered is the correlation of the scope of effect and scope of control.
- The scope of effect of a decision is the collection of all the modules that contain any processing that is conditional on that decision or whose invocation is dependent on the outcome of the decision.
- The scope of control of a module is the module itself and all its subordinates.
- Ideally, the scope of effect should be limited to the modules that are immediate subordinates.



Verification

- The output of the system design phase, should be verified before proceeding with the activities of the next phase.
- The purpose of design reviews is to ensure that the design satisfies the requirements and is of “good quality.”
- Detecting errors in design is the purpose of the design reviews
- The system design review process is similar to the inspection process.
- Sample checklist are also used for any review. The checklist can be used by each member during private study of the design and during the review meeting.



Object Oriented Design

- An OO model closely represents the problem domain, which makes it easier to produce and understand designs.
- As requirements change, the objects in a system are less immune to these changes, thereby permitting changes more easily.
- OO approaches are believed to be more natural and provide richer structures for thinking and abstraction.
- In OOD, we are identifying the classes that should exist in the software and the relationship between these classes.



OO Analysis and OO Design

- Pure object-oriented development requires that object-oriented techniques be used during the analysis, design, and implementation of the system.
- The fundamental difference between OOA and OOD is that the former models the problem domain, while the latter models the solution to the problem.
- The objects during OOA focus on the problem domain and generally represent some things or concepts in the problem. These objects are sometimes called ***semantic objects***.
- The solution domain, consists of semantic objects as well as other objects

Relationship between OOA and OOD

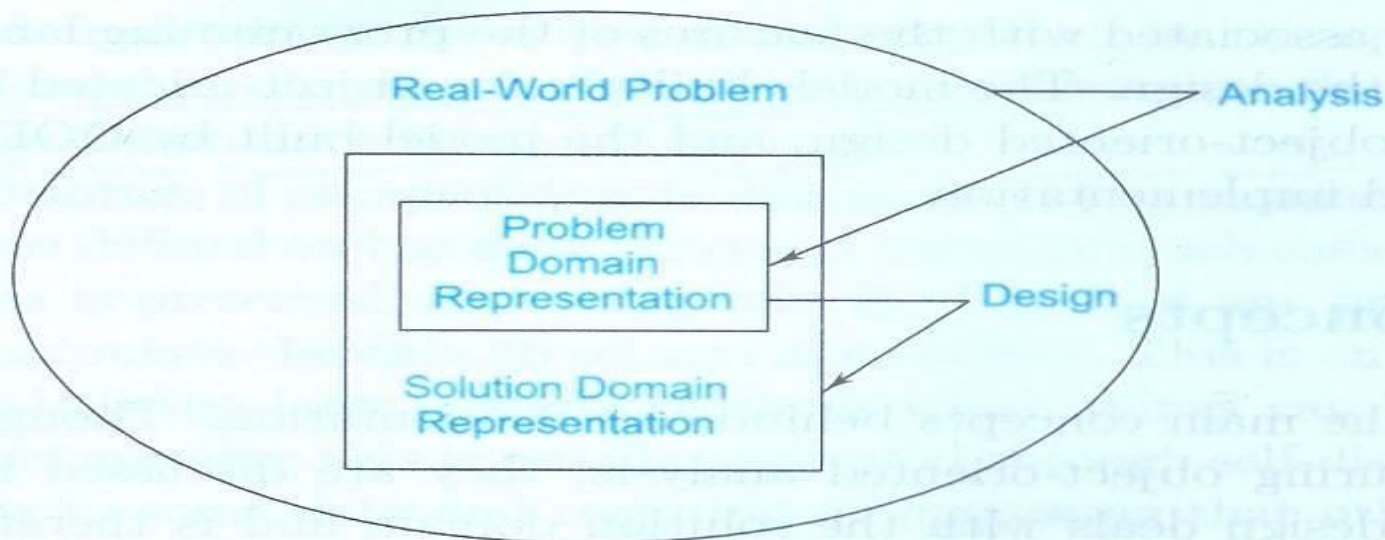


Figure 7.1: Relationship between OOA and OOD.



OO Analysis and OO Design

- During design, as the focus is on finding and defining a solution, the semantic objects identified during OOA may be refined and extended from the point of view of implementation, and other objects are added that are specific to the solution domain.
- The solution domain objects include interface, application and utility objects.
- The interface objects deal with the user interface.
- The application objects specify the control mechanisms for the proposed solution.
- The utility objects are those needed to support the services of the semantic objects or to implement them efficiently.



OO Analysis and OO Design

- The basic goal of the analysis and design activities is to identify the classes in the system and their relationships, and frequently represented by class diagrams.
- In addition to concentrating on the static structure of the problem or solution domains, the dynamic behavior or the system has to be studied to make sure that the final design supports the desired dynamic behaviors.



OO Concepts

- Classes and Objects
- Relationships Among Objects
- Inheritance and Polymorphism



Unified Modeling Language

- UML is a graphical notation for expressing object oriented designs.
- It is called a modeling language and not a design notation as it allows representing various aspects of the system, not just the design that has to be implemented.
- It has become aid for understanding the system, designing the system, as well as notation for representing design.



Use Case Diagram

- Use case diagrams are used:
 - During the analysis phase of a project
 - Part of SRS document
 - To guide design of system
- Provides View of a system that emphasizes the behaviour as it appears to outside users.
- Provides top-down view of system
- Use Case diagram shows a set of use cases, actors and their relationships.

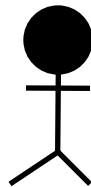
Use Case Diagram

- System boundary defines scope of the system.
- Symbol:



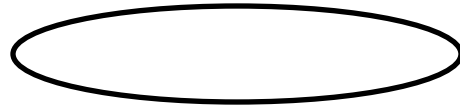
Use Case Diagram

- Actors are roles outside the system derive value from the system
- Entities that interact with the system
 - Provide input
 - Consume data
 - Observe results
 - Provide control information
- Actors are 'nouns' in the problem definition
- Symbol:



Use Case Diagram

- Use Case is a sequence of actions that provide a measurable value to an actor
- Every requirement is a use case
- Every functionality that supports the implementation of a requirement is a use case
- Search for “verbs” in the problem definition.
- Symbol:





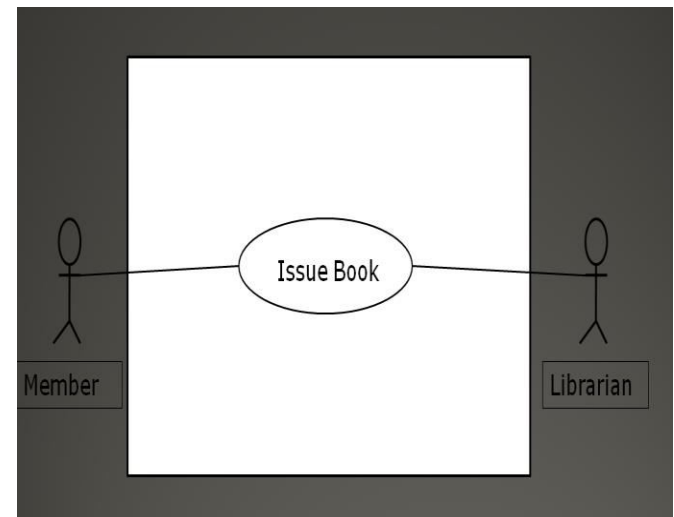
Use Case Diagram

- Mainly four types of relationships are shown in use cases:
 - ☐ Association
 - ☐ Generalization
 - ☐ Includes
 - ☐ Extends

Use Case Diagram


■ Association:

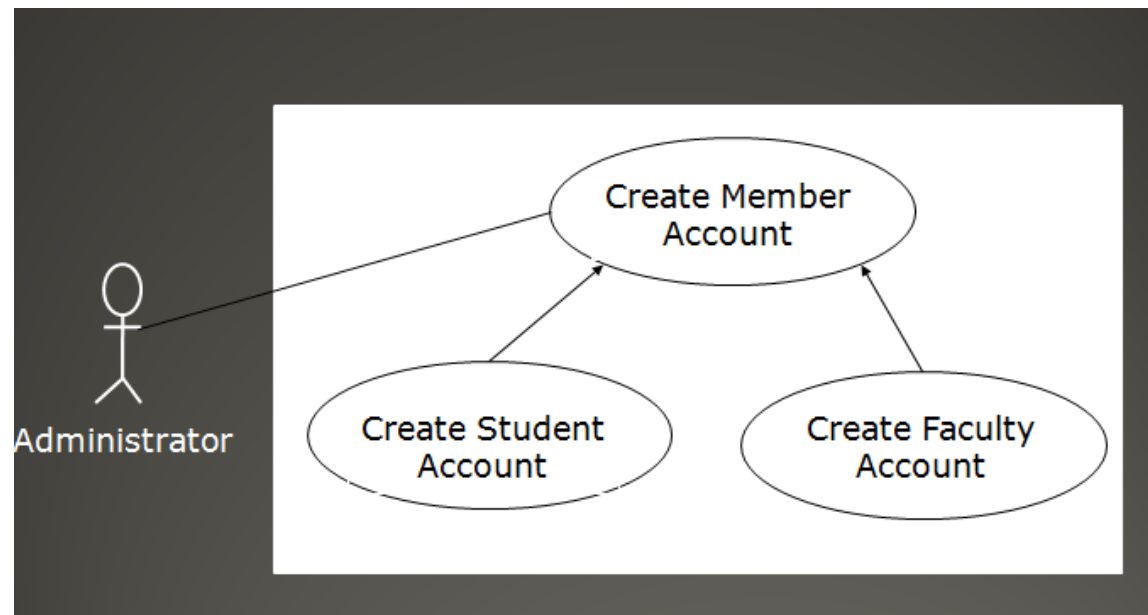
- Relationship between actor and use case
- Represents the fact that actor communicates with use case
- May be directional
- Symbol : _____



Use Case Diagram

■ Generalization:

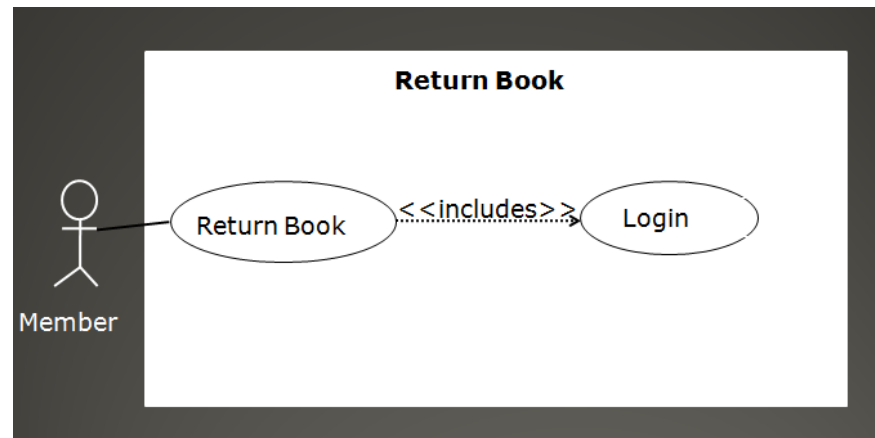
- Shows inheritance between use cases
- Also called “is a” relation
- Symbol : 



Use Case Diagram

Includes:

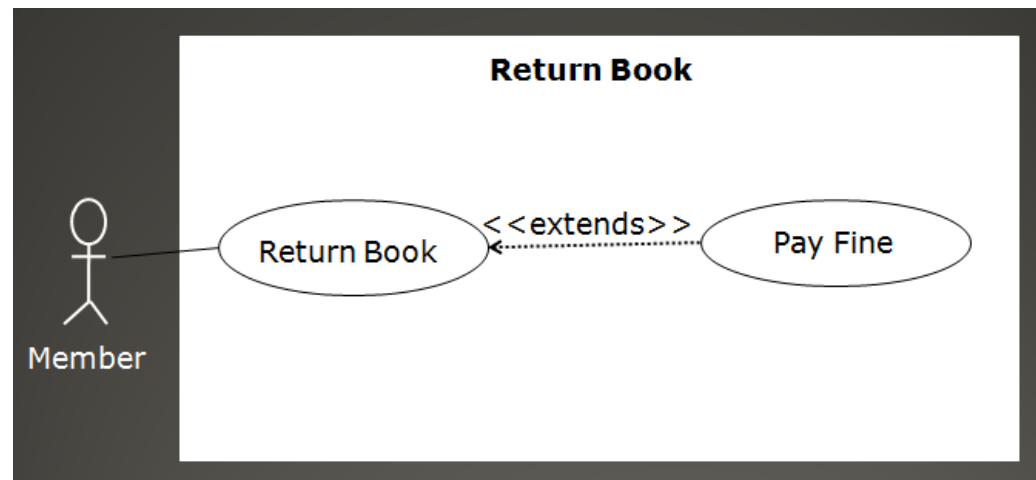
- ☐ Base use case *will always* call included use case
- ☐ Links to general purpose functions, used by many other use cases
- ☐ Shows sub-routine call
- ☐ Symbol : <<includes>>
 →
- ☐ Arrow points from base to included use case



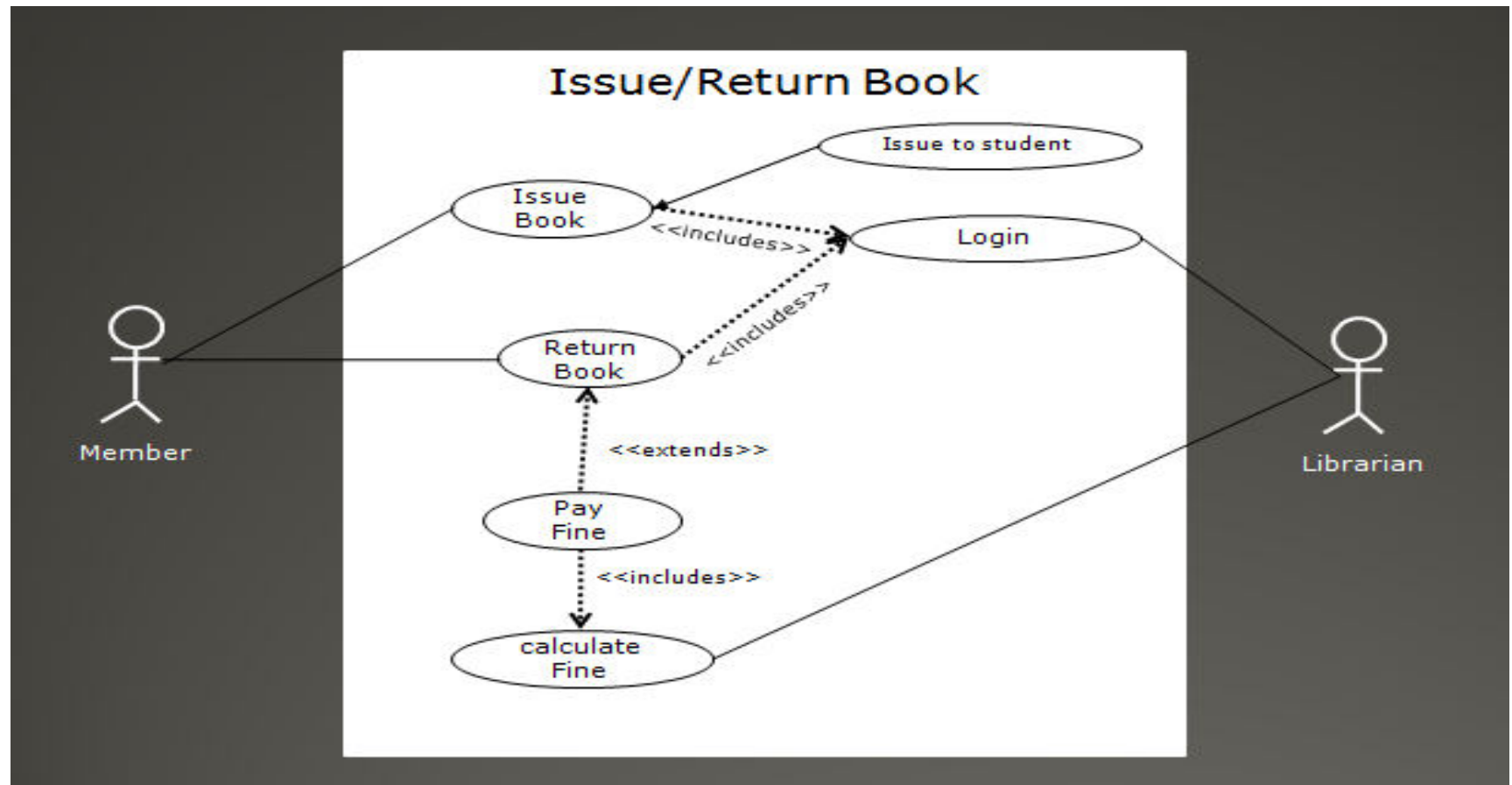
Use Case Diagram

❏ Extends:

- ❑ Base use case *may use* extended use case
- ❑ Extends a use case by adding new behavior
- ❑ Shows alternative flow
- ❑ Symbol :>
- ❑ Arrow points from extended to base use case



Use Case Diagram





Class Diagram

- The class diagram of UML is the central piece in a design or model.
- These diagram describe the classes that are there in the design.
- A class diagram defines:
 - 1 Classes that exist in the system
 - 2 Associations between classes
 - 3 Subtype, super type relationship

Class Diagram

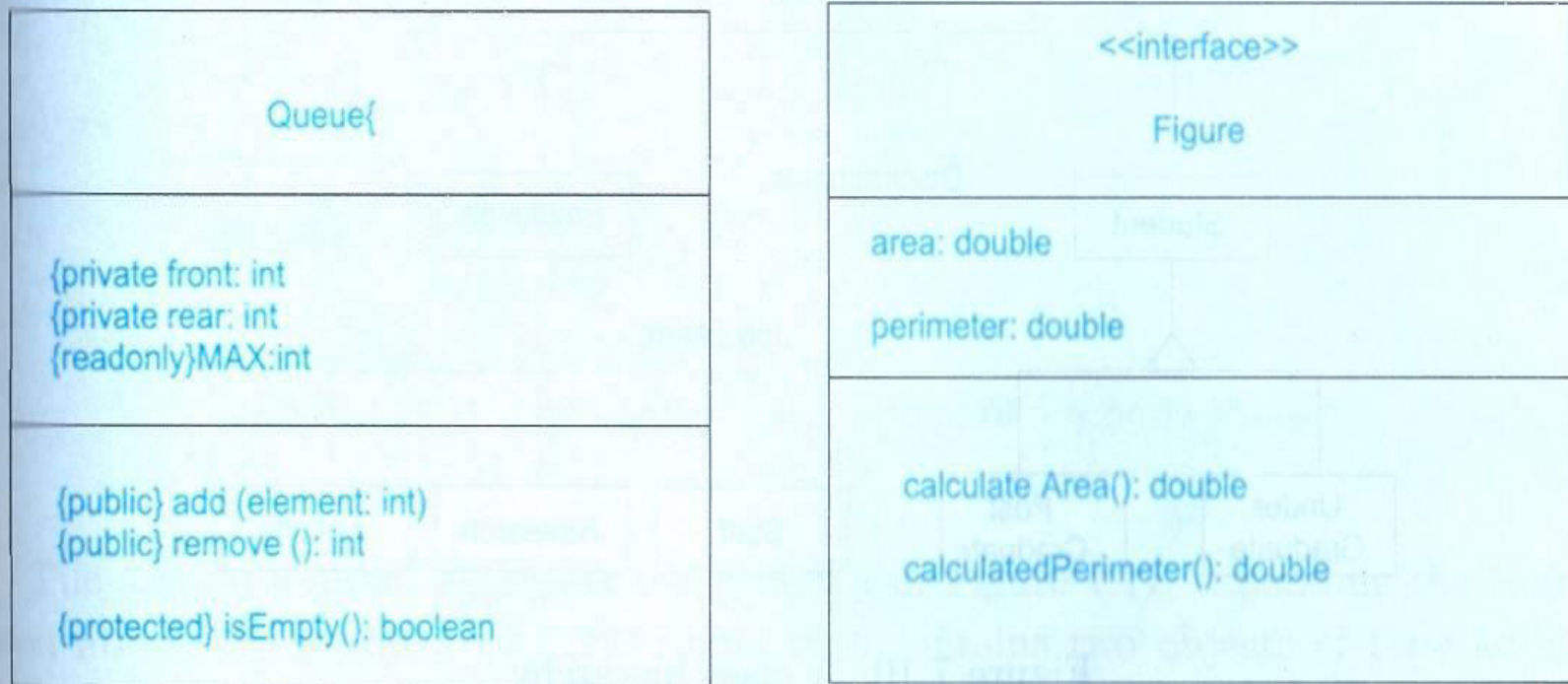


Figure 7.9: Class, stereotypes, and tagged values.

Class Diagram

- A class itself is represented as a rectangular box which is divided into three areas.
- The top part gives the class name.
- The middle part lists the key attributes or fields of the class.
- The bottom part lists the methods or operations of the class.
- If a class is an interface, this can be specified by marking the class with the stereotype
“<< interface>>”, which is written above the class name.
- The generalization-specialization relationship is specified by having arrows coming from the subclass to the super class.

Class Diagram

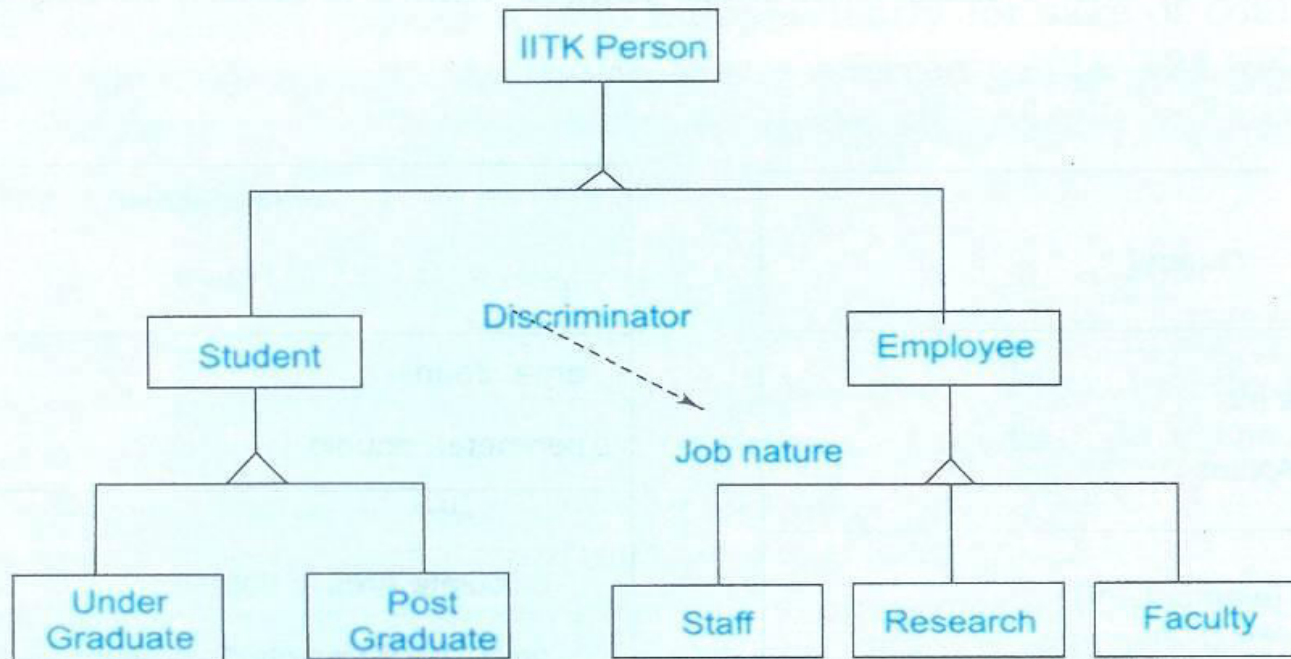


Figure 7.10: A class hierarchy.

Class Diagram

- Another common relationship is association, which allows objects to communicate with each other.
- The association is shown by a line between the two classes.
- Another type of relationship is the part-whole relationship which represents the situation when an object is composed of many parts, each part itself is an object.
- This situation represents containment or aggregation. That is, object of a class are contained inside the object of another class.
- For representing this aggregation relationship, the class which represents the “whole” is shown at the top and a line emanating from a little diamond connecting it to classes which represent the parts.

Class Diagram

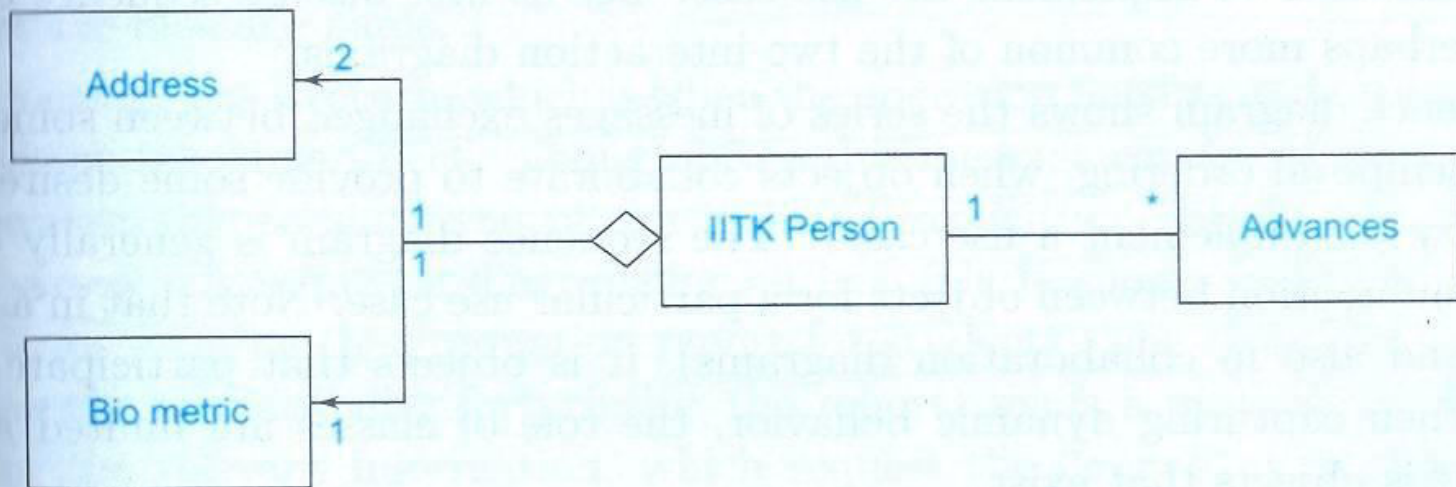
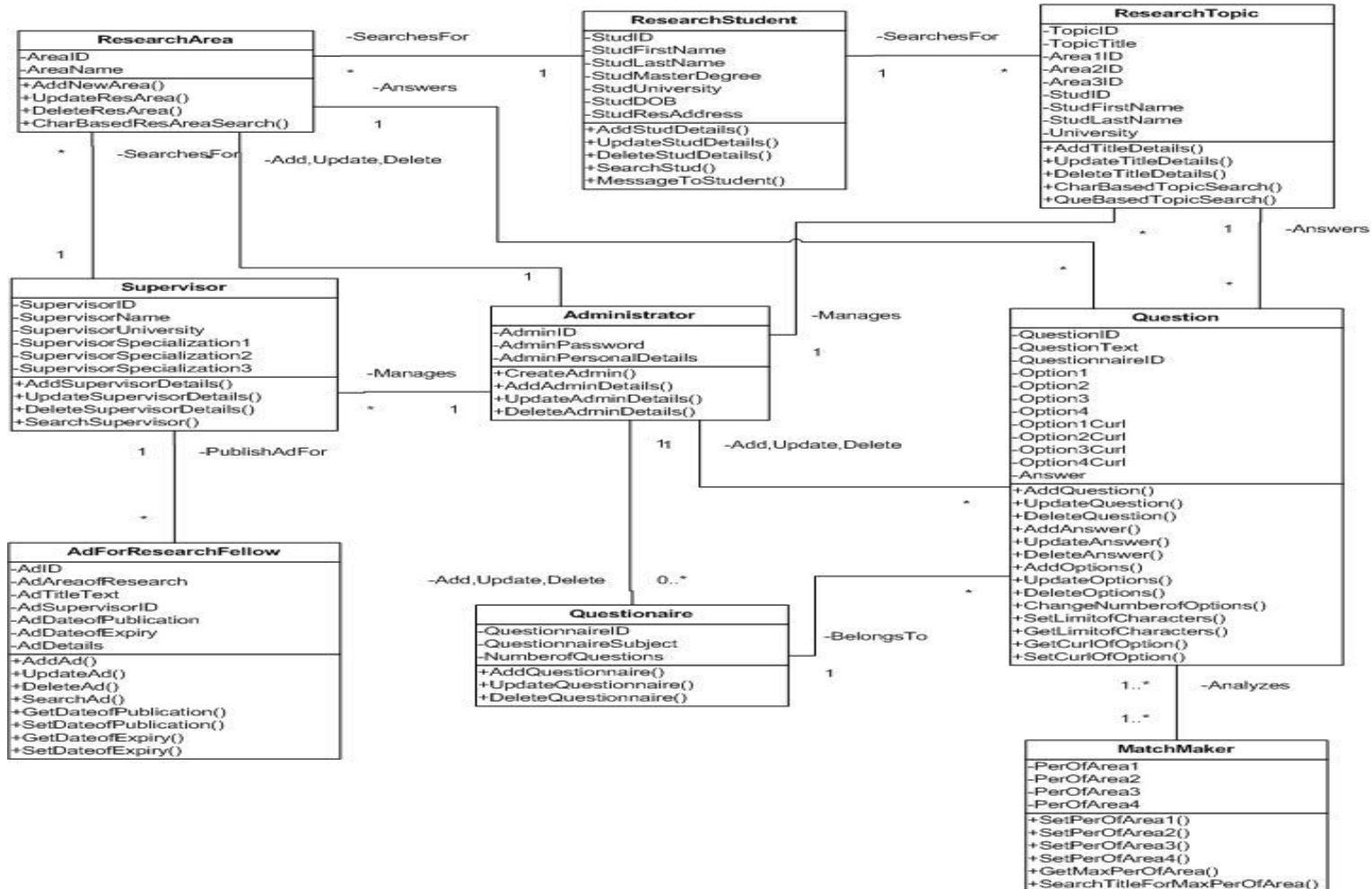


Figure 7.11: Aggregation and association among classes.

Class Diagram





Sequence and Collaboration Diagrams

- Class diagrams do not represent the dynamic behavior of the system. That is, how the system behaves when it performs some of its functions cannot be represented by class diagrams.
- This is done through sequence diagrams or collaboration diagrams, together called interaction diagrams.
- An interaction diagram typically captures the behavior of a use case and models how the different objects in the system collaborate to implement the use case.



Sequence Diagram

- A sequence diagram shows the series of messages exchanged between some objects, and their temporal ordering, when objects collaborate to provide some desired system functionality.
- The sequence diagram is generally drawn to model the interaction between objects for a particular use case.
- Note that in a sequence diagram, it is object that participates and not classes.

Sequence Diagram

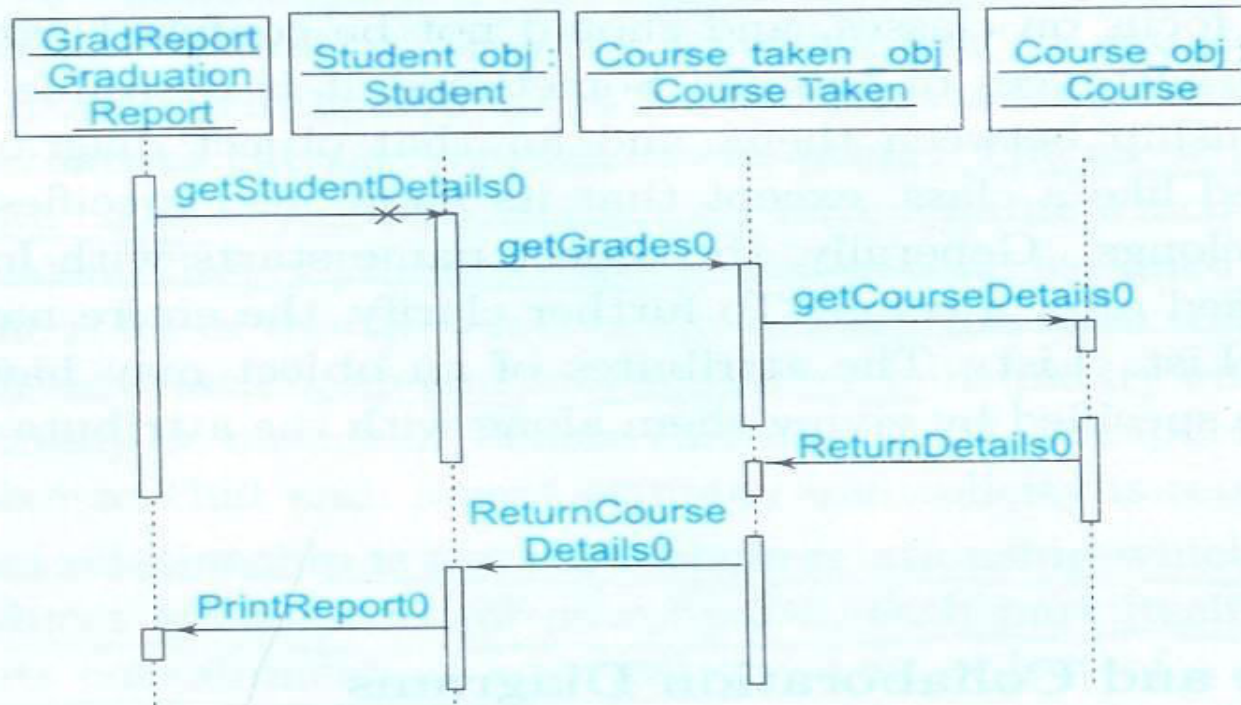
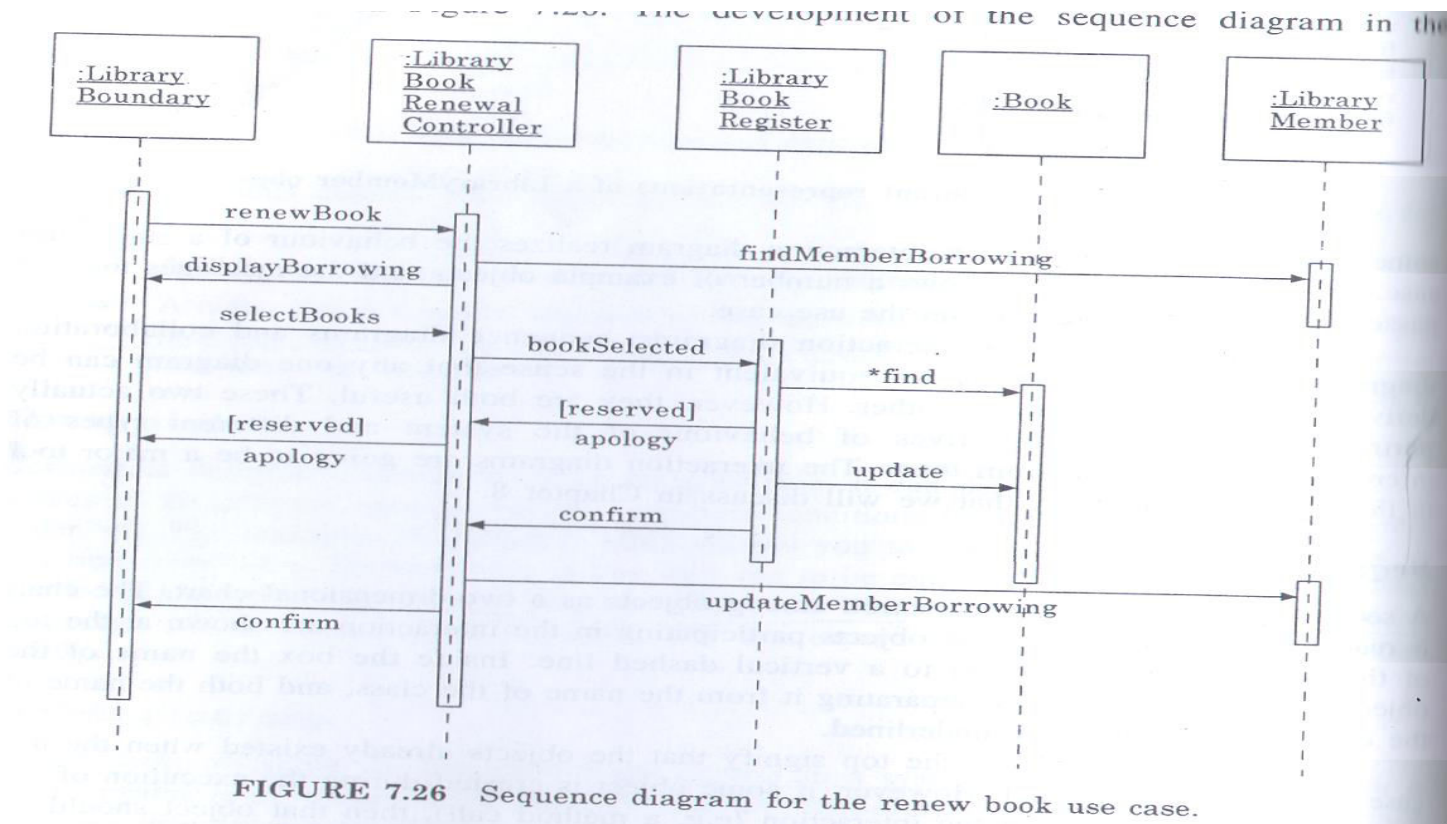


Figure 7.12: Sequence diagram for printing a graduation report.

Sequence Diagram

- In a sequence diagram, all the objects that participate in the interaction are shown at the top as boxes with object names.
- For each object, a vertical bar representing its lifeline is drawn downwards.
- A message from one object to another is represented as an arrow from the lifeline of one to the lifeline of the other.
- Each message is labeled with the message name, which typically should be the name of a method in the class of the target object.
- An object can also make a self call, which is shown as an message starting and ending in the same object lifeline.
- If a message is sent to multiple receiver objects, then this multiplicity is shown by having a “*” before the message name.

Sequence Diagram





Collaboration Diagram

- A collaboration diagram also shows how objects communicate.
- A collaboration diagram looks more like a state diagram.
- Each object is represented in the diagram, and the messages sent from one object to another are shown as numbered arrows from one object to another
- Sequence diagram have become more popular, as people find the visual representation of sequencing quicker to grasp.

Collaboration Diagram

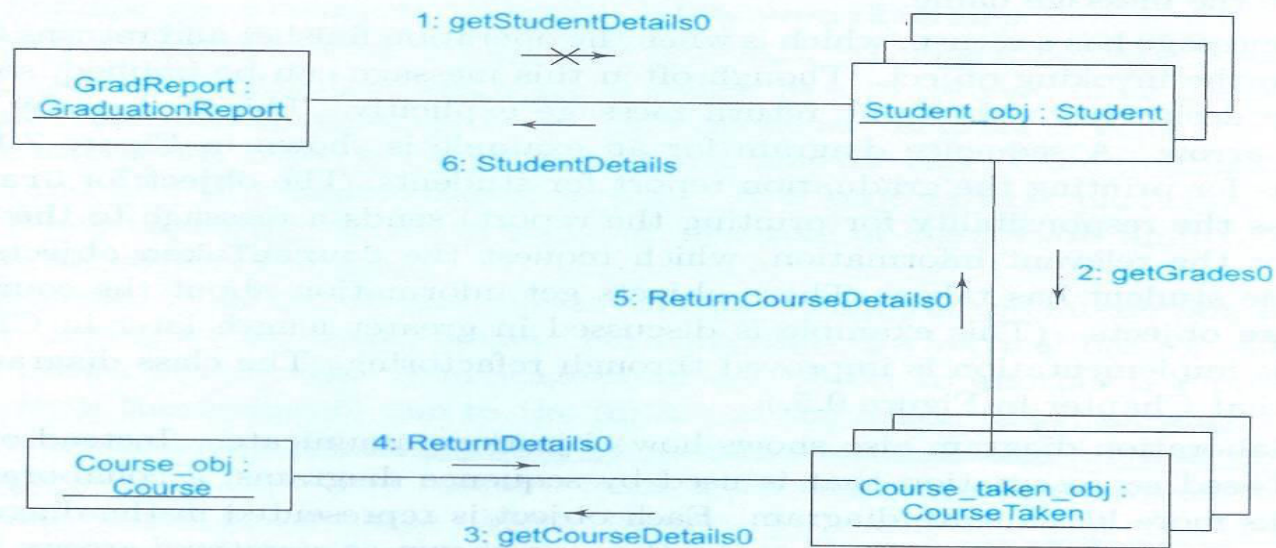


Figure 7.13: Collaboration diagram for printing a graduation report.

Collaboration Diagram

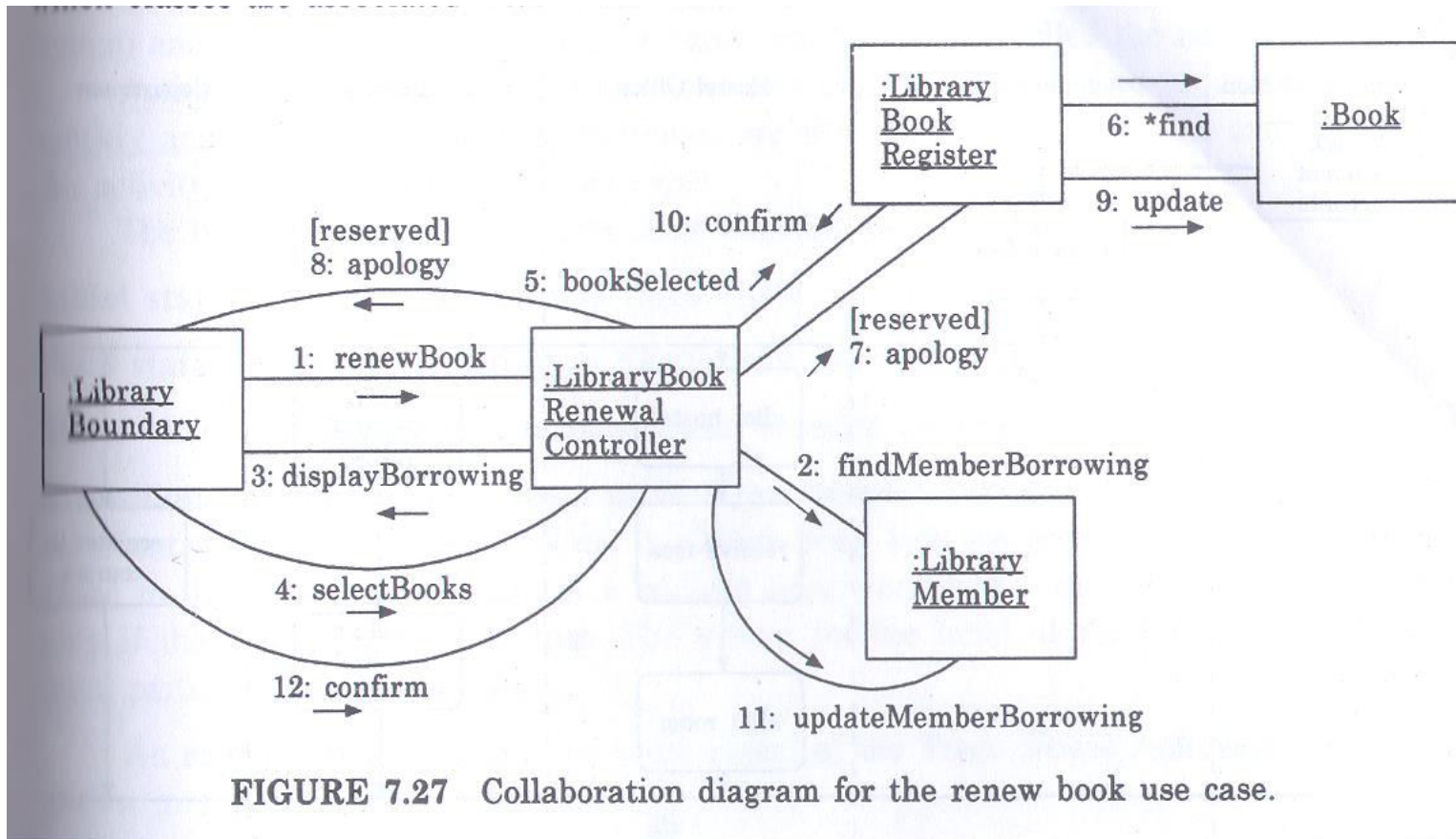


FIGURE 7.27 Collaboration diagram for the renew book use case.

State Diagram

- A state diagram is a model in which the entity being modeled is viewed as a set of states, with transitions between the states taking place when some event occurs.
- A state is represented as a rectangle with rounded edges or as ellipses or circles.
- Transitions are represented by arrows connecting two states.
- State diagrams are often used to model the behavior of objects of a class.

State Diagram

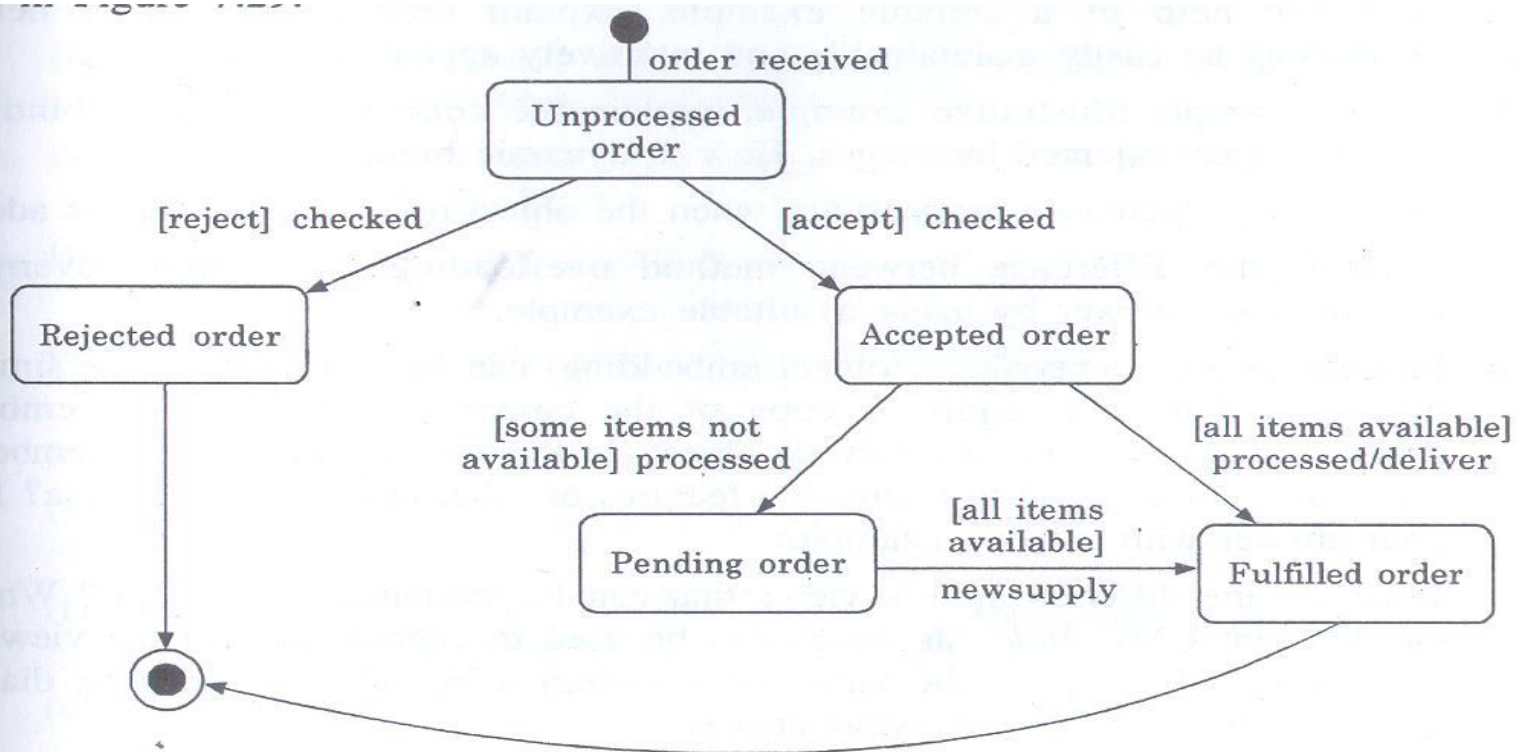


FIGURE 7.29 State chart diagram for an order object.

Activity Diagram

- Activity Diagrams provide another method for modeling dynamic behavior.
- These diagrams model a system by modeling the activities that take place in it when the system executes for performing some function.
- Each activity is represented like an oval, with the name of the activity within it.
- From the activity, the system proceeds to other activities.
- Often, which activity to perform next depends on some decision. This decision is shown as a diamond leading to multiple activities.

Activity Diagram

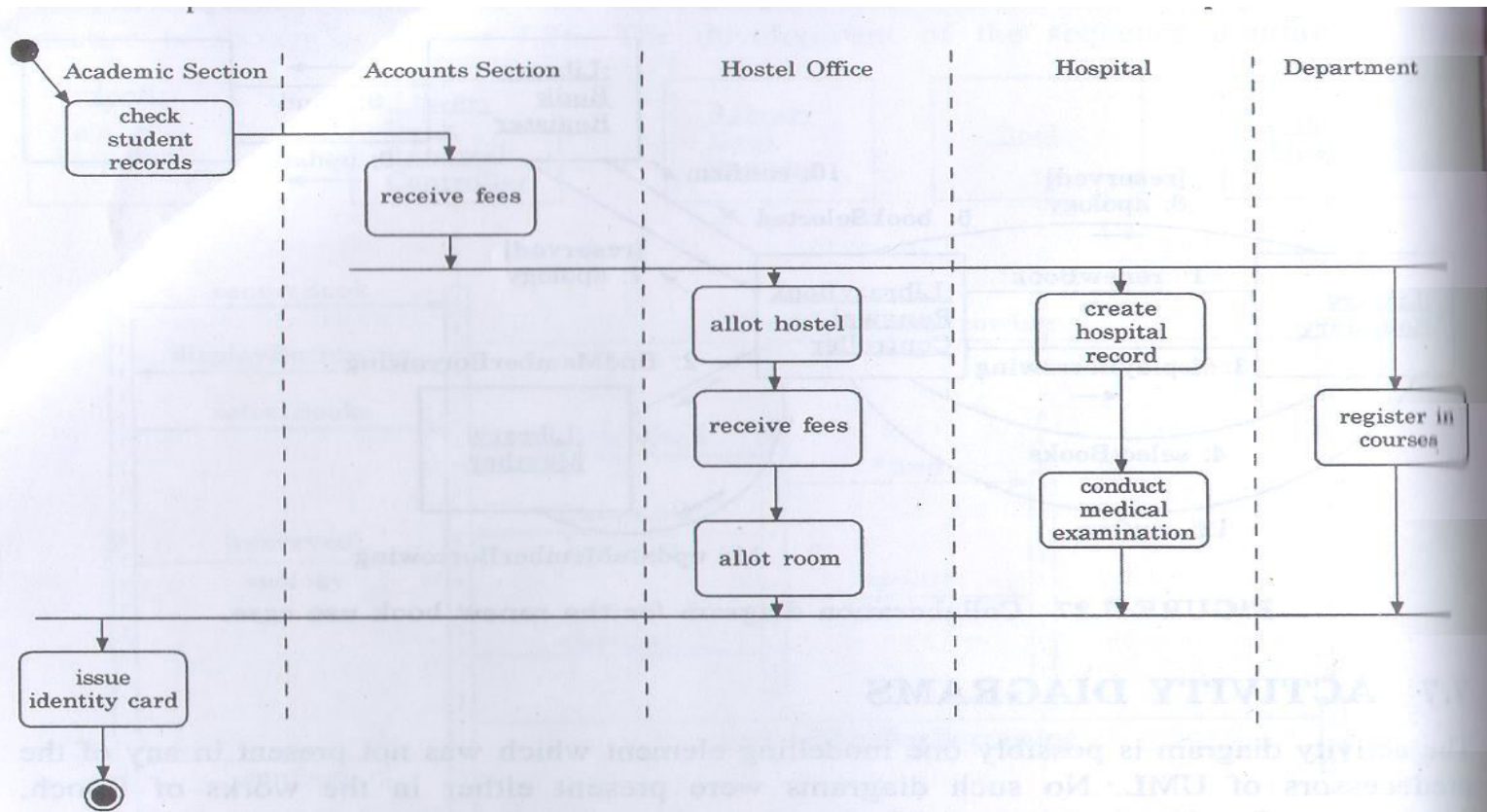


FIGURE 7.28 Activity diagram for student admission procedure at IIT.



A Design Methodology

- A complete OO design should be such that in the implementation phase only further details about methods or attributes need to be added.
- On OO design, OO analysis forms the starting step.
- Using the model produced during analysis, a detailed model of the final system is built.
- The design methodology for producing an OO design consists of the following sequence of steps:
 1. Produce the class diagram
 2. Produce the dynamic model and use it to define operations on class.
 3. Produce the functional model and use it to define operations on classes.
 4. Identify internal classes and operations.
 5. Optimize and package




Dynamic Modeling

- The dynamic model of a system aims to specify how the state of various objects changes when events occur.
- For an object, an event is essentially a request for an operation.
- Each event has an initiator and a responder.
- Event can be internal to the system, in which case the event initiator and the event responder are both within the system.
- In use cases, dynamic modeling involves preparing interaction diagrams for the important scenarios, identifying events on classes, ensuring that events can be supported, and perhaps build state models for the classes.




Functional Modeling

- The functional model describes the computations that take place within a system.
- It specifies what happens in the system.
- A functional model of a system specifies how the output values are computed in the system from the input values, without considering the control aspects of the computation.
- Generally, when the transformation from the inputs to outputs is complex, consisting of many steps, the functional modeling is likely to be useful.



Defining Internal Classes and Operations

- The classes identified so far are the ones that come from the problem domain.
- While considering implementation issues, algorithm and optimization issues arise.
- These issues are handled in this step:
- First, each class is critically evaluated to see if it is needed in its present form in the final implementation. (some may be discarded)



Defining Internal Classes and Operations

- Then the implementation of operations on the classes is considered. For this, rough algorithms for implementation might be considered.
- While doing this, complex operation may get defined in terms of lower-level operations on simpler classes.
- Once the implementation of each class and each operation on the class has been considered and it has been satisfied that they can be implemented, the system design is complete.



Optimize and Package

- In this final step, the issue of efficiency is considered, keeping in mind that the final structures should not deviate too much from the logical structure produced by analysis, as the more the deviation, the harder it will be to understand a design.



Detailed Design and PDL

- Process design language (PDL) is one way in which the design can be communicated precisely and completely to whatever degree of detail desired by the designer.
- It can be used to specify the system design and to extend it to include the logic design.
- PDL is useful in top-down refinement techniques to design a system or module.



Detailed Design and PDL

- One way to communicate a design is to specify it in a natural language, which leads to misunderstanding.
- This approach is not useful while converting design into code.
- The other extreme is to communicate it precisely in a formal language, like a programming language.
- Such representations often have great detail.



Detailed Design and PDL

- Ideally, we would like to express the design in a language that is as precise and unambiguous as possible without having too much detail and that can be easily converted into an implementation.
- The PDL has an overall outer syntax of a structured programming language and have a vocabulary of a natural language.
- It can be thought of an “Structured English”.
- One way to use PDL is to first generate a rough outline of the entire solution at a given level of detail.
- When the design is agreed on at this level, more detail can be added.



Detailed Design and PDL

- This allows successive refinement approach, and can save considerable cost by detecting the design errors early during the design phase.
- It also aids design verification by phases, which helps in developing error-free designs.
- The basic constructs of PDL are similar to those of a structured language.
- For example, IF construct is similar to the other programming language, but the conditions and the statements to be executed need not be stated in a formal language.

Detailed Design and PDL

```
Initialize buf to empty
DO FOREVER
    DO UNTIL (#chars in buf ≥ 80 & word boundary is reached)
        OR (end-of-text reached)
        read chars in buf
    ENDDO
    IF #chars > 80 THEN
        remove last word from buf
        PRINT-WITH-FILL (buf)
        set buf to last word ELSEIF #chars = 80 THEN
        print (Buf)
        set buf to empty
    ELSE EXIT the loop
ENDDO

PROCEDURE PRINT-WITH-FILL (buf)

Determine #words and #character in buf
#of blanks needed = 80 - #character
DO FOR each word in the buf
    print (word)
    if #printed words ≥ (#word - #of blanks needed) THEN
        print (two blanks)
    ELSE print (single blank)
ENDDO
```

Figure 8.2: PDL description of text-formatter.

Detailed Design and PDL

- The iteration criteria can be chosen to suit the problem, and unlike a formal

```
minmax(infile)

ARRAY a

DO UNTIL end of input
    READ an item into a
ENDDO
max, min := first item of a
DO FOR each item in a
    IF max < item THEN set max to item
    IF min > item THEN set min to item
ENDDO

END
```

Figure 8.1: PDL description of the minmax program.



Verification

- There are few techniques available to verify that the detailed design is consistent with the system design.
- The focus of verification in the detailed design phase is on showing that the detailed design meets the specifications laid down in the system design.
- The three verification methods we consider are:
 1. Design Walkthroughs
 2. Critical Design Review
 3. Consistency Checkers



Design Walkthroughs

- A design walkthrough is a manual method of verification.
- It is done in an informal meeting called by the designer or the leader of the designer's group.
- In a walkthrough the designer explains the logic step by step, and the members of the group ask questions, point out possible errors or seek clarification.



Critical Design Review

- The purpose of critical design review is to ensure that the detailed design satisfies the specifications laid down during system design.
- The critical design review process is same as the inspections process.



Consistency Checkers

- If the design is specified in PDL or some other formally defined design language, it is possible to detect some design defects by using consistency checkers.
- Consistency checkers are compilers that take as input the design specified in a design language (PDL).
- A consistency checker can ensure that any modules invoked or used by a given module actually exist in the design and that the interface used by the caller is consistent with the interface definition of the called module.