SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – I - Object Oriented Analysis and Design– SBSA1403

OBJECT ORIENTED ANALYSIS AND DESIGN– SBSA1403

COURSE OBJECTIVES:

- To Understand the fundamentals of Object-Oriented System Development
- To understand the object-oriented methodologies.
- To use UML in requirements elicitation and designing.
- To understand concepts of relationships and aggregations.
- To test the software against its requirements specification

UNIT I - INTRODUCTION

Overview of object-oriented language systems development – Object basics hierarchy – Object and identity –Static and dynamic binding – Object oriented SDLC.

UNIT II - OBJECT ORIENTED METHODOLOGIES

Rumbaugh et al.'s technique – Booch, Jacobson Methodologies – Patterns – Framework – Unified approach –UML – UML diagrams – UML dynamic modelling – UML extensibility – UML meta-model.

UNIT III - OBJECT ORIENTED ANALYSIS

Use case model – Object analysis classification – Approaches for identifying classes – Classes responsibilities and collaborators – Identifying object relationships, attributes and methods.

UNIT IV - OBJECT ORIENTED DESIGN

Design process and design axioms – Designing classes – Access Layer – Object storage and object Interoperability – View layer – Designing interface objects.

UNIT V - SOFTWARE QUALITY

Software quality assurance – Testing strategies – Test cases – Test plan – Myers debugging principle – System usability and measuring user satisfaction.

COURSE OUTCOMES

CO1 - Understand the basics object model for System development.

CO2 - Understand the object-Oriented Methodologies

CO3 - Express software design with UML diagrams

CO4 - Understand the concept of Relationships

CO5 - Design software applications using OO concepts.

CO6 -Understand the various testing methodologies for OO software

UNIT – I

INTRODUCTION

1. Overview of object-oriented language systems development

- System development refers to all activities that produce an information systems solution.
- System development activities are,
    - Analysis
    - Modelling
    - Design
    - Implementation
    - Testing
    - Maintenance
- A **software development methodology** is a series of processes that can lead to the development of an application.
- Two orthogonal views of the software,
    - Algorithms + Data structures = Programs
- **Program:** A software system is a mechanism for performing certain action on certain data.

1.1 The two orthogonal views of the software

- Traditional system development methodology
    - This approach focuses on the **“Functions”**
- Object oriented system development
    - This approach focuses on the **“Object”** which combines **“Data and Functionality”**

1.2 Object oriented system development methodology

- OOSD methodology is based on functions and procedures.
- OOSD is the way to develop software by building self- contained modules or objects that can be easily,
    - Replaced
    - Modified
    - Reused.
- In an object-oriented system, everything is an object: numbers, arrays, records, fields, files, forms, an invoice, etc.
- An Object is anything, real or abstract, about which we store data and those methods that manipulate the data.
- Conceptually, each object is responsible for itself.
    - E.g., 1. A chart object is responsible for things like maintaining its data andlabels, and even for drawing itself.
    - E.g., 2. A window object is responsible for things like opening, sizing, andclosing itself.

1.3 Why an object orientation?

- To create sets of objects that work together with to produce software the problem.
- The systems are easier to adapt to changing requirements, easier to maintain, more robust and promote greater design and code reuse.
- Object oriented development allows us to create modules of functionality.
- Once objects are defined, they will perform their desired functions.
- Here are some reasons,
    - Higher level of abstraction
    - Seamless transition among different phases of software development.
    - Encouragement of good programming techniques
    - Promotion of reusability
- Higher level of abstraction
    - The top-down approach supports abstraction at the function level
    - The object-oriented approach supports abstraction at the object level.Since **“objects encapsulate both data and functions they work at a higher level of abstraction.”**
- Seamless transition among different phases of software development.
    - The object-oriented approach essentially uses the same language to talk about analysis, design, programming, database design.
    - This seamless approach **reduces the level of complexity and redundancy and makes for clear, more robust system development**.
- Encouragement of good programming techniques
    - Object oriented languages produce more modular and reusable code via the concepts of class and inheritance.
    - The object-oriented languages are C++, Smalltalk or Java.
- Promotion of reusability
    - Objects are reusable because they are modelled directly out of a real-world problem domain.
    - The object orientation adds inheritance which is a powerful technique that allows classes to be built from each other.

1.4 Overview of Unified approach

- Unified approach is a methodology for software development.
- The UA is based on methodologies by BOOCH, RUMBAUGH & JACOBSON tries to combine the best practices, processes and guidelines along with the object management group’s unified modelling language.
- The unified modelling language (UML) is a set of notations and conventions used to describe and model an application.
- Applying past development efforts to future projects will improve the quality of the product and reduce the cost and development time.
- Steps to be followed in unified approach are

1. Identify the users/actors
2. Develop a simple business process model
3. Develop the use case
4. Interaction diagrams
5. Classification
6. Apply design axioms to design classes
7. Design the access layer
8. Design the view layer
9. Iterate and refine the design/analysis.

2. Object basics

- Object: It is a real-world entity that has properties and methods.
    - E.g., Car.
- Properties (or Attributes): It describes the state (or data) of an object, refer Fig.1
    - E.g., Color, manufacturer, cost, model and owner etc.

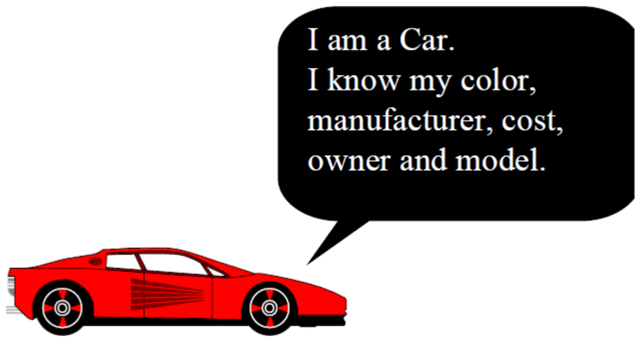


Fig. 1.1. Car Attributes

- Methods (or Procedures): It defines its behavior (or operations), refer Fig.2
    - E.g., Go, stop, turn left, turn right and etc.…

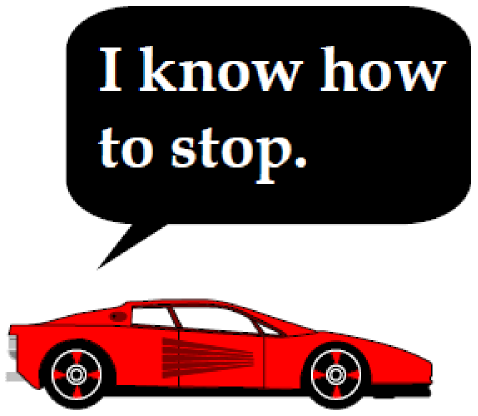


Fig. 1.2. Car Methods

- When developing an object-oriented application, two basic questions always arise,
  - What objects does the application needs?
  - What functionality should those objects have?

2.1 Object-Oriented Philosophy

- **Object oriented programming** is that it allows the base concepts of the language to be extended to include ideas and terms closer to those of its applications.
- The **traditional development methodologies** are either algorithm centric or data centric
  1. **Algorithm-centric methodology:** An algorithm that can accomplish the task, and then build data structures for that algorithm to use.
  2. **Data-centric methodology:** To structure the data, and then build the algorithm around that structure.
- **Object-oriented methodology** is the algorithm and the data structures are packaged together as an object, which has a set of attributes or properties.

2.2 Objects are grouped into classes

- Classes are used to distinguish one type of objects from another
- A class is a set of objects that share a common structure and a common behavior
- Each object is an instance of a class, refer Fig. 3 and 4
- A class is a specification of,
  1. Structure (instance variable)
  2. Behaviour (methods)
  3. Inheritance for objects

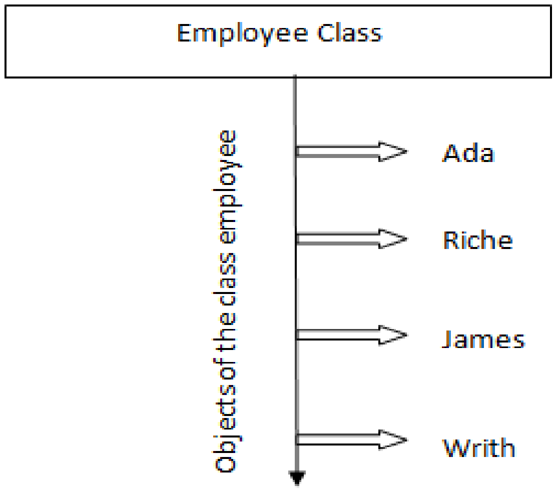


Fig. 1.3. Ada, Riche, James and Writh are objectsof the class Employee

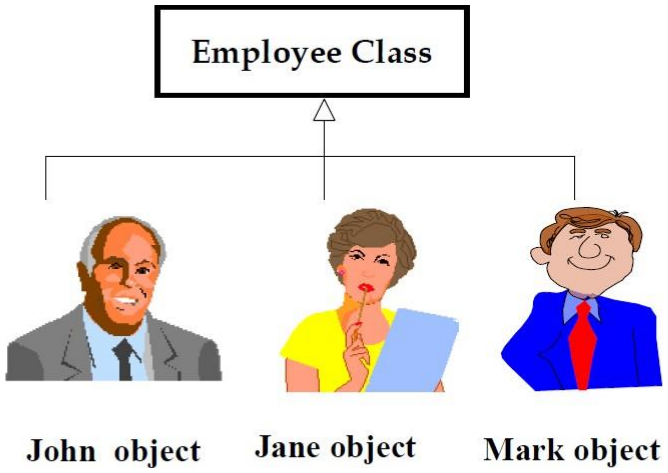


Fig. 1.4. Employee class

2.3 Attributes or properties

- It represented by data type.
- It describe the state of an object
- Eg. Car attributes, see Fig. 5.

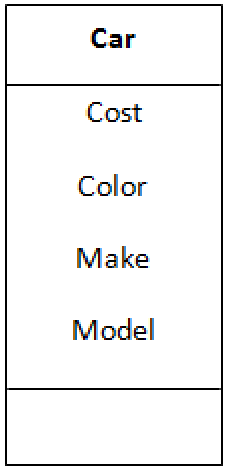


Fig. 1.5. Attributes

2.4 Behavior or methods

- An object behaviour is described in methods or procedures
- A method implements the behaviour of an object.
- Behaviour denotes the collection of methods that abstractly describes what an object is capable of doing.
- Eg. Drive a car – Methods, refer Fig.6.

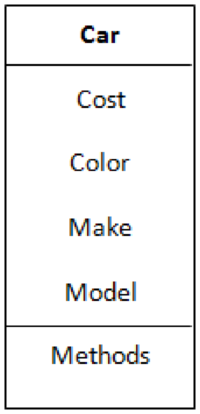


Fig. 1.6. Methods

2.5 Objects Respond to Messages

- A message is much more general than function call.
- E.g., Problem is to make French onion soup. Sol: Your instruction is the message, the way the French onion soup prepared is the method, and the French onion soup is the object.
- Objects perform operations in response to messages.
- Objects respond to messages according to methods defined in its class.
- E.g., See the Fig. 7
    1. Send a Brake message to a Car object
    2. Send a multiplication *7 message to 5 objects

3. Send a Compute payroll message to Employee object

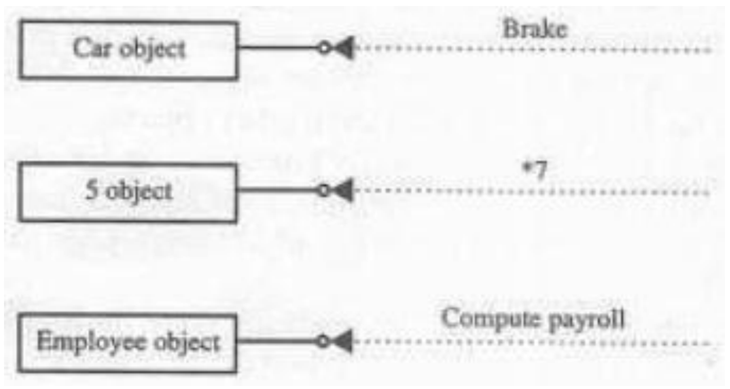


Fig. 1.7. Object responds to message

2.6 Encapsulation and information hiding

- Information hiding is the principle of hiding internal data and procedures of an object, refer Fig.8

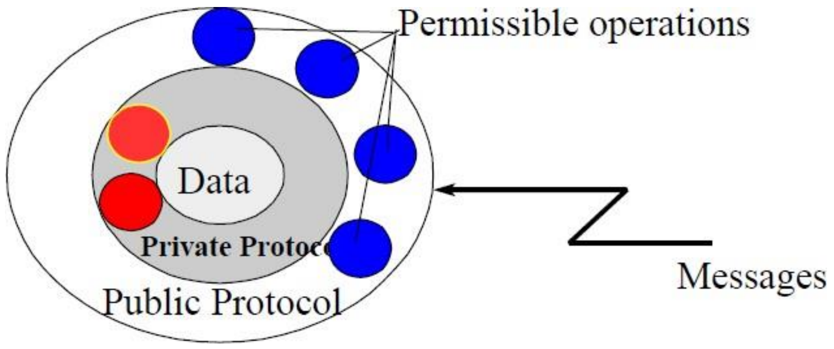


Fig. 1.8. Access specifiers

- Encapsulation protects the data from corruption
- Providing an interface to each object in such a way as to reveal as little as possible about inner workings.
- Encapsulation means in general, protection mechanism by using access specifiers such as public, protected and private.
- Data are abstracted when they are shielded or hided by a full set of methods and only those methods can access the data portion of an object.
- E.g., A car engine.

3. Class Hierarchy

- An object-oriented system organized classes into a “Subclass-Super class hierarchy”.
- At the top of the hierarchy are the most general (base) classes and at the bottom are the most specific classes (sub-classes).
- The parent class also is known as the bases class or super class.
- A subclass inherits all of the properties and methods (procedures) defined in its super class, refer Fig. 9 and 10.

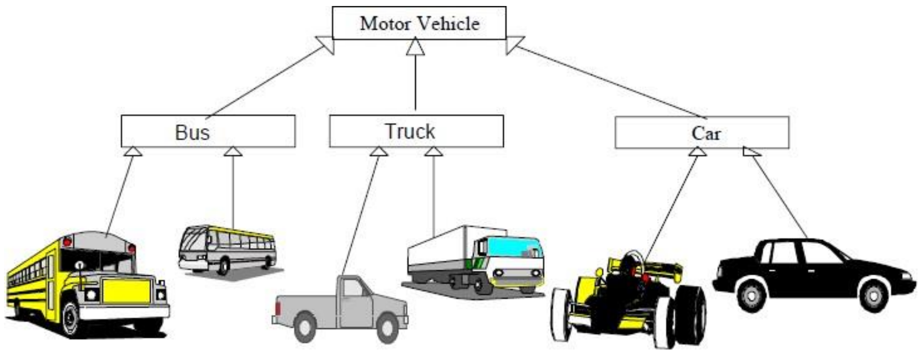


Fig. 1.9. Super class / Sub class Hierarchy

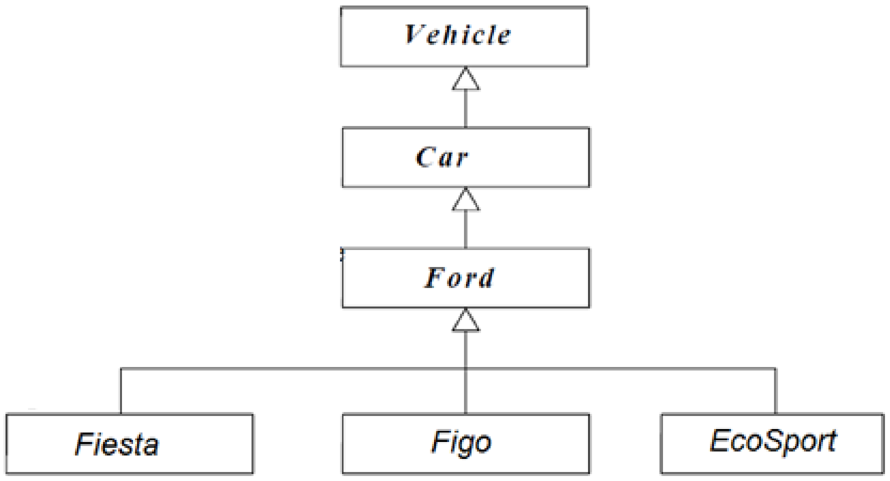


Fig. 1.10. Class hierarchy for Ford class

3.1 Inheritance

- Inheritance is a relationship between classes where one class is the parent class of another (derived) class, refer Fig. 11.
- Inheritance allows classes to share and reuse behaviours and attributes.
- The real advantage of inheritance is that we can build and reuse what we already have.

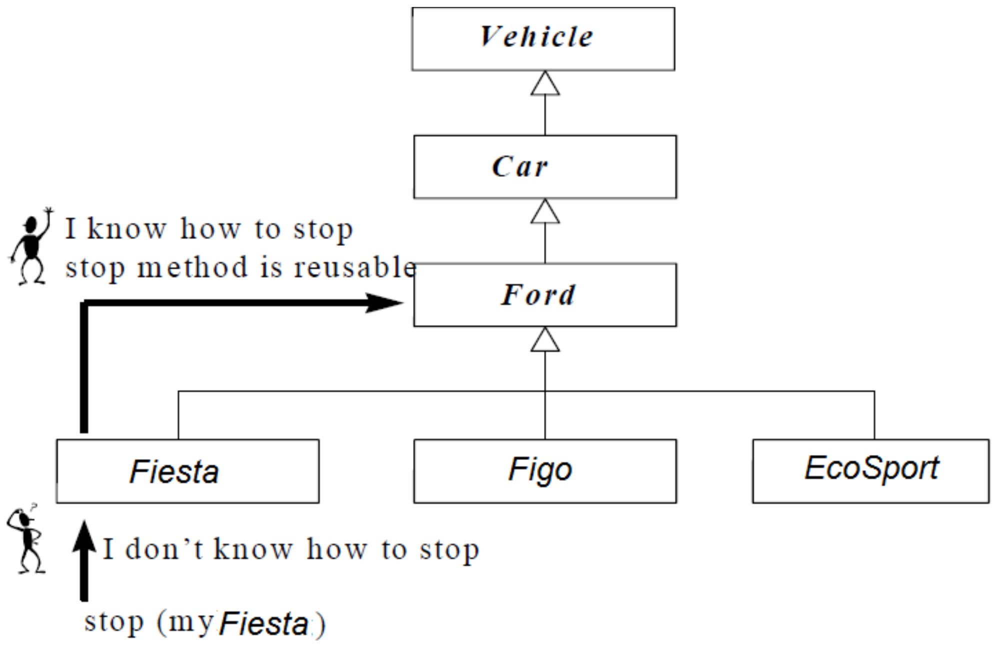


Fig. 1.11. Inheritance allows reusability

- **Dynamic Inheritance**
  1. It allows objects to change and evolve over time.
  2. Dynamic inheritance refers to the ability to add, delete or change parents from objects at run time.

3.2 Multiple Inheritance

- Object oriented systems permit a class to inherit its state (attributes) and behaviors from more than one super class. This kind of inheritance is referred to as "Multiple Inheritance". Refer Fig. 12.

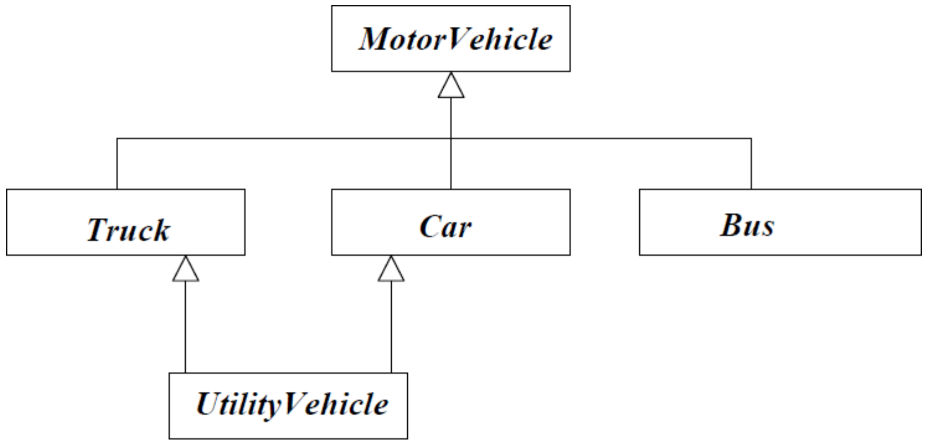


Fig. 1.12. Utility vehicle inherits from both the car and truck class

3.3 Polymorphism

- Poly means “Many” and morph means “Form”
- Polymorphism means that the same operation may behave differently on different classes. Refer Fig. 13 and 14

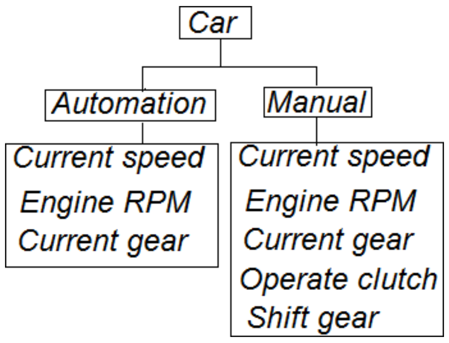


Fig. 1.13. Car object represents polymorphism

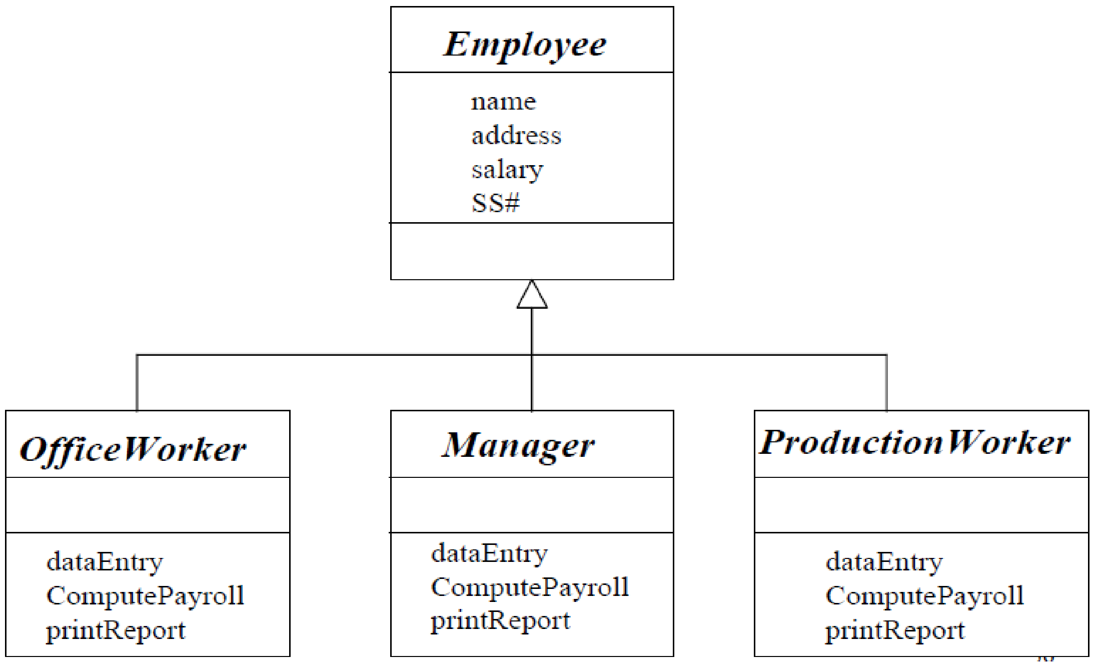


Fig. 1.14. Compute Payroll represents polymorphism

3.4 Object relationship and Association

- The concept of association represents relationships between objects and class.
- Associations are bidirectional by which they can be traversed in both directions
- E.g., A pilot can fly planes, refer Fig.15.

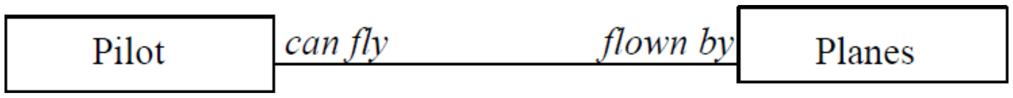
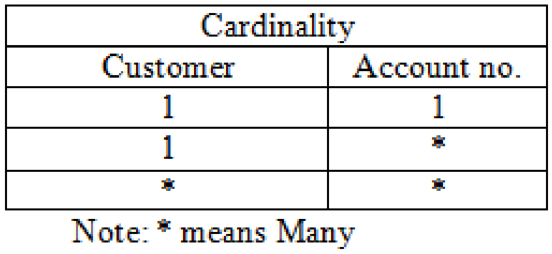


Fig. 1.15. Association represents the relationship among objects which is bidirectional

- **Cardinality:** it is an important issue in associations which specifies how manyinstances of one class may relate to a single instance of a associated class.
- E.g., Client account relationship, refer Fig. 16.



Cardinality	
Customer	Account no.
1	1
1	*
*	*

Note: * means Many

Fig. 1.16. Cardinality in association

- Consumer - producer association, also known client-server association or use relationship.
- A special form of association is a client-server relationship.
- This relationship can be viewed as one-way interaction: one object (client) requests the service of another object (server). Refer Fig. 17.

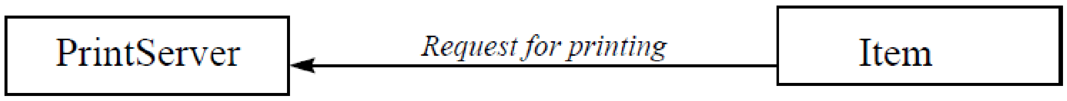


Fig. 1.17. One-way direction association

4. Object and Identity

- A special future of object-oriented system is that every object has its own unique and immutable identity.
- The identity name never changes even it all the properties of the object change, i.e., it isindependent of the object's state.

- The identity does not depend on the objects name, or its key or its location.
- In an object system, object identity often is implemented through some kind of "OBJECT IDENTIFIER" (OID) or "UNIQUE IDENTIFIER" (UID). An OID is dispersed by a part of object-oriented programming system that is responsible for guaranteeing the uniqueness of every identifier.
- OID's are never reused.
- E.g., The car may have an **owner** property, which refer the class person. A person has an **owns** property that contains a reference to the car instance. The relationship between the car and person can be implemented and maintained by a reference, refer Fig. 18.

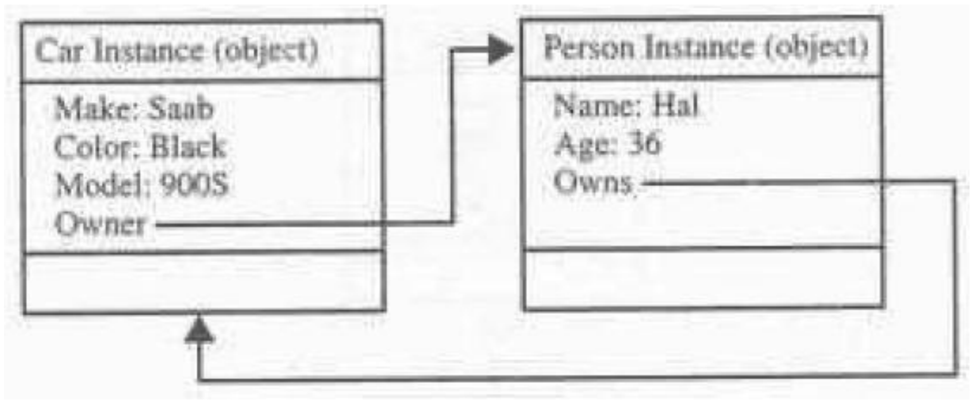


Fig. 1.18. The owner property of the car contains the reference to the personinstance named Hal

4.1 Static and Dynamic Binding

- o The process of determining, which function to invoke at run time is called as "Dynamic binding".
- o E.g., Polymorphism function calls
- o Making this determination earlier, at compile time is called as "Static binding".
- o E.g., Normal function calls

4.2 Object Persistence

- o Objects have a life time. They are explicitly created and can exist for a period of time.
- o An object can persist beyond application session boundaries, during which the object is stored in a database or a file.

4.3 Meta Classes

- o Class is an object. If it is an object it must belong to a class, such a class belongs to a class called a "Meta Class" or a class of classes.
- o Meta-classes are used by a compiler.
    1. E.g. The meta-classes handle messages to classes, such as constructors, new and class variables, which are variables shared between all instance of a class.

5. Object Oriented – SDLC

- The essence of the software development process that consists of
  - Analysis
  - Design
  - Implementation
  - Testing
  - Refinement
- To transform users' needs into a software solution that satisfies those needs.

5.1 Software Development Process

- "System or software development can be viewed as a process"
- The process can be divided into small, interacting phases called sub-processes.
- Each sub-processes must have the following
  - A description in terms of how it works.
  - Specification of the input required for the process.
  - Specification of the output to be produced.
- The "Software development process" can be viewed as a series of transformations, where the output of one transformation becomes the input of the subsequent transformation. Refer Fig.19.

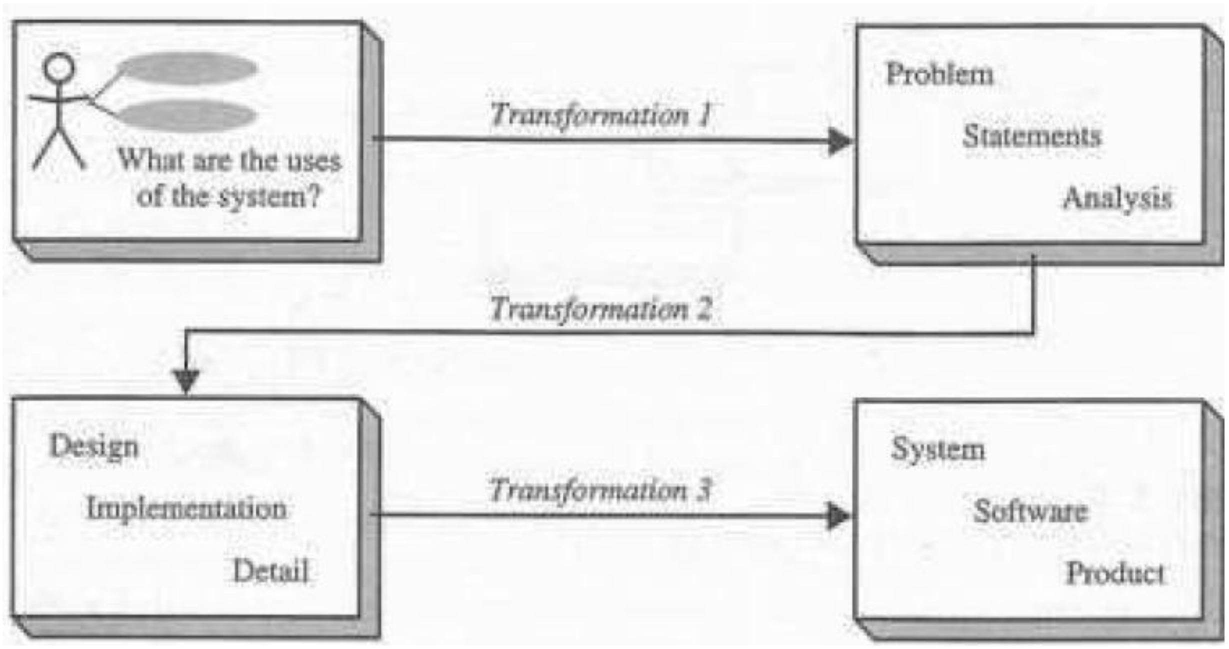


Fig. 1.19. Software process reflecting transformation from needs to a software product that satisfiesthose needs.

- **Transformation 1 (Analysis):** Translates the users' needs into system requirementsand responsibilities.
- **Transformation 2 (Design):** Begins with a problem statement and ends with adetailed design that can be transformed into an operational system.
- **Transformation 3 (Implementation):** Refines the detailed design into the systemdeployment that will satisfy the user's needs.

"An Example (see Fig. 20) of the software development process is Waterfall approach"

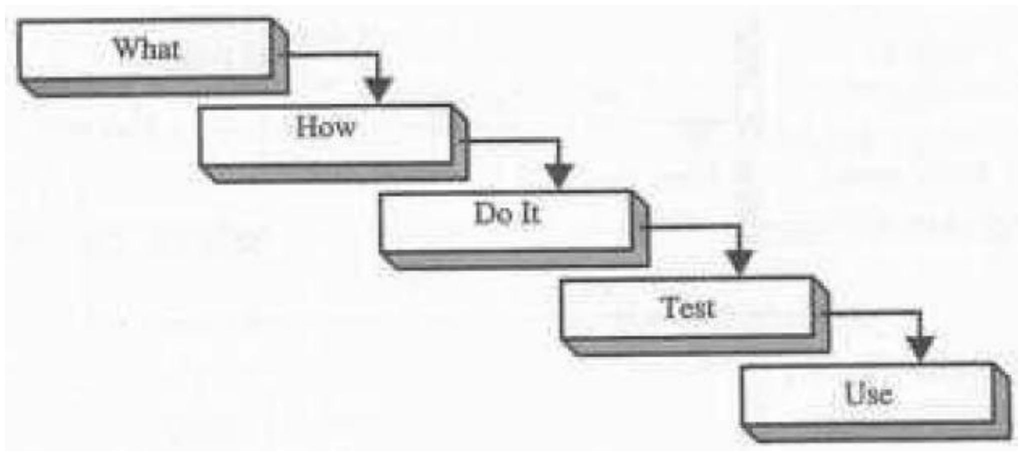


Fig. 1.20. The waterfall software development process

"Waterfall Approach" starts with deciding **what** is to be done. Once the requirements have been determined, we must decide **how** to accomplish them. This is followed by a step in which we **do it**, whatever it has required us to do. We then must**test** the result to see if we have satisfied the user's requirements, finally we **use** what we have done.

5.2 Building high quality Software

- The ultimate goal of building high quality software is user satisfaction.
- To achieve high quality in software we need to be able to answer the following questions.
  1. How do we determine when the system is ready for delivery?
  2. Is it now an operational system that satisfies user's needs?
  3. Is it correct and operating as we thought it should?
  4. Does it pass an evaluation process?
- Four Quality measures for software evaluation (see Fig. 21)
  1. **Correspondence:** It measures how well the delivery system matches the needsof the operational environment
  2. **Validation:** It is the task of predicting correspondence

3. **Correctness:** It measures the consistency of the product requirements withrespect to the design specification
4. **Verification:** It is the exercise of determining correctness

- **Verification:** Am, I building the product, right?
- **Validation:** Am, I building the right product?

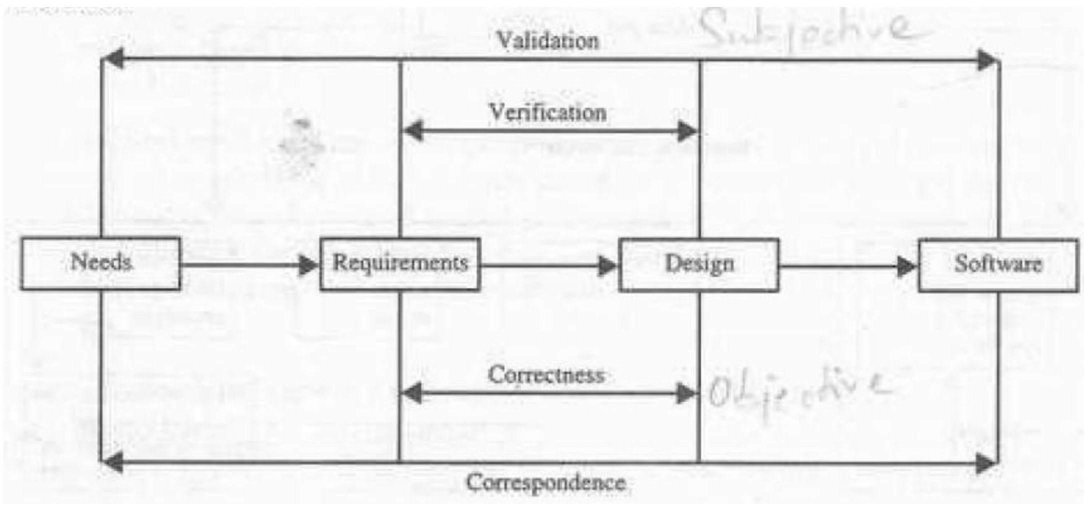


Fig. 1.21. Four quality measures for software evaluation

Validation begins as soon as the project starts, but verification can begin only after a specification has been accepted. Validation & verification are independent of each other.

6. Object Oriented System Development Life Cycle (SDLC): A Use Case Driven Approach

- The object oriented SDLC (Software Development Life Cycle) consists of 3 macro processes, refer Fig. 22.
  i) Object Oriented Analysis (OOA)
  ii) Object Oriented Design (OOD)
  iii) Object Oriented Implementation (OOI)
- It includes the following activities
  1. Object Oriented Analysis – Use case driven
  2. Object Oriented Design
  3. Prototyping
  4. Component based development
  5. Incremental testing

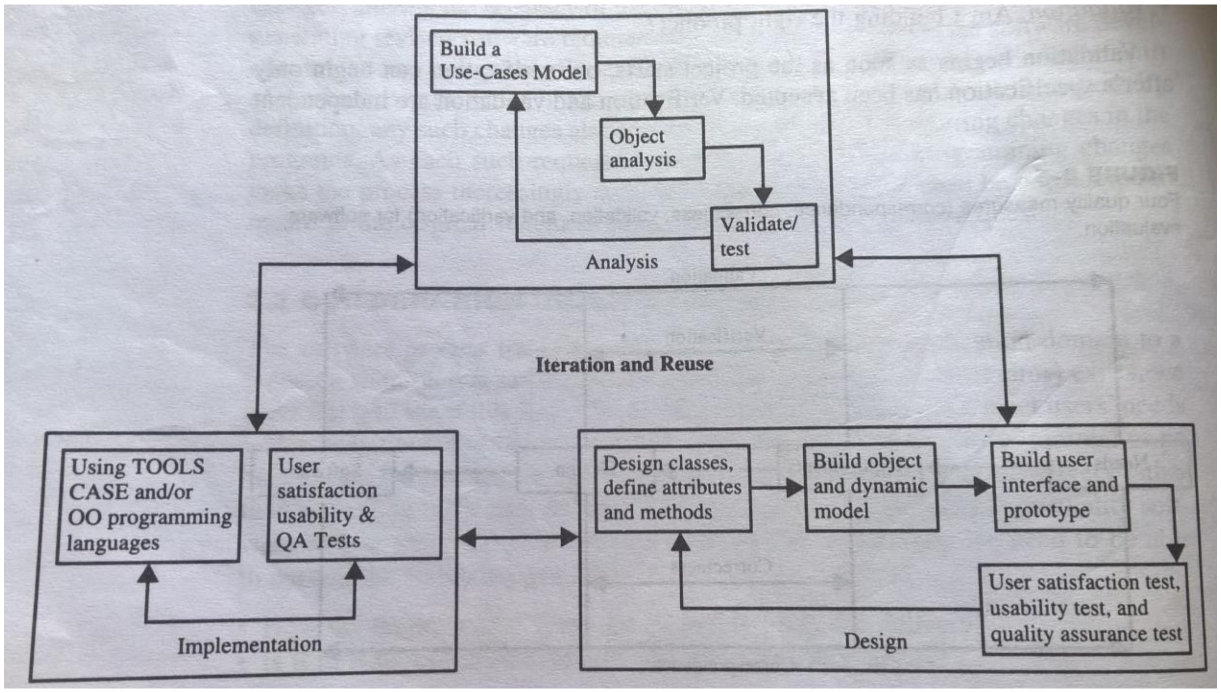

Build a Use-Cases Model
Object analysis
Validate/ test
Analysis
Iteration and Reuse
Using TOOLS CASE and/or OO programming languages
User satisfaction usability & QA Tests
Implementation
Design classes, define attributes and methods
Build object and dynamic model
Build user interface and prototype
User satisfaction test, usability test, and quality assurance test
Design

Fig. 1.22. The object-oriented system development approach

1) Object Oriented Analysis (OOA) – Use case driven

- OOA phase of software development is concerned with

a) Determining the system requirements
b) Identifying classes & their relationship
c) To understand the system requirements, we need to identify the users or actors
d) **"Use Case"** – It is a typical interaction between user and a system that captures users' goals and needs
e) **"Use Case Model"** – The use case model represents the users view of the system or the user needs
f) The process of developing the use cases, like other object-oriented activities are iterative
g) The objects need to have meant only within the content of the application domain.
   e.g., The application domain might be a payroll system and the objects are paycheck, employee, worker, supervisor, office administrator.
h) Documentation
   a. It is another important activity which does not end with object-oriented analysis but should be carried out throughout the system development.
   b. 80-20 rule generally applies for documentation, i.e., 80% of the work canbe done with 20% of the documentation.

2) Object Oriented Design (OOD)

a) The goal of OOD is to design the classes identified during the analysis phase
b) OOA and OOD are distinct disciplines, but they can be interconnected.
c) In OOD, design classes, define attributes and methods then build object and dynamic model.
d) First build the object model based on objects and their relationship then iterate and refine the model
    a. Design and refine classes
    b. Design and refine attributes
    c. Design and refine the methods
    d. Design and refine the structures
    e. Design and refine the associations
e) Guidelines to use in your OOD
    a. Reuse rather than build a new class know the existing classes
    b. Design a large number of simple classes, rather than a small number of complex classes
    c. Design methods
    d. A detailed analysis, what you have proposed. If possible, go-back and refine the classes.

3) Prototyping

a) A prototype is a version of a software product developed in the early stages of the products life cycle for specific, experimental purposes.
b) Prototyping can further define the use-cases, and actually makes use-case modelling much easier.
c) Prototyping provides the developer a means to test and refine the user interface and increase the usability of the system.
d) Prototypes have been categorized in various ways
    a. Horizontal prototype:
        a. It's a simulation of the interface but contains no functionality.
        b. An advantage is quick to implement.
    b. Vertical prototype:
        a. It is a subset of the system features with complete functionality.
        b. An advantage is that the few implemented functions can be tested in great depth.
    c. Analysis prototype:
        a. It is an aid for exploring the problem domain.
        b. The final product will use the concepts exposed by the prototype, not its code.
    d. Domain prototype:

a. It is an aid for the incremental development of the ultimate software solution.
b. It demonstrates the feasibility of the implementation.

4) Implementation: Component Based Development

- Component Based Development (CBD) is an industrial approach to software development.
- Software components are functional units, or building blocks offering a collectionof reusable services.
- A CBD developer can assemble components to construct a complete software system.
- A component-based development is concerned with the implementation and system integration aspects of software development.
- **CASE:** Computer Aided Software Engineering Tools allow their users to rapidly develop information systems.
- **Goal:** The main goal of CASE technology is the automation of the entire systems development life cycle process using a set of integrated software tools, such as modelling, methodology and automatic code generation.
- **RAD:** Rapid Application Development
- RAD is to build a version of an application rapidly to see whether we actually understood the problem.
- It builds the application quickly through tools such as Delphi, Visual Age, Visual Basic.

5) Incremental Testing

- To test an application for bugs and performance.
- After development of applications are finally given to Quality Assurance group for testing the process.

TEXT/ REFERENCE BOOKS:

1. Ali Bahrami, “Object oriented systems development using the unified modelling language”, 1st Edition, McGraw-Hill, 1998.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson, “The Unified Modelling Language User Guide”, 3rd Edition Addison Wesley,2007.
3. John Deacon, "Object Oriented Analysis and Design", 1st Edition, Addison Wesley, 2005.
4. Grady Booch, James Rumbaugh, and Ivar Jacobson, “The Unified Modelling Language User Guide”, 3rd Edition Addison Wesley.
5. John Deacon, “Object Oriented Analysis and Design”, 1st Edition, Addison Wesley.
6. Bernd Oestereich, “Developing Software with UML, Object - Oriented Analysis and Design in Practice”, Addison-Wesley

Questions

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Describe software development methodology?	Knowledge	BTL 1
2.	Write are the orthogonal views of the software?	Create	BTL 6
3.	Define object-oriented system development methodology?	Knowledge	BTL 1
4.	Classify traditional approach and object-oriented approach	Understand	BTL 2
5.	Compare the advantages of object-oriented development?	Understand	BTL 2
6.	Describe the components of the unified approach.	Knowledge	BTL 1
7.	Illustrate the uses of UML?	Analyze	BTL 4
8.	Contrast the methods and the messages	Analyze	BTL 4
9.	Illustrate How are the objects identified in object-oriented system?	Analyze	BTL 4
10.	Develop the software development process?	Create	BTL 6
Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Summarize the overview of object-oriented language systems development in detail.	Understand	BTL 2
2.	Illustrate the various object or class hierarchy in detail.	Analyze	BTL 4
3.	Reframe the object basics in object-oriented development methodology in detail.	Evaluate	BTL 5
4.	Sketch the following: a) Object identity b) Static and dynamic binding	Apply	BTL 3
5.	Illustrate the object-oriented system development life cycle (SDLC): A Use Case Driven Approach in detail.	Analyze	BTL 4
6.	Defend prototyping? How it is useful?	Evaluate	BTL 5
7.	Contrast software verification is can differ from software validation?	Analyze	BTL 4

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – II - Object Oriented Analysis and Design– SBSA1403

OBJECT ORIENTED ANALYSIS AND DESIGN– SBSA1403

COURSE OBJECTIVES:

- To Understand the fundamentals of Object-Oriented System Development
- To understand the object-oriented methodologies.
- To use UML in requirements elicitation and designing.
- To understand concepts of relationships and aggregations.
- To test the software against its requirements specification

UNIT I - INTRODUCTION

Overview of object-oriented language systems development – Object basics hierarchy – Object and identity –Static and dynamic binding – Object oriented SDLC.

UNIT II - OBJECT ORIENTED METHODOLOGIES

Rumbaugh et al.'s technique – Booch, Jacobson Methodologies – Patterns – Framework – Unified approach –UML – UML diagrams – UML dynamic modelling – UML extensibility – UML meta-model.

UNIT III - OBJECT ORIENTED ANALYSIS

Use case model – Object analysis classification – Approaches for identifying classes – Classes responsibilities and collaborators – Identifying object relationships, attributes and methods.

UNIT IV - OBJECT ORIENTED DESIGN

Design process and design axioms – Designing classes – Access Layer – Object storage and object Interoperability – View layer – Designing interface objects.

UNIT V - SOFTWARE QUALITY

Software quality assurance – Testing strategies – Test cases – Test plan – Myers debugging principle – System usability and measuring user satisfaction.

COURSE OUTCOMES

CO1 - Understand the basics object model for System development.
CO2 - Understand the object-Oriented Methodologies
CO3 - Express software design with UML diagrams
CO4 - Understand the concept of Relationships
CO5 - Design software applications using OO concepts.
CO6 -Understand the various testing methodologies for OO software

UNIT II

OBJECT ORIENTED METHODOLOGIES

INTRODUCTION

- Object-oriented methodology is a set of methods, models, and rules for developing systems.
- Modelling is the process of describing an existing or proposed system. It can be used during any phase of the software life cycle.

2.1 RUMBAUGH'S OBJECT MODELING TECHNIQUE

- The object modelling technique (OMT) presented by Jim Rumbaugh and his co-workers describes a method for the analysis, design, and implementation of a system using an object-oriented technique.
- OMT is a fast, intuitive approach for identifying and modelling all the objects making up a system.
- Class, attributes, methods, inheritance & association also can be expressed easily.
- The dynamic behavior of objects can be described using the **OMT dynamic model**.
- A process description and consumer-producer relationships can be expressed using **OMT's functional model**.

OMT consists of four phases, which can be performed iteratively:

1. **Analysis**. The results are objects and dynamic and functional models.
2. **System design**. The results are a structure of the basic architecture of the system along with high-level strategy decisions.
3. **Object design**. This phase produces a design document, consisting of detailedobjects static, dynamic, and functional models.
4. **Implementation.** This activity produces reusable, extendible, and robust code.

OMT separates modeling into three different parts:

1. An **object model**, presented by the object model and the data dictionary.
2. A **dynamic model**, presented by the state diagrams and event flow diagrams.
3. A **functional model**, presented by data flow and constraints.

THE OBJECT MODEL

- The object model describes the structure of objects in a system: their identity, relationships to other objects, attributes, and operations.
- The object diagram contains classes interconnected by association lines.
- Each class represents a set of individual objects.
- The association lines establish relationships among the classes.
- Each association line represents a set of links from the objects of one class to the objects of another class.

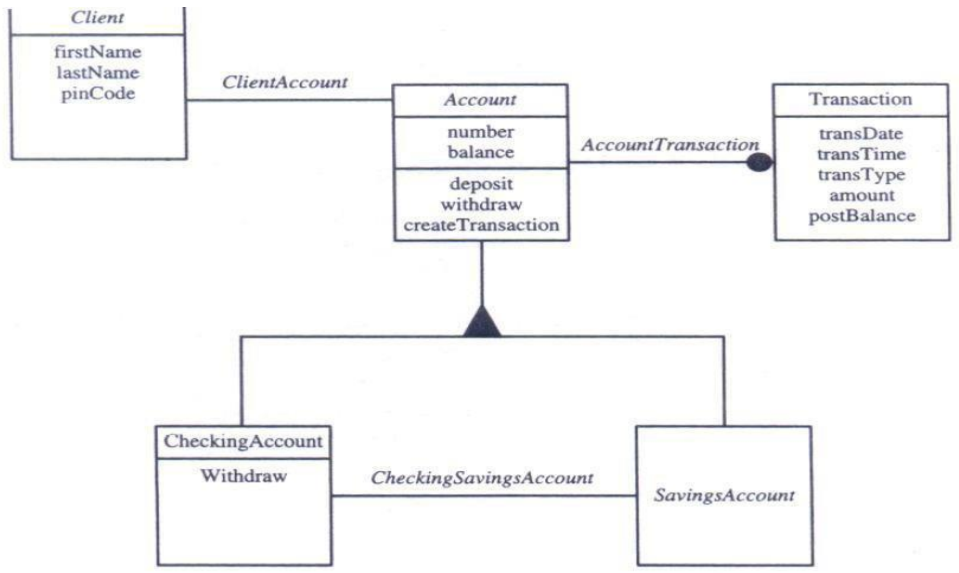


Fig. 2.1. OMT object model of a bank system

Boxes- represents classes, Filled Triangle – represents Specialization, Association between account and transaction represents one to many, Filled Circle – represents many (zero or more). Association between Client and Account represents one to one.

THE OMT DYNAMIC MODEL

- OMT provides a detailed and comprehensive dynamic model, in addition to letting you depict states, transitions, events, and actions.
- The OMT state transition diagram is a network of states and events (see Fig. 2).
- Each state receives one or more events, at which time it makes the transition to the next state.
- The next state depends on the current state as well as the events.

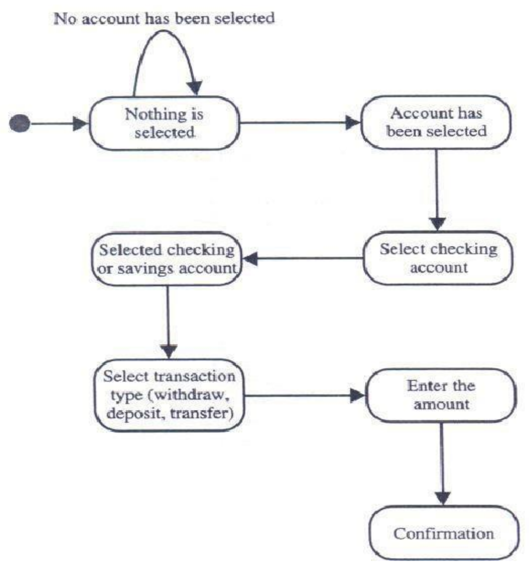

No account has been selected
Nothing is selected
Account has been selected
Select checking account
Selected checking or savings account
Select transaction type (withdraw, deposit, transfer)
Enter the amount
Confirmation

Fig. 2.2. State transition diagram for the bank application user interface. The roundboxes represent states and the arrows represent transitions.

THE OMT FUNCTIONAL MODEL

- The OMT data flow diagram (DFD) shows the flow of data between different processes in a business.

Data flow diagrams use four primary symbols:

1. ⬭ The **process** is any function being performed; for example, verify Password or PIN in the ATM system (see Fig .3).
2. → - The **data flow** shows the direction of data element movement; for example, PIN code.
3. ═ - The **data store** is a location where data are stored; for example, account is a data store in the ATM example.
4. ▭ - An **external entity** is a source or destination of a data element; for example, the ATM card reader.

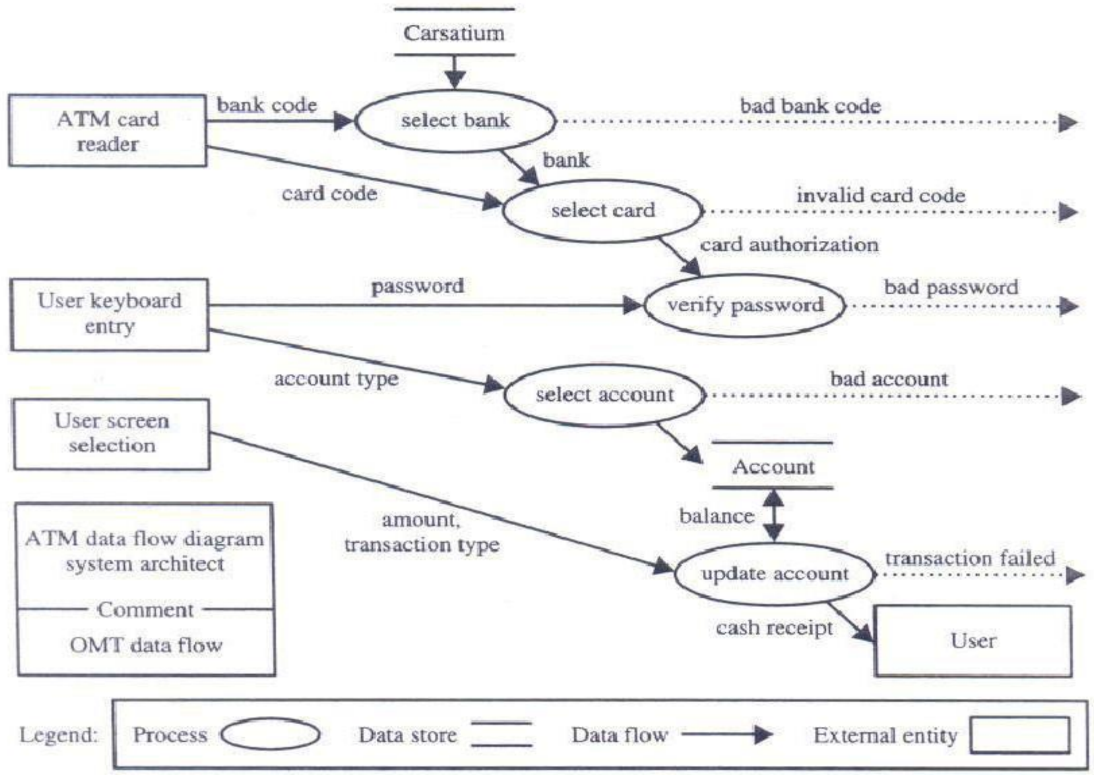


Fig 2.3. OMT DFD of the ATM system.

2.2 BOOCH METHODOLOGY

- The Booch methodology is a widely used object-oriented method that helps you design your system using the object paradigm.
- The Booch method consists of the following diagrams:
  - Class diagrams
  - Object diagrams
  - State transition diagrams
  - Module diagrams
  - Process diagrams
  - Interaction diagrams
- The Booch methodology prescribes a **macro development process** and **a micro development process**.

THE MACRO DEVELOPMENT PROCESS

- The macro process serves as a controlling framework for the micro process.
- The primary concern of the macro process is technical management of the system.

The macro development process consists of the following steps:

1. **Conceptualization.** During conceptualization,
   - establish the core requirements of the system.
   - establish a set of goals and
   - develop a prototype to prove the concept.
2. **Analysis and development of the model.**
   - use the **class diagram** to describe the roles and responsibilities objects
   - **object diagram** to describe the desired behavior of the system in terms of scenarios
3. **Design or create the system architecture.** In the design phase,
   - use the **class diagram** to decide what classes exist and how they relate to each other.
   - use the **object diagram** to decide what mechanisms are used to regulate how objects collaborate.
   - use the **module diagram** to map out where each class and object should be declared.
   - use the **process diagram** to determine to whichprocessor to allocate a process.
4. **Evolution or implementation.**
   - refine the system through many iterations.
   - Produce a stream of software implementations (or executable releases), each of which is a refinement of the prior one.

5. **Maintenance.** Make localized changes to the system to add new requirements and eliminate bugs.

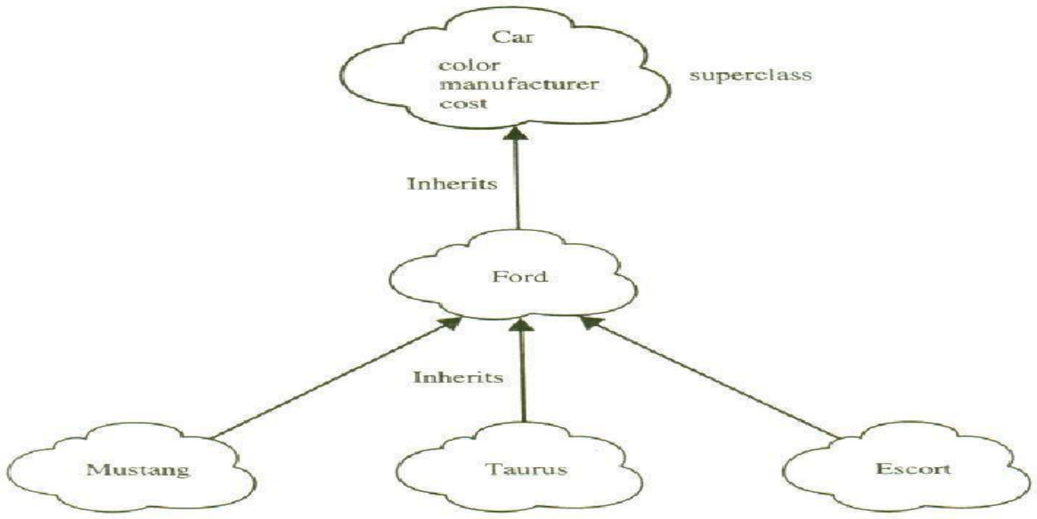


Fig. 2.4. Object modeling using Booch notation

The arrows represent specialization; for example, the class Taurus is subclass ofthe class Ford.

THE MICRO DEVELOPMENT PROCESS

- Each macro development process has its own micro development processes.
- The micro process is a description of the day-to-day activities by a single or small group of software developers.

The micro development process consists of the following steps:

1. Identify classes and objects.
2. Identify class and object semantics.
3. Identify class and object relationships.
4. Identify class and object interfaces and implementation**.**

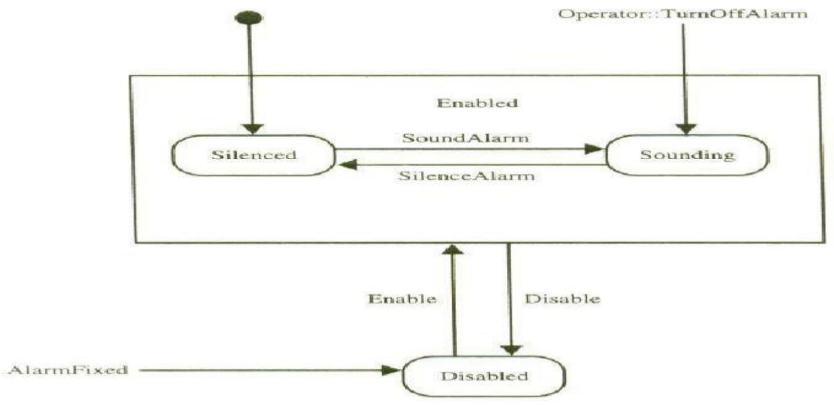


Fig. 2.5. An alarm class state transition diagram with Booch notation.

This diagram can capture the state of a class based on a stimulus. For example, a stimulus causes the class to perform some processing, followed by a transition to anotherstate. In this case, the alarm silenced state can be changed to alarm sounding state and vice versa.

2.3 THE JACOBSON METHODOLOGIES

- The Jacobson et al. methodologies (e.g., object-oriented Business Engineering (OOBE), object-oriented Software Engineering (OOSE), and Objectory) cover the entire life cycle and stress traceability between the different phases, both forward and backward.
- This traceability enables reuse of analysis and design work, possibly much bigger factors in the reduction of development time than reuse of code.
- At the heart of their methodologies is the use-case concept, which evolved with Objectory (Object Factory for Software Development).

USE CASES

- Use cases are scenarios for understanding system requirements.
- A use case is an interaction between users and a system.
- The use-case model captures
  - the goal of the user and
  - the responsibility of the system to its users

Some uses of a library. As you can see, these are external views of the library system from an actor such as a member. The simpler the use case, the more effective it will be. It is unwise to capture all of the details right at the start; you can do that later.

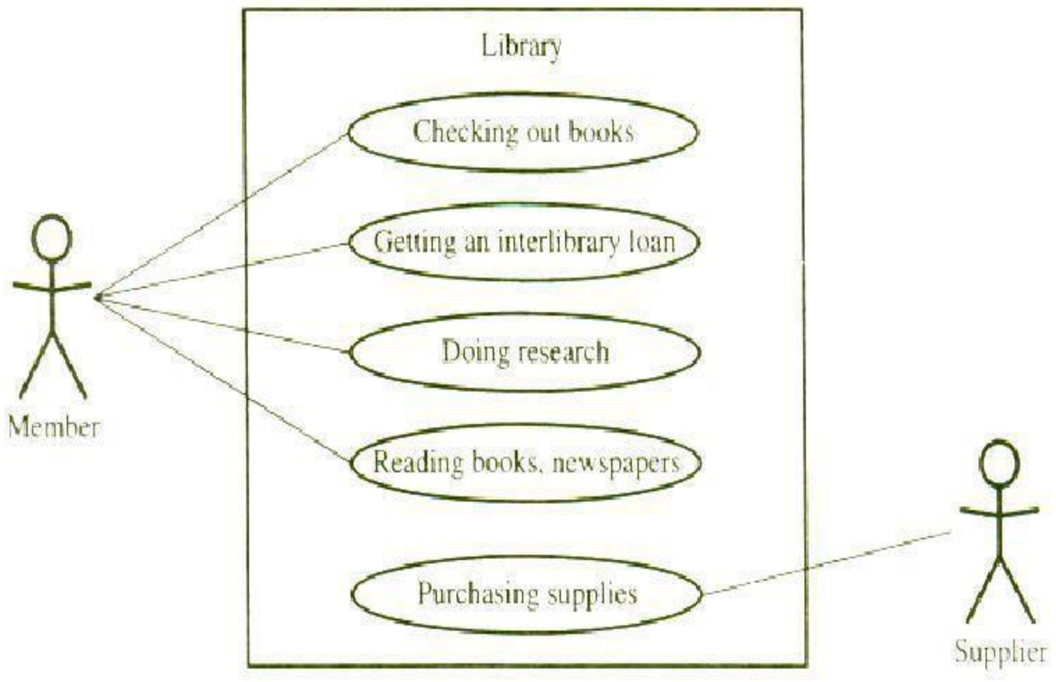


Fig. 2.6. Some Uses of a Library.

The use case description must contain

- How and when the use case begins and ends.
- The interaction between the use case and its actors, including when the interaction occurs and what is exchanged.

- How and when the use case will need data stored in the system or will store data in the system.
  Exceptions to the flow of events.
- How and when concepts of the problem domain are handled.

OBJECT-ORIENTED SOFTWARE ENGINEERING: OBJECTORY

- Object-oriented software engineering (OOSE), also called **Objectory, is a method** of object-oriented development with the specific aim to fit the development of large, real-time systems.

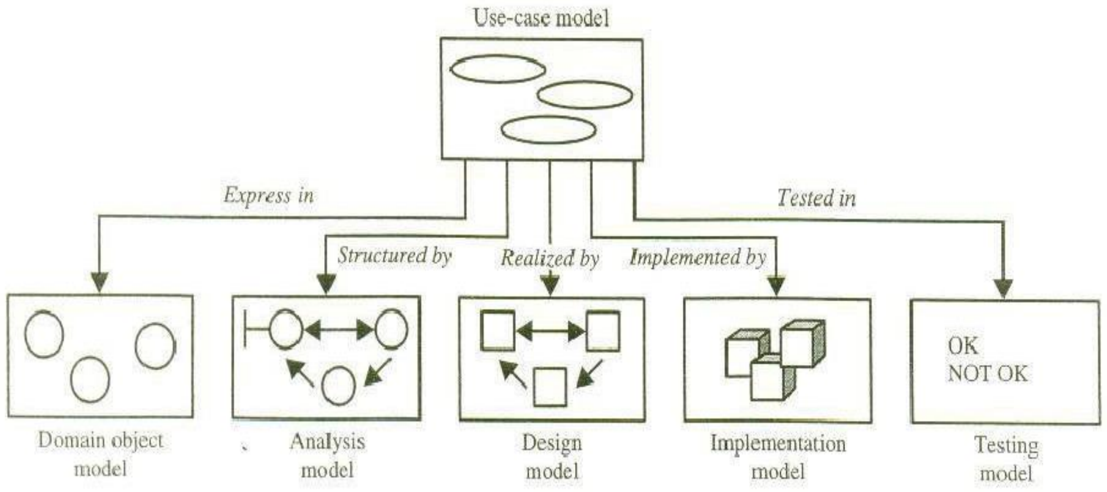


Fig. 2.7. The use case model is considered in every model and phase.

Objectory is built around several different models:

- **Use case-model.** The use-case model defines the outside (actors) and inside (usecase) of the system's behavior.
- **Domain object model.** The objects of the "real" world are mapped into the domain object model.
- **Analysis object model.** The analysis object model presents how the source code (implementation) should be carried out and written.
- **Implementation model.** The implementation model represents the implementation of the system.
- **Test model.** The test model constitutes the test plans, specifications, and reports.

OBJECT-ORIENTED BUSINESS ENGINEERING

- Object-oriented business engineering (OOBE) is object modelling at the enterprise level.
  - Analysis phase
  - Design and implementation Phase
  - Testing Phase

1. **Analysis phase.** It defines the system to be built in terms of
   a. problem-domain object model,

b. the requirements model, and
c. the analysis models.

2. **Design and implementation phases.** This includes **factors** such as
   a. Database Management System (DBMS),
   b. distribution of process,
   c. constraints due to the programming language,
   d. available component libraries, and
   e. incorporation of graphical user interface tools
3. **Testing phase.**
   Jacobson describes several testing levels and techniques.
   The levels include
   -unit testing,
   - integration testing, and
   -system testing.

2.4 PATTERNS

Definition:

A pattern is [an] instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces.

- A pattern involves a general description of a solution to a recurring problem bundle with various goals and constraints.

proto-pattern.

- A "pattern in waiting," which is not yet known to recur, sometimes is called a **proto-pattern**.

A good pattern will do the following:

- **It solves a problem**. Patterns capture solutions, not just abstract principles orstrategies.
- **It is a proven concept**. Patterns capture solutions with a track record, not theoriesor speculation.
- **The solution is not obvious**. The best patterns generate a solution to a problem indirectly-a necessary approach for the most difficult problems of design.
- **It describes a relationship**. Patterns do not just describe modules, but describe deeper system structures and mechanisms.
- **The pattern has a significant human component**. All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

Most pattern encompass all aspects of S/W Engg including

- Development organization
- The s/w development process
- Project Planning
- Requirement engineering
- s/w configuration management

Generative and Non generative Patterns

- **Generative patterns** describe how to generate something

- **Nongenerative patterns** are static and passive:
  - They describe how to reproduce them.

Patterns Template

- Every pattern must be expressed "in the form of a rule [template] which establishes a relationship between a context and a configuration

Essential components should be clearly recognizable on reading a pattern:

- **Name.** A meaningful name.
- **Problem.** A statement of the problem
- **Context.** The preconditions
- **Forces.** how they interact or conflict with one another and with the goals
- **Solution.** how to realize the desired outcome?
- **Examples.** how the pattern is applied to and transforms that context. One or more sample applications
- **Resulting context.** The state of the system after the pattern has been applied, including the consequences (both good and bad) of applying the pattern, and other problems and patterns that may arise from the new context.
- **Rationale.** how it works, why it works, and why it is "good."
- **Related patterns.** Related patterns often share common forces.
- **Known uses.** The known occurrences

ANTIPATTERNS

- A pattern represents a "best practice," whereas an antipattern represents "worst practice" or a "lesson learned."

Anti-patterns come in two varieties:

- Those describing a bad solution to a problem that resulted in a bad situation.
- Those describing how to get out of a bad situation and how to proceed from there toa good solution.

Anti-patterns are valuable because often it is just as important to see and understand badsolutions as to see and understand good ones.

Capturing Patterns

- Writing good patterns is very difficult
- A pattern should help its users comprehend existing systems, customize systems tofit user needs, and construct new systems.
- The process of looking for patterns to document is called **pattern mining (or sometimes reverse architecting).**

2.5. FRAMEWORKS

A framework is a way of presenting a generic solution to a problem that can be applied to all levels in a development.

A definition of an object-oriented software framework is given by Gamma:

- A framework is a set of cooperating classes that make up a reusable design for a specific class of software.
- A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations.
- A developer customizes a framework to a particular application by subclassing and composing instances of framework classes.
- The framework captures the design decisions that are common to its application domain.
- A single framework typically encompasses several design patterns.

In fact, **a framework can be viewed as the implementation of a system of designpatterns.**

Differences between frameworks and design patterns:

*. A framework is executable software, whereas design patterns represent knowledge and experience about software.

*. Frameworks are of a physical nature, while patterns are of a logical nature.

*. Frameworks are the physical realization of one or more software pattern solution; patternsare the instructions for how to implement those solution.

Gamma et al. describe **the major differences between design patterns andframeworks as follows:**

Design patterns are more abstract than frameworks.
Design patterns are smaller architectural elements than frameworks.

Design patterns are less specialized than frameworks.

2.6 THE UNIFIED APPROACH

- The unified approach (UA) used to describe, model, and document the software development process.
- The idea behind the UA is to combine the best practices, processes, methodologies, and guidelines along with UML notations and diagrams for better understanding object-oriented concepts and system development.
- The unified approach to software development revolves around the following processes and concepts (see Fig.8). The processes are:
  - □ Use-case driven development
  - □ Object-oriented analysis
  - □ Object-oriented design
  - □ Incremental development and prototyping
  - □ Continuous testing

the methods and technology employed include

1. Unified modeling language used for modeling.
2. Layered approach.

3. Repository for object-oriented system development patterns and frameworks.
4. Component based development.

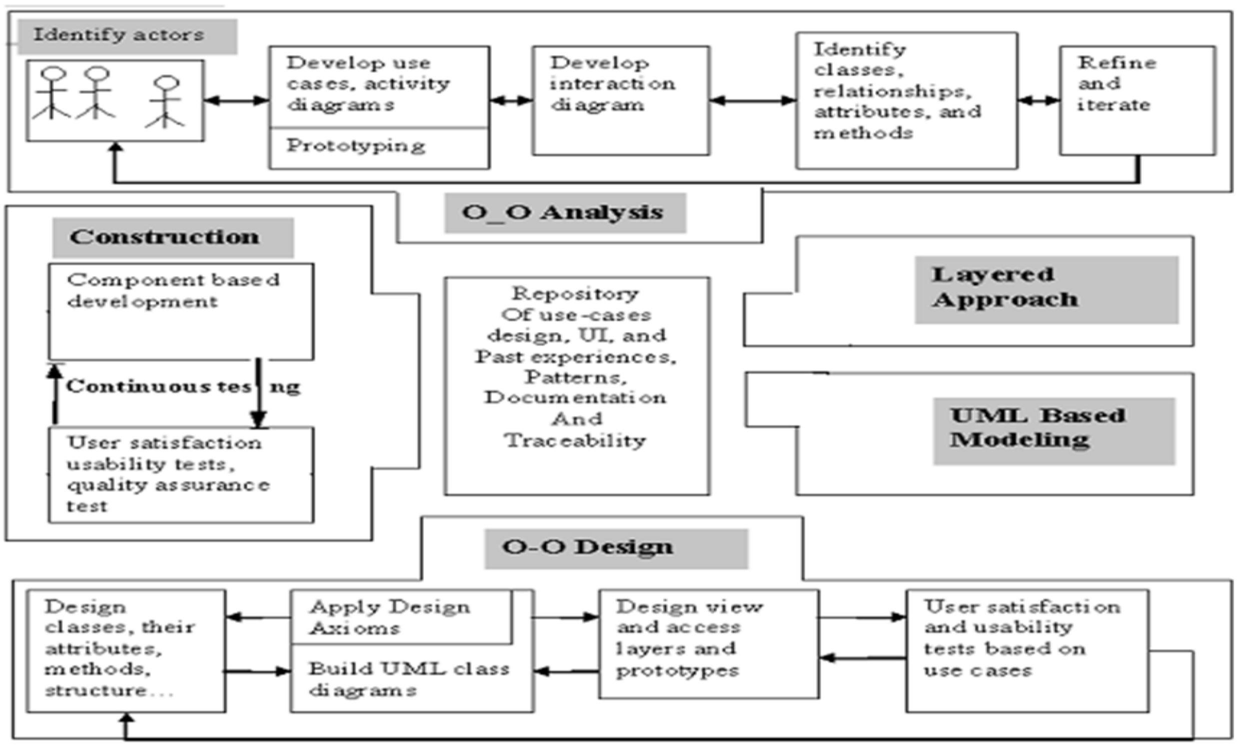

Identify actors
Develop use cases, activity diagrams
Prototyping
Develop interaction diagram
Identify classes, relationships, attributes, and methods
Refine and iterate
O_O Analysis
Construction
Component based development
Continuous testing
User satisfaction usability tests, quality assurance test
Repository Of use-cases design, UI, and Past experiences, Patterns, Documentation And Traceability
Layered Approach
UML Based Modeling
O-O Design
Design classes, their attributes, methods, structure...
Apply Design Axioms
Build UML class diagrams
Design view and access layers and prototypes
User satisfaction and usability tests based on use cases

Fig. 2.8. The Process and components of the unified approach

OBJECT-ORIENTED ANALYSIS

- Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirements.
- The goal of object-oriented analysis is to first understand the domain of the problem and the system's responsibilities by understanding how the users use or will use the system.
- This is accomplished by constructing several models of the system.
- These models concentrate on describing what the system does rather than how it doesit. Separating the behavior of a system from the way it is implemented requires viewingthe system from the user's perspective rather than that of the machine.

OOA Process consists of the following Steps:

1. Identify the Actors.
2. Develop a simple business process model using UML Activity diagram.
3. Develop the Use Case.
4. Develop interaction diagrams.
5. Identify classes.

OBJECT-ORIENTED DESIGN

OOD Process consists of:

- Designing classes, their attributes, methods, associations, structures and protocols, apply design axioms
- Design the Access Layer
- Design and prototype User interface
- User Satisfaction and Usability Tests based on the Usage/Use Cases
- Iterate and refine the design

ITERATIVE DEVELOPMENT AND CONTINUOUS TESTING

You must iterate and reiterate until, eventually, you are satisfied with the system.

1. **MODELING BASED ON THE UNIFIED MODELING LANGUAGE**

The UA uses the UML to describe and model the analysis and design phases of system development.

The UA Proposed Repository

In a repository that allows the maximum reuse of previous experience and previously defined objects, patterns, frameworks, and user interfaces in an easily accessible manner with a completely available and easily utilized format.

The **advantage of repositories** is that for reuse.

2. **The Layered Approach to Software Development**

Most systems developed with today's CASE tools or client-server application development environments tend to lean toward what is known **as two-layered architecture**: interface and data (see Fig). In a two-layered system, user interface screens are tied to the data through routines that sit directly behind the screens

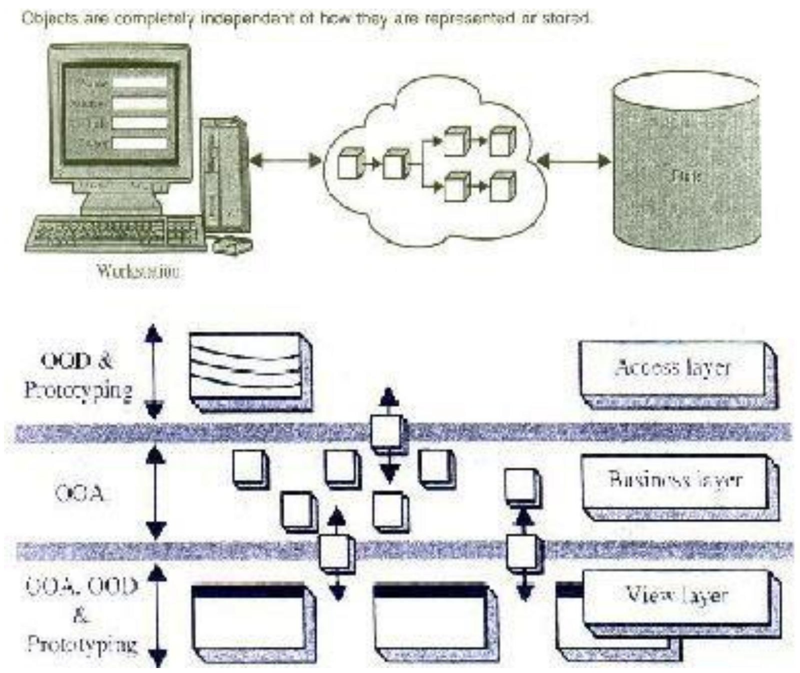


Fig. 2.9. The Layered Approach

The **three-layered approach** consists of

1. a view or user interface layer,
2. a business layer, and
3. an access layers

The Business Layer The business layer contains all the objects that represent the business (both data and behavior). This is where the real objects such as Order, Customer, Line item, Inventory, and Invoice exist. Most modem object-oriented analysis and design methodologies are generated toward identifying these kinds of objects.

The responsibilities of the business layer are very straightforward: Model the objects of the business and how they interact to accomplish the business processes. When creating the business layer, however, it is important to keep in mind a couple of things. These objectsshould not be responsible for the following:

Displaying details. Business objects should have no special knowledge of how they are being displayed and by whom. They are designed to be independent of any particular interface, so the details of how to display an object should exist in the interface (view) layer of the object displaying it.

Data access details. Business objects also should have no special knowledge of "where they come from." It does not matter to the business model whether the data are stored and retrieved via SQL or file I/O. The business objects need to know only to whom to talk about being stored or retrieved. The business objects are modeled during the object- oriented analysis. A business model captures the static and dynamic relationships among a collection of business objects. Static relationships include object associations and aggregations. For example, a customer could have more than one account or an order couldbe aggregated from one or more-line items. Dynamic relationships show how the business objects interact to perform tasks. For example, an order interacts with inventory to determineproduct availability. An individual business object can appear in different business models. Business models also incorporate control objects that direct their processes. The business objects are identified during the object oriented analysis. Use cases can provide a wonderful tool to capture business objects.

The User Interface (View) Layer: The user interface layer consists of objects with which the user interacts as well as the objects needed to manage or control the interface. The userinterface layer also is called the view layer. This layer typically is responsible for two major aspects of the applications:

Responding to user interaction. The user interface layer objects must be designed to translate actions by the user, such as clicking on a button or selecting. from a menu, into an appropriate response. That response may be to open or close another interface or to send amessage down into the business layer to start some business process; remember, the business logic does not exist here, just the knowledge of which message to send to
which business object.

Displaying business objects. This layer must paint the best possible picture of the business objects for the user. In one interface, this may mean entry fields and list boxes to display an order and its items. In another, it may be a graph of the total price of a customer'sorders.

The Access layer: The access layer contains objects that know how to communicate with the place where the data actually reside, whether it be a relational database mainframe, internet or file. The Access layer has 2 major responsibilities.

1. **Translate request:** The access layer must be able to translate any data-relatedrequests from the business layer into the appropriate protocol for data access.
2. **Translate results:** the access Layer also must be able to translate the data retrieved back into the appropriate business objects and pass those objects back up into the business layer.

   Access objects are identified during object-oriented design.

2.9 UNIFIED MODELING LANGUAGE

A **model** is an abstract representation of a system, constructed to understand the system prior to building or modifying it. Most of the modeling techniques involve graphical languages.

Modeling frequently is used during many of the phases of the software life cycle, such as analysis, design, and implementation. For example, Objectory is built around several different models:

Use-case model. The use-case model defines the outside (actors) and inside (usecase) of the system's behavior.

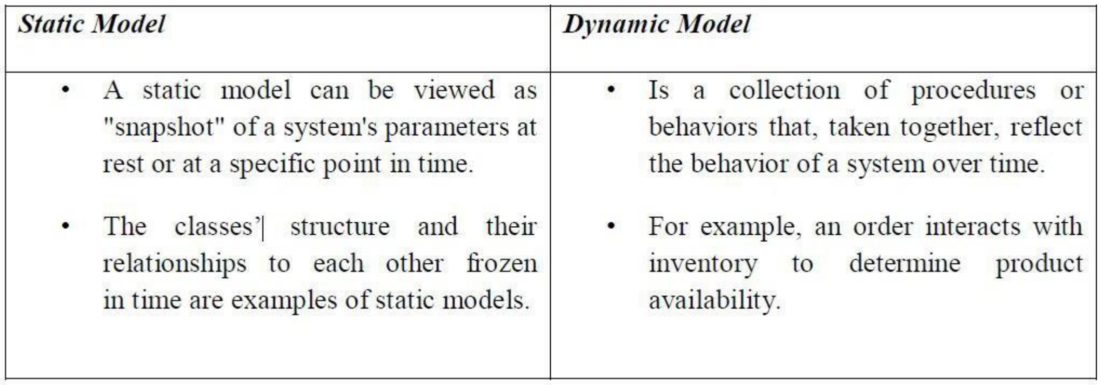
Domain object model. Objects of the "real" world are mapped into the domain object model.

Analysis object model. The analysis object model presents how the source code(i.e., the implementation) should be carried out and written.

Implementation model. The implementation model represents the implementationof the system.

Test model. The test model constitutes the test plans, specifications, and reports. Modeling is an iterative process.

Static or Dynamic Models



Static Model	*Dynamic Model*
• A static model can be viewed as "snapshot" of a system's parameters at rest or at a specific point in time. • The classes'| structure and their relationships to each other frozen in time are examples of static models.	• Is a collection of procedures or behaviors that, taken together, reflect the behavior of a system over time. • For example, an order interacts with inventory to determine product availability.

WHY MODELING?

Building a model for a software system prior to its construction is as essential as having a blueprint for building a large building. Good models are essential forcommunication among project teams. As the complexity of systems increases, so does the importance of good modeling techniques. Many other factors add to a project's success, but having a rigorous modeling language is essential. A modeling language must include Model elements- fundamental modeling concepts and semantics.

Turban cites the following advantages of modeling:

1. Models make it easier to express complex ideas. For example, an architect builds a model to communicate ideas more easily to clients.
2. The main reason for modelling is the reduction of complexity. Models reduce complexity by separating those aspects that are unimportant from those that are important. Therefore, it makes complex situations easier to understand.
3. Models enhance and reinforce learning and training.
4. The cost of the modelling analysis is much lower than the cost of similar experimentation conducted with a real system.
5. Manipulation of the model (changing variables) is much easier than manipulatinga real system.

key ideas regarding modeling:

- A model is rarely correct on the first try.
- Always seek the advice and criticism of others. You can improve amodel by reconciling different perspectives.
- Avoid excess model revisions, as they can distort the essence of your model.

What Is the UML?

The unified modeling language is a language for specifying, constructing, visualizing, and documenting the software system and its components. The UML is a graphical language with sets of rules and semantics. The rules and semantics of a model are expressed in English, in a form known as object constraint language (OCL). OCL is a specification language that uses simple logic for specifying the properties of a system.

The UML is not intended to be a visual programming language in the sense of having all the necessary visual and semantic support to replace programming languages. However,the UML does have a tight mapping to a family of object-oriented languages, so that you canget the best of both worlds.

What it is/isn’t? Is NOT

- A process
- A formalism

Is

- A way to describe your software

• more precise than English
• less detailed than code

What is UML Used For?

- Trace external interactions with the software
- Plan the internal behavior of the application
- Study the software structure
- View the system architecture
- Trace behavior down to physical components

The primary goals in the design of the UML were as follows:

1. Provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts.
7. Integrate best practices and methodologies.

UML DIAGRAMS
The UML defines nine graphical diagrams:

1. **Class diagram (static)**
2. **Use-case diagram**
3. **Behavior diagrams (dynamic):**
   – 3.1. Interaction diagram:
      • **3.1.1. Sequence diagram**
      • **3.1.2. Collaboration diagram**
   – 3.2. State chart diagram

   – 3.3. Activity diagram

4. **Implementation diagram:**
   4.1. Component diagram
   4.2. Deployment diagram

UML CLASS DIAGRAM

The UML **class diagram**, also referred to as **object modelling**, is the main static analysis diagram. These diagrams show the static structure of the model.

A class diagram is a collection of static modeling elements, such as classes and their relationships, connected as a graph to each other and to their contents; for example, the things that exist (such as classes), their internal structures, and their relationships to other classes.

Class diagrams do not show temporal information, which is required in dynamic modeling.

- A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them.
- A graphical representation of a static view on declarative static elements.
- A central modelling technique that runs through nearly all object-oriented methods.
- The richest notation in UML.
- A class diagram shows the existence of classes and their relationships in the logical view of a system

Class Notation: Static Structure

A class is drawn as a rectangle with three components separated by horizontal lines. The top name compartment holds the class name, other general properties of the class, such as attributes, are in the middle compartment, and the bottom compartment holds a list of operations

Either or both the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment if a compartment is suppressed; no inference can be drawn about the presence or absence of elements in it.

The class name and other properties should be displayed in up to three sections. A stylistic convention of UML is to use an italic font for abstract classes and a normal (roman) font for concrete classes.

Essential Elements of a UML Class Diagram

– Class

– Attributes

– Operations

– Relationships
    - Associations
    - Generalization

– Dependency

– Realization

Constraint Rules and Notes

A class is the description of a set of objects having similar attributes, operations, relationships and behavior.

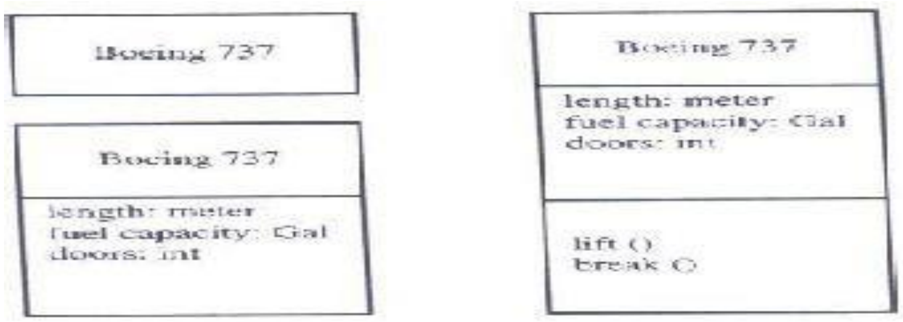


Fig. 2.12. In class notation, either or both the attributes and operation compartmentsmay be suppressed.

Attributes

– Classes have attributes that describe the characteristics of their objects.

– Attributes are atomic entities with no responsibilities.

– Attribute syntax (partial):
o [visibility] name [: type] [= default Value]
– Class scope attributes are underlined

Visibility

• Visibility describes whether an attribute or operation is visible and can be referenced from classes other than the one in which they are defined.

• Language dependent
o Means different things in different languages

• UML provides four visibility abbreviations: + (public) – (private) # (protected) ~(package)

Object Diagram

A static object diagram is an instance of a class diagram. It shows a snapshot of the detailed state of the system at a point in time. Notation is the same for an object diagram and a class diagram. Class diagrams can contain objects, so a class diagram with objects and no classes is an object diagram.

UML modeling elements in class diagrams

• Classes and their structure, association, aggregation, dependency, and inheritance relationships

• Multiplicity and navigation indicators, etc.

Class Interface Notation

Class interface notation is used to describe the externally visible behavior of a class; for example, an operation with public visibility. Identifying class interfaces is a design activity of object-oriented system development.

The UML notation for an interface is a small circle with the name of the interface connected to the class. A class that requires the operations in the interface may be attachedto the circle by a dashed arrow. The dependent class is not required to actually use all of the operations.

For example, a Person object may need to interact with the BankAccount object to get the Balance; this relationship is depicted in Fig13. with UML class interface notation.

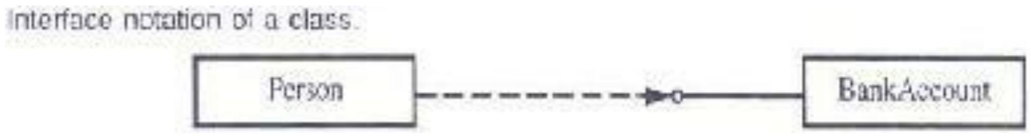


Fig. 2.13. Interface notation of a Class

Binary Association Notation

A binary association is drawn as a solid path connecting two classes, or both ends may be connected to the same class. An association may have an association name. The association name may have an optional black triangle in it, the point of the triangle indicatingthe direction in which to read the name. The end of an association, where it connects to a class, is called the **association role (**see Fig. 14).

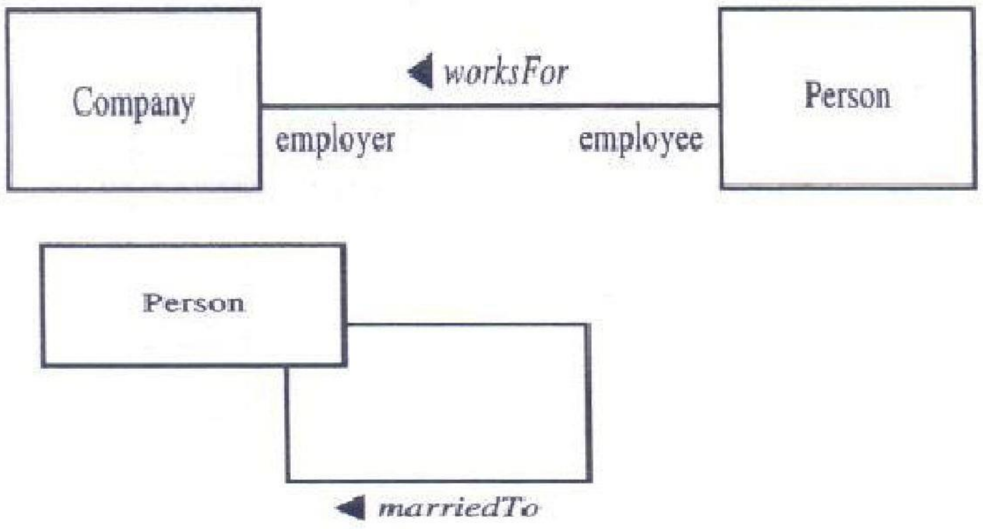


Fig. 2.14. Association Notation.

Association Role

A simple association- **binary association**-is drawn as a solid line connecting two class symbols. The end of an association, where it connects to a class, shows the association role. The role is part of the association, not part of the class. Each association has two or more roles to which it is connected.

In above Fig14. the association worksfor connects two roles, employee and employer. A Person is an employee of a Company and a Company is an employer of a Person.

The UML uses the term association navigation or navigability to specify a roleaffiliated with each end of an association relationship. An arrow may be attached to the end of the path to indicate that navigation is supported in the direction of the class pointed to. An arrow may be attached to neither, one, or both ends of the path. In particular, arrows could be shown whenever navigation is supported in a given direction.

In the UML, association is represented by an open arrow, as represented in Fig.15. Navigability is visually distinguished from inheritance, which is denoted by an unfilled arrowhead symbol near the superclass.

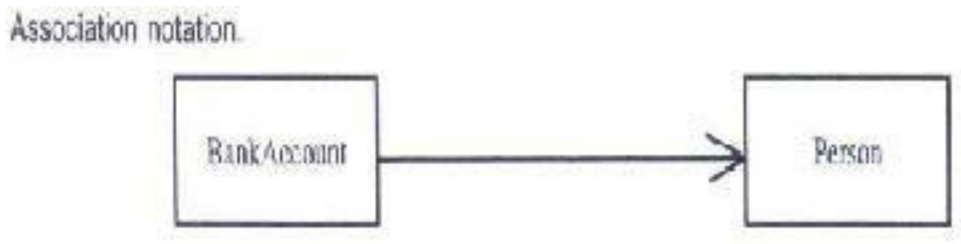


Fig. 2.15. Association Notation

In this example, the association is navigable in only one direction, from the BankAccount to Person, but not the reverse. This might indicate a design decision, but it also might indicate an analysis decision, that the Person class is frozen and cannot be extended to know about the BankAccount class, but the BankAccount class can know about the Person class.

Qualifier

A **qualifier** is an association attribute. For example, a person object may be associated to a Bank object. An attribute of this association is the account#. The account# isthe qualifier of this association. (Fig 16)

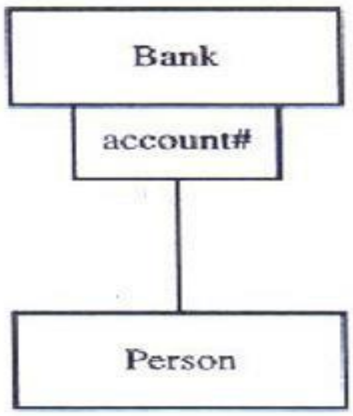


Fig. 2.16. Association Qualifier.

A qualifier is shown as a small rectangle attached to the end of an association path, between the final path segment and the symbol of the class to which it connects. The qualifier rectangle is part of the association path, not part of the class. The qualifier rectangle usually is smaller than the attached class rectangle (see above Fig).

Multiplicity

Multiplicity specifies the range of allowable associated classes. It is given for roles within associations, parts within compositions, repetitions, and other purposes. A multiplicity specification is shown as a text string comprising a period-separated sequence of integer intervals, where an interval represents a range of integers in this format (see Fig 17):

lower bound. upper bound.

The terms lower bound and upper bound are integer values, specifying the range of integers including the lower bound to the upper bound. The star character (*) may be used for the upper bound, denoting an unlimited upper bound. If a single integer value is specified, then the integer range contains the single values.

For example, 0..10..*

1..3, 7..10, 15, 19..*

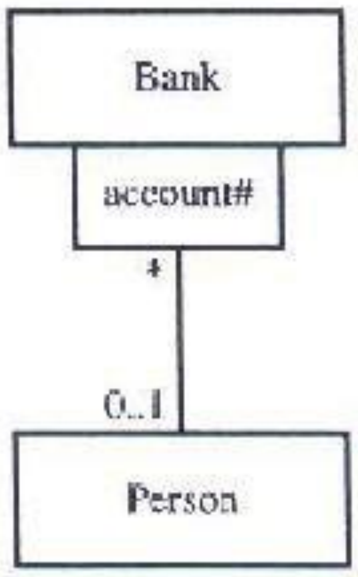


Fig. 2.17. Association Qualifier and its multiplicity.

OR Association

An **OR association** indicates a situation in which only one of several potential associations may be instantiated at one time for any single object. This is shown as a dashed line connecting two or more associations, all of which must have a class in common,with the constraint string {or} labeling the dashed line (see Fig 18). In other words, any instance of the class may participate in, at most, one of the associations at one time.

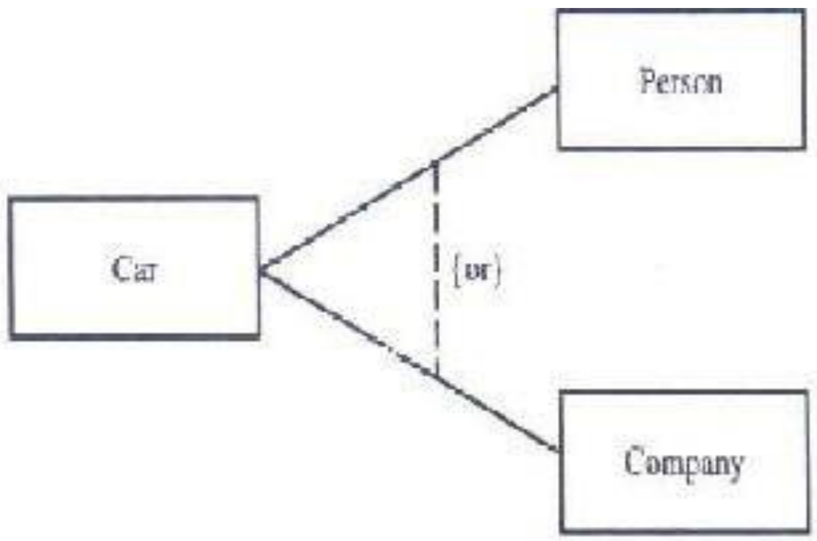


Fig. 2.18. An OR association notation. A car may associate with a person or a company.

Association Class

An **association class** is an association that also has class properties. An association class is shown as a class symbol attached by a dashed line to an association path. The name in the class symbol and the name string attached to the association pathare the same (see Fig 19). The name can be shown on the path or the class symbol or both.If an ssociation class has attributes but no operations or other associations, then the name may be displayed on the association path and omitted from the association class to emphasize its "association nature." If it has operations and attributes, then the name may beomitted from the path and placed in the class rectangle to emphasize its "class nature."

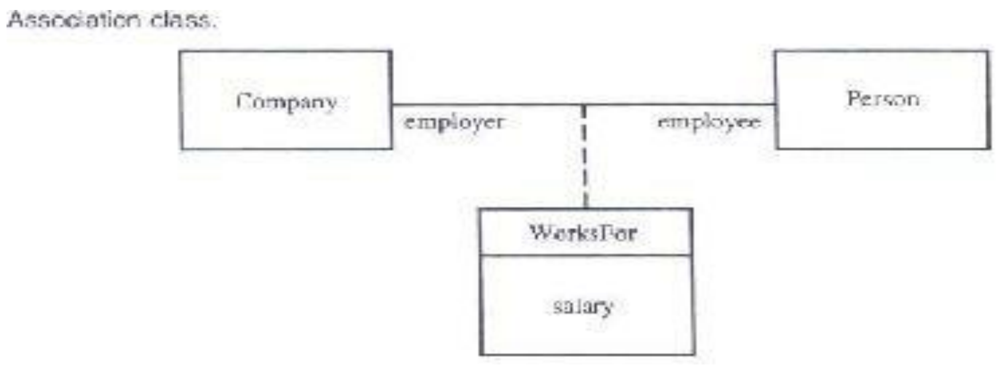


Fig. 2.19. Association Class

N-Ary Association

An n-ary association is an association among more than two classes. Since n-ary association is more difficult to understand, it is better to convert an n-ary association to binary association.

An n-ary association is shown as a large diamond with a path from the diamond to each participant class. The name of the association (if any) is shown near the diamond. The role attachment may appear on each path as with a binary association. Multiplicity may be indicated; however, qualifiers and aggregation are not permitted. An association class symbol may be attached to the diamond by a dashed line, indicating an n-ary association that has attributes, operation, or associations. The example depicted in Fig 20 shows the grade book of a class in each semester.

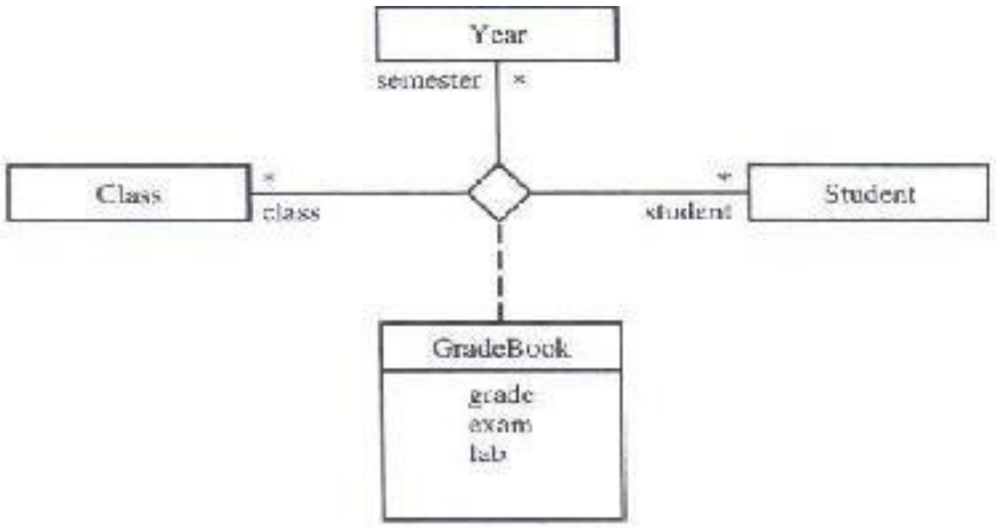


Fig. 2.20. An n-ary (ternary) association that shows association among class, year, and student classes. The association class GradeBook which contains the attributes of the associations such as grade, exam, and lab.

Aggregation and Composition (a.part.of)

Aggregations a form of association. A hollow diamond is attached to the end of the path to indicate aggregation. However, the diamond may not be attached to both ends of a line, and it need not be presented at all (see Fig 21).

Composition, also known as the **apart-of**, is a form of aggregation with strong ownership to represent the component of a complex object. Composition also is referred to as **a part-whole relationship**. The UML notation for composition is a solid diamond at the end of a path. Alternatively, the UML provides a graphically nested form that, in many cases, is more convenient for showing composition (see Fig 22).

Parts with multiplicity greater than one may be created after the aggregate itself but, once created, they live and die with it. Such parts can also be explicitly removed before the death of the aggregate.

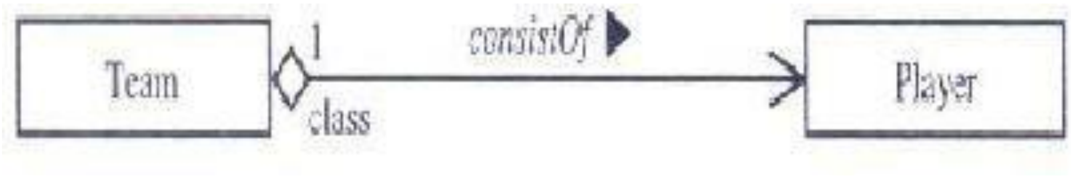


Fig. 2.21. Association Path

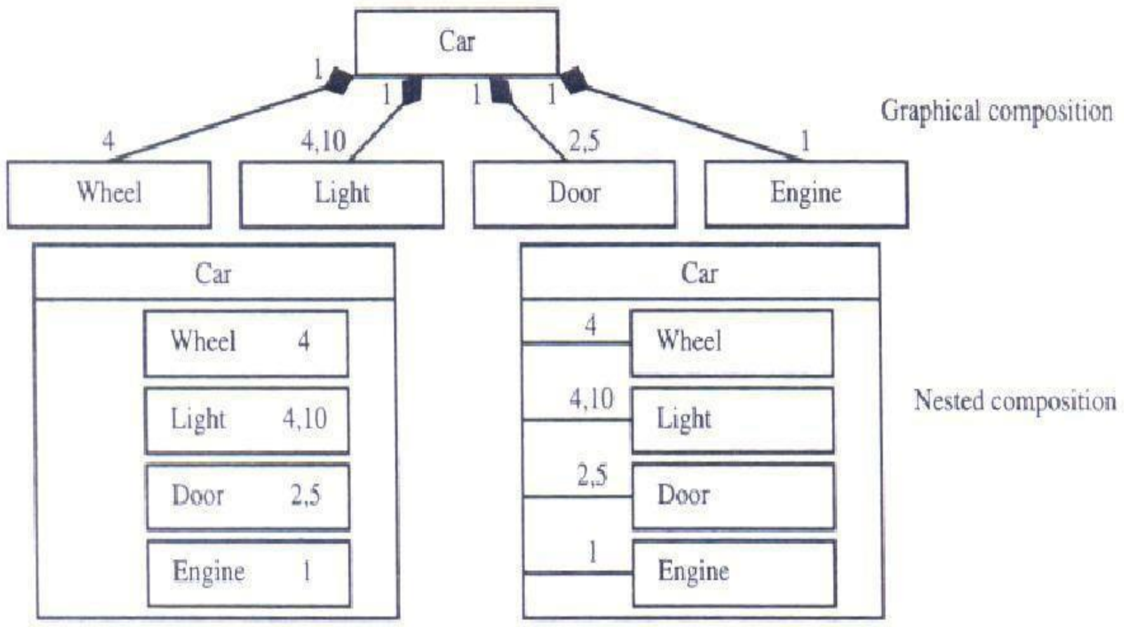


Fig. 2.22. Different ways to show Composition.

Generalization

Generalization is the relationship between a more general class and a more specific class. Generalization is displayed as a directed line with a closed, hollow arrowheadat the superclass end (see Fig 23). The UML allows a **discriminator label** to be attached toa generalization of the superclass.

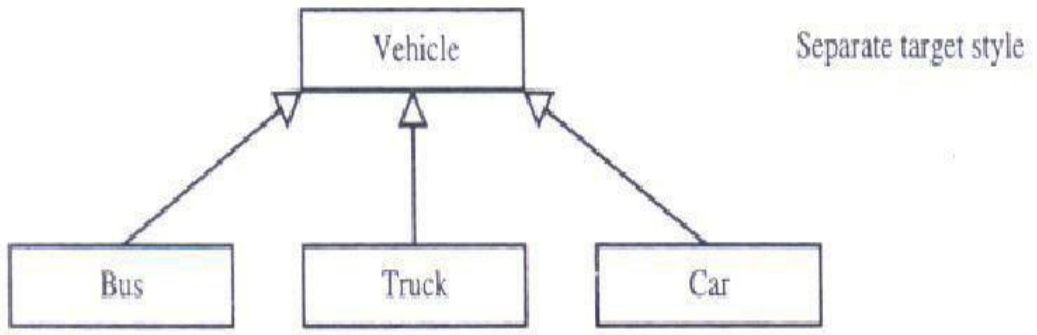
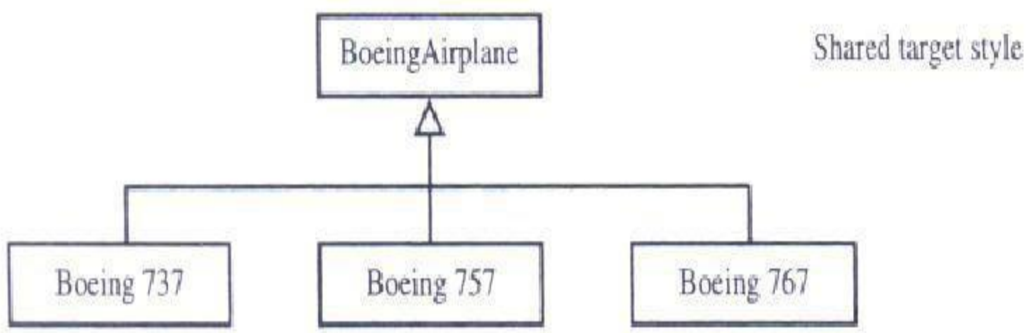
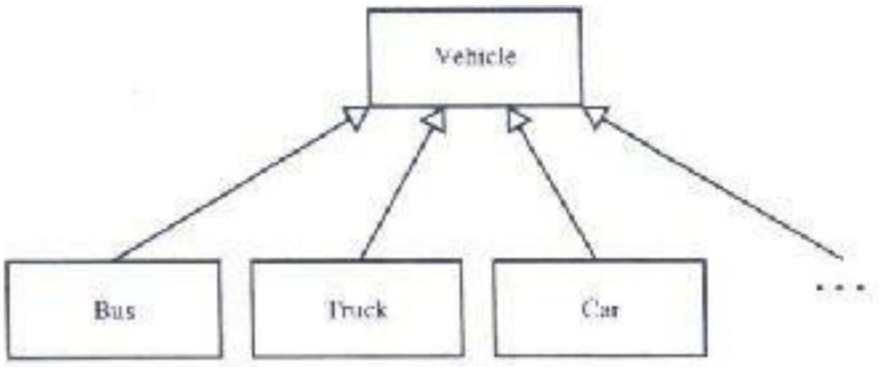


Fig. 2.23. Generalization Notation





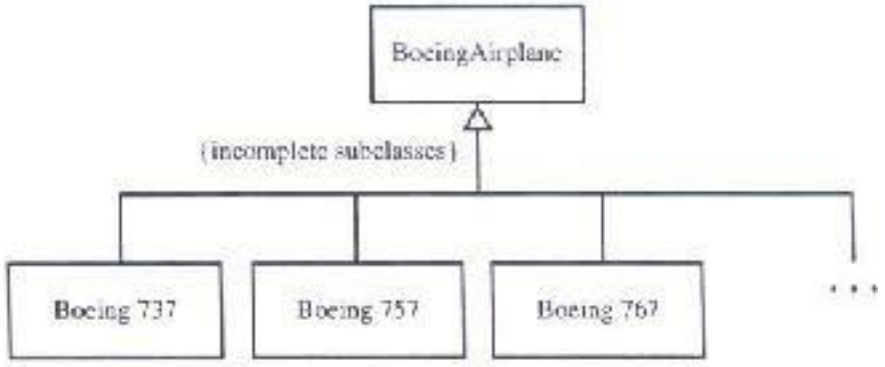


Fig. 2.24. Ellipses(...) indicate that additional classes exist and are not shown.

USE-CASE DIAGRAM

The use-case concept was introduced by Ivar Jacobson in the object-oriented software engineering (OOSE) method. The functionality of a system is described in anumber of different use cases, each of which represents a specific flow of events in the system.

A use case corresponds to a sequence of transactions, in which each transaction is invoked from outside the system (actors) and engages internal objects to interact with one another and with the system's surroundings.

The description of a **use case defines what happens in the system when the use case is performed.** In essence, the use-case model defines the outside (**actors)** and inside(**use case**) of the system's behavior. Use cases represent specific flows of events in the system. The **use cases** are initiated by actors and describe the flow of events that these actors set off. **An actor** is anything that interacts with a use case: It could be a human user, external hardware, or another system. An actor represents a category of user rather than a physical user. Several physical users can play the same role. For example, in termsof a member actor, many people can be members of a library, which can be represented by one actor called **Member**.

A use-case diagram is a graph of actors, a set of use cases enclosed by a system boundary, communication (participation) associations between the actors and the use cases, and generalization among the use cases.

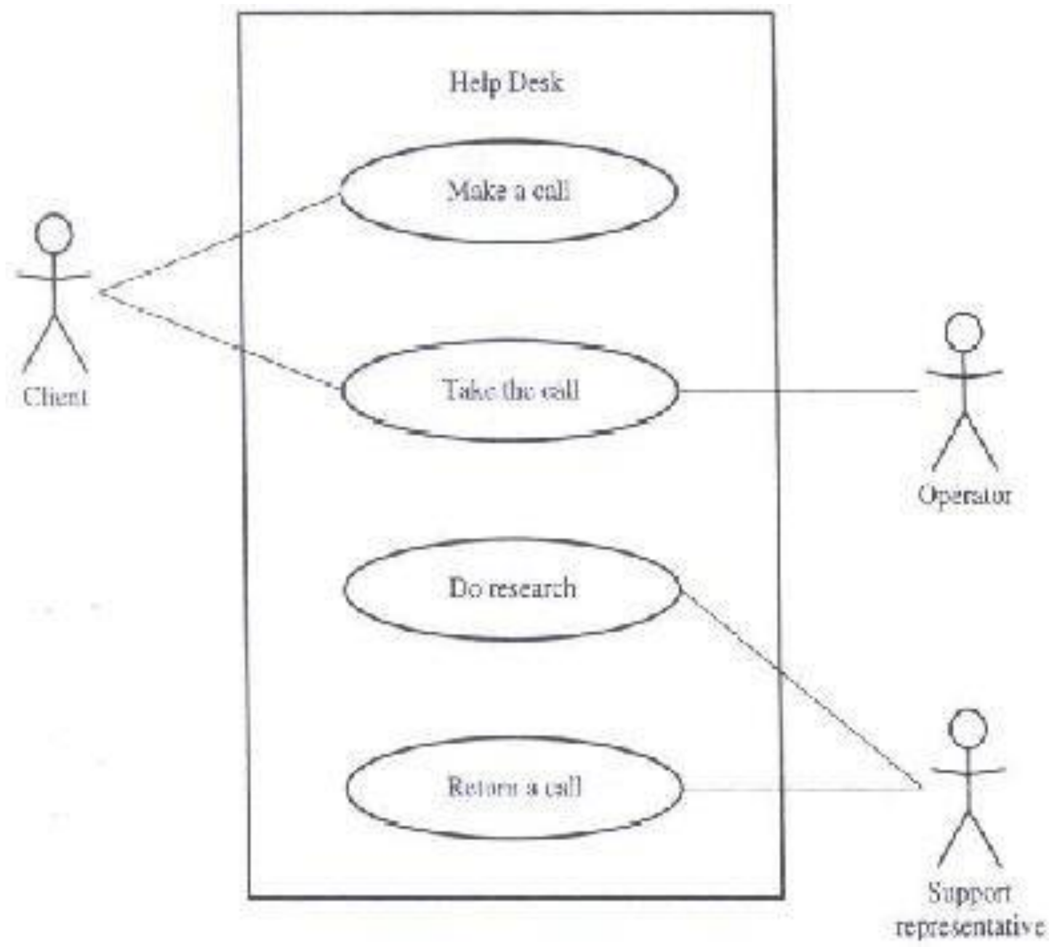


Fig. 2 . 25. Use-case diagram shows the relationship among actors and use cases within a system.

Fig 25. diagrams use cases for a Help Desk. A use-case diagram shows the relationship among the actors and use cases within a system. A client makes a call that is taken by an operator, who determines the nature of the problem. Some calls can beanswered immediately; other calls require research and a return call.

A use case is shown as an ellipse containing the name of the use case. The name ofthe use case can be placed below or inside the ellipse. Actors' names and use case names should follow the capitalization and punctuation guidelines of the model.

An actor is shown as a class rectangle with the label < <actor> >, or the label and astick fig, or just the stick fig with the name of the actor below the fig (see Fig 26).

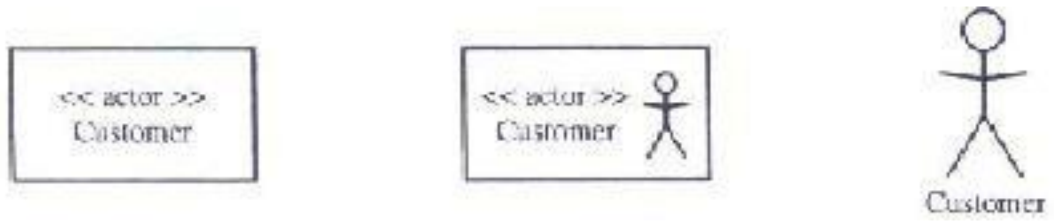


Fig. 2.26. The three representations of an actor are equivalent.

These relationships are shown in a use-case diagram:

1. **Communication.** The communication relationship of an actor in a use case is shown by connecting the actor symbol to the use-case symbol with a solid path. The actor is said to "communicate" with the use case.
2. **Uses.** A uses relationship between use cases is shown by a generalization arrow fromthe use case.
3. **Extends.** The extends relationship is used when you have one use case that is similar to another use case but does a bit more. In essence, it is like a subclass.

UML DYNAMIC MODELING (BEHAVIOR DIAGRAMS)

Booch explains that describing a systematic event in a static medium such as on asheet of paper is difficult, but the problem confronts almost every discipline.

The Dynamic semantics of a problem with the following diagrams:

Behavior diagrams (Dynamic)

- □ Interaction Diagrams:
  - □ Sequence diagrams
  - □ Collaboration diagrams □
- State Chart diagrams
- □ Activity diagrams

Each class may have an associated activity diagram that indicates the behavior of the class's instance (its object). In conjunction with the use-case model, we may provide a scripts or an interaction diagram to show the time or event ordering of messages as they are evaluated
.

UML INTERACTION DIAGRAMS

Interaction diagrams are diagrams that describe how groups of objects collaborate to get the job done.

Interaction diagrams capture the behavior of a single use case, showing the patternof interaction among objects. The diagram shows a number of example objects and the messages passed between those objects within the use case. There are two kinds of interaction models: **sequence diagrams and collaboration diagrams.**

UML Sequence Diagram: Sequence diagrams are an easy and intuitive way of describing the behavior of a system by viewing the interaction between the system and its environment.A sequence diagram shows an interaction arranged in a time sequence. It shows the objects participating in the interaction by their lifelines and the messages they exchange, arrangedin a time sequence.

A sequence diagram has two dimensions: the **vertica**l dimension represents time, the **horizontal** dimension represents different objects. The vertical line is called the **object's lifeline**. The **lifeline** represents the object's existence during the interaction. This form was first popularized by Jacobson. **An object** is shown as a box at the top of a dashed vertical line (see Fig 27). A role is a slot for an object within a collaboration that describes the type ofobject that may play the role and its relationships to other roles. a sequence diagram does not show the relationships among the roles or the association among the objects. An object role is shown as a vertical dashed line, the lifeline.

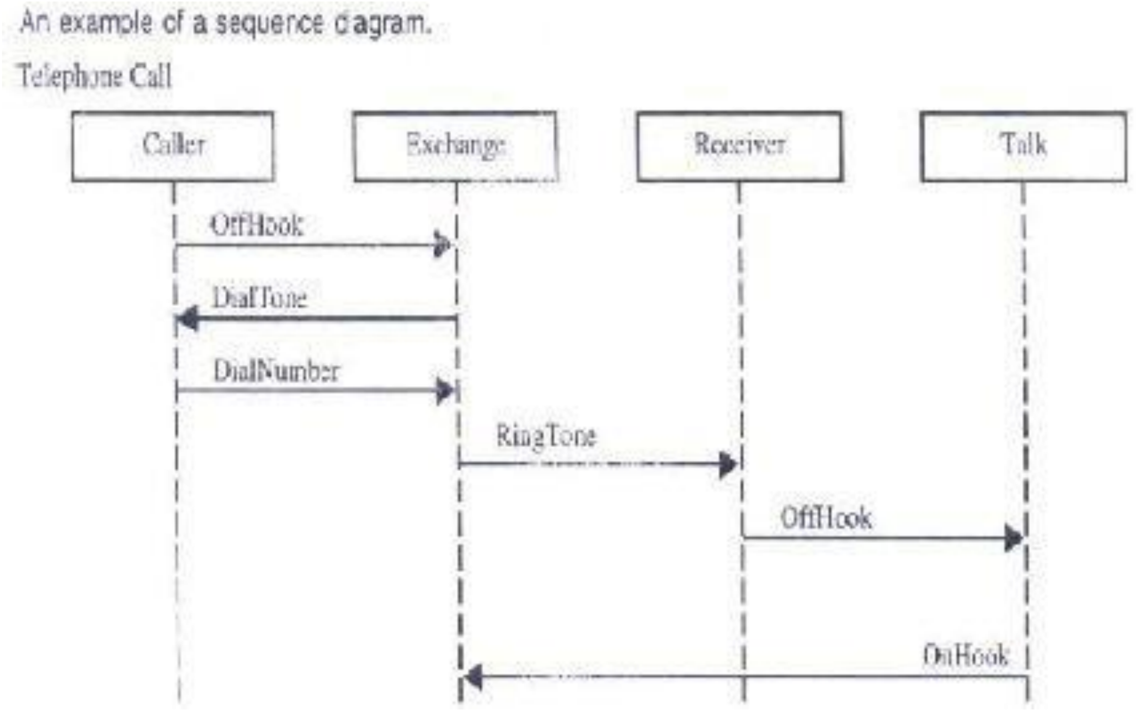


Fig. 2.27. An example of a Sequence Diagram

Each message is represented by an arrow between the lifelines of two objects. The order in which these messages occur is shown top to bottom on the page. Each message is labeled with the message name. The label also can include the argument and some control information and show self-delegation, a message that an object sends to itself, by sending the message arrow back to the same lifeline. The horizontal ordering of the lifelines is arbitrary. Often, call arrows are arranged to proceed in one direction across the page, but this is not always possible and the order conveys no information.

The sequence diagram is very simple and has immediate visual appeal-this is its great strength. A sequence diagram is an alternative way to understand the overall flow of the control of a program. Instead of looking at the code and trying to find out the overall sequence of behavior, you can use the sequence diagram to quickly understand thatsequence.

UML Collaboration Diagram: Another type of interaction diagram is the collaboration diagram. **A collaboration diagram** represents a collaboration, which is a set of objects related in a particular context, and interaction, which is a set of messages exchanged among the objects within the collaboration to achieve a desired outcome. In a collaboration diagram, objects are shown as figs. As in a sequence diagram, arrows indicate the messagesent within the given use case. In a collaboration diagram, the sequence is indicated by numbering the messages.

A collaboration diagram provides several numbering schemes. The simplest is illustrated in Fig 28.

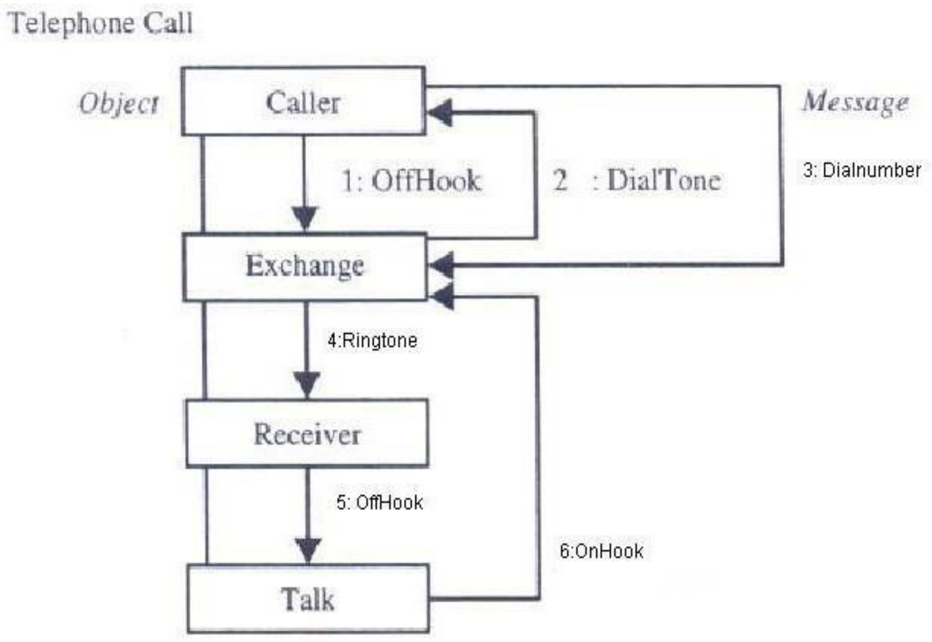


Fig. 2.28. A collaboration diagram with simple numbering.

Fowler and Scott argue that the **main advantage** of interaction diagrams (both collaboration and sequence) is **simplicity**. You easily can see the message by looking atthe diagram. The **disadvantage** of interaction diagrams is that they are great only for representing a single sequential process; they begin to break down when you want to represent conditional looping behavior.

Conditional behavior can be represented in sequence or collaboration diagrams through two methods.

*. The preferred method is to use separate diagrams for each scenario. *. Another way is to use conditions on messages to indicate the
behavior.

UML STATECHART DIAGRAM

A **state chart diagram** (also called a **state diagram**) shows the sequence of states that an object goes through during its life in response to outside stimuli and messages.

The state is the set of values that describes an object at a specific point in time andis represented by state symbols and the transitions are represented by arrows connecting the state symbols. A state chart diagram may contain sub diagrams.

A state diagram represents the state of the method execution (that is, the state of the object executing the method), and the activities in the diagram represent the activities of the object that performs the method.

The purpose of the state diagram is to understand the algorithm involved in performing a method. To complete an object-oriented design, the activities within the diagram must be assigned to objects and the control flows assigned to links in the object diagram.

A state is represented as a rounded box, which may contain one or more compartments. The compartments are all optional. The name compartment and the internal transition compartment are two such compartments:

- □ The **name compartment** holds the optional name of the state. States without names are "anonymous" and all are distinct. Do not show the same named state twice in the same diagram, since it will be very confusing.
- □ The **internal transition compartment** holds a list of internal actions or activities performed in response to events received while the object is in the state, without changing states:

The syntax used is this: event-name argument-list / action-expression; for example, help / display help.

Two special events are entry and exit, which are reserved words and cannot be usedfor event names. These terms are used in the following ways: entry I action expression (the action is to be performed on entry to the state) and exit I action expressed (the action is to beperformed on exit from the state).

The statechart supports nested state machines; to activate a substate machine use the keyword do: do I machine-name (argument-list). If this state is entered, after the entry action is completed, the nested (sub)state machine will be executed with its initial state. When the nested state machine reaches its final state, it will exit the action of the current state, and the current state will be considered completed. An initial state is shown as a smalldot, and the transition from the initial state may be labeled with the event that creates the objects; otherwise, it is unlabeled. If unlabeled, it represents any transition to the enclosing state.

A final state is shown as a circle surrounding a small dot, a bull's-eye. This represents the completion of activity in the enclosing state and triggers a transition on the enclosing state labeled by the implicit activity completion event, usually displayed as an unlabeled transition (see Fig 29).

The transition can be simple or complex. A simple transition is a relationship between two states indicating that an object in the first state

will enter the second state and perform certain actions when a specific event occurs; if the specified conditions are satisfied, the transition is said to "fire." Events are processed one at a time. An event that triggers no transition is simply ignored.

A complex transition may have multiple source and target states. It represents a synchronization or a splitting of control into concurrent threads. A complex transition is enabled when all the source states are changed, after a complex transition "fires" all its destination states. A complex transition is shown as a short heavy bar.! The bar may have one or more solid arrows from states to the bar (these are source states); the bar also may have one or more solid arrows from the bar to states (these are the destination states). A transition string may be shown near the bar. Individual arrows do not have their own transition strings (see Fig 5-30).

A simple state Idle and a nested state. The dialing state contains substates, which consist of start and dial states.

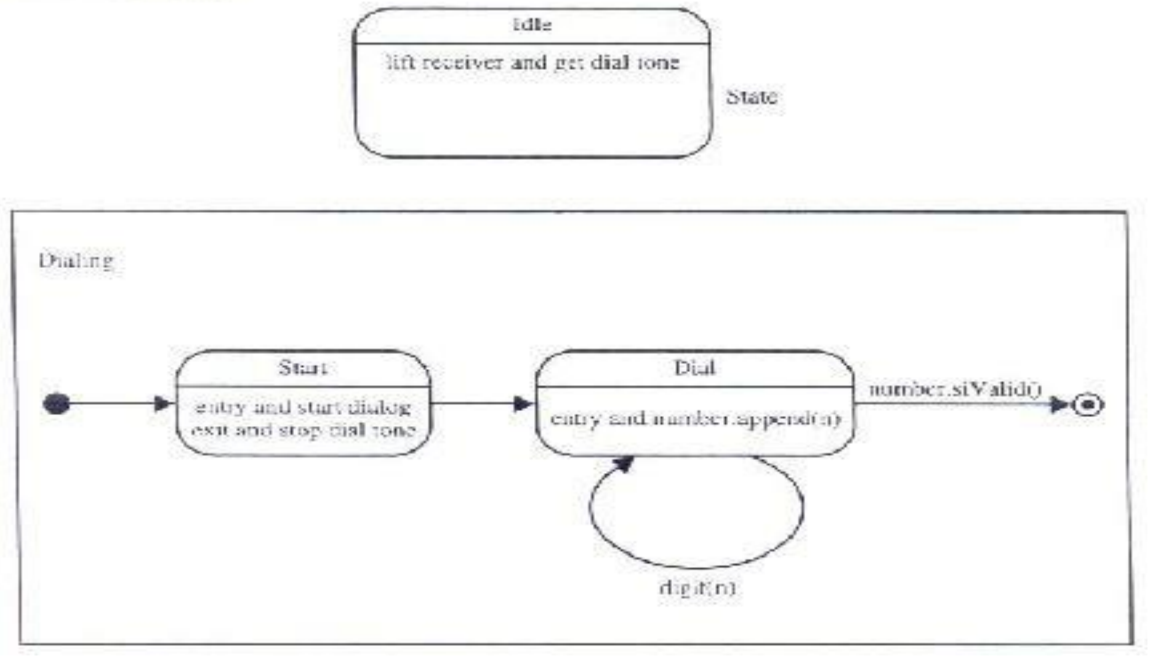


Fig. 2.29. A simple ideal and a nested state.

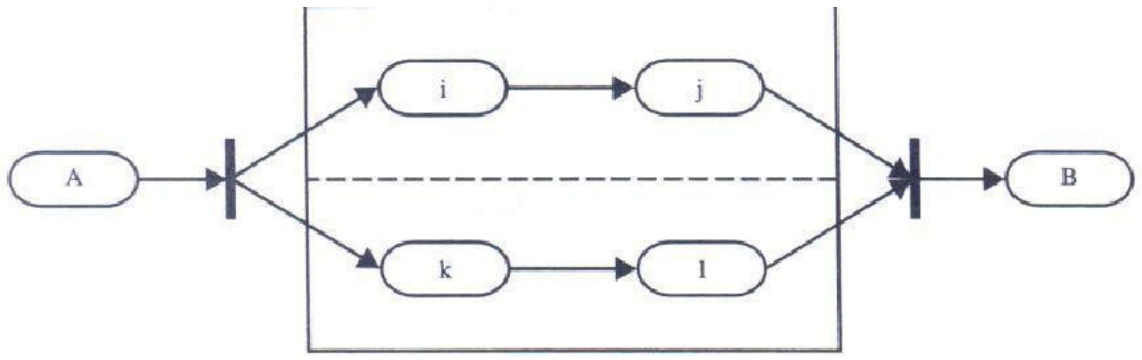
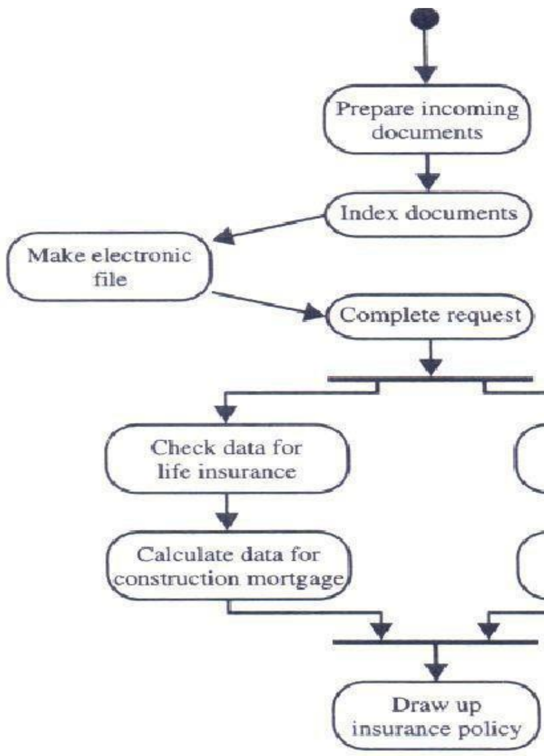


Fig. 2.30. A complex Transition

UML ACTIVITY DIAGRAM

An activity diagram is a variation or special case of a state machine, in which the states are activities representing the performance of operations and the transitions are triggered by the completion of the operations. Unlike state diagrams that focus on the eventsoccurring to a single object as it responds to messages, an activity diagram can be used to model an entire business process. The purpose of an activity diagram is to provide a view offlows and what is going on inside a use case or among several classes. An activity diagram can also be used to represent a class's method implementation.



Prepare incoming documents
Index documents
Make electronic file
Complete request
Check data for life insurance
Draw up contract mortgage-deed
Calculate data for construction mortgage
Pay provision to insurance agent
Draw up insurance policy

Fig. 2.31. An activity diagram for processing mortgage requests (Loan: ProcessingMortgage Request).

An activity model is similar to a state chart diagram, where a token (shown by a black dot) represents an operation. An activity is shown as a round box, containing the name of the operation. When an operation symbol appears within an activity diagram or other state diagram, it indicates the execution of the operation.

Executing a particular step within the diagram represents a state within the execution of the overall method. The same operation name may appear more than once in a state diagram, indicating the invocation of the same operation in different phases.

An outgoing solid arrow attached to an activity symbol indicates a transition triggered by the completion of the activity. The name of this implicit event need not be written, but the conditions that depend on the result of the activity or other values may be included (Fig. 31).

Several transitions with different conditions imply a branching off of control. If conditions are not disjoint, then the branch is nondeterministic. The concurrent control is represented by multiple arrows leaving a synchronization bar, which is represented by a short thick bar with incoming and outgoing arrows. Joining concurrent control is expressedby multiple arrows entering the synchronization bar.

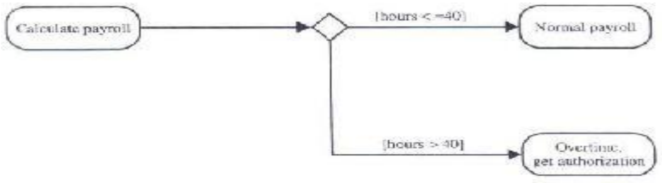


Fig. 2.32. A decision.

An activity diagram is used mostly to show the internal state of an object, but external events may appear in them. An external event appears when the object is in a "waitstate," a state during which there is no internal activity by the object and the object is waitingfor some external event to occur as the result of an activity by another object (such as a user input or some other signal). **The two states are wait state and activity state**. More than one possible event might take the object out of the wait state; the first one that occurs triggers the transition. A wait state is the "normal" state.

Activity and state diagrams express a decision when conditions (the UML calls them **guard conditions**) are used to indicate different possible transitions that depend on Boolean conditions of container object. The fig 2.32 provided for a decision is the traditional diamond shape, with one or more incoming arrows and two or more outgoing arrows, each labeled by a distinct guard condition. All possible outcomes should appear on one
of the outgoing transitions (see Fig 2.32).

Actions may be organized into swimlanes, each separated from neighboring swimlanes by vertical solid lines on both sides. Each **swimlane** represents responsibility for part of the overall activity and may be implemented by one or more objects. The relative ordering of the swimlanes has no semantic significance but might indicate some affinity. Each action is assigned to one swimlane. A transition may cross lanes; there is no significance to the routing of the transition path (see Fig 2.33).

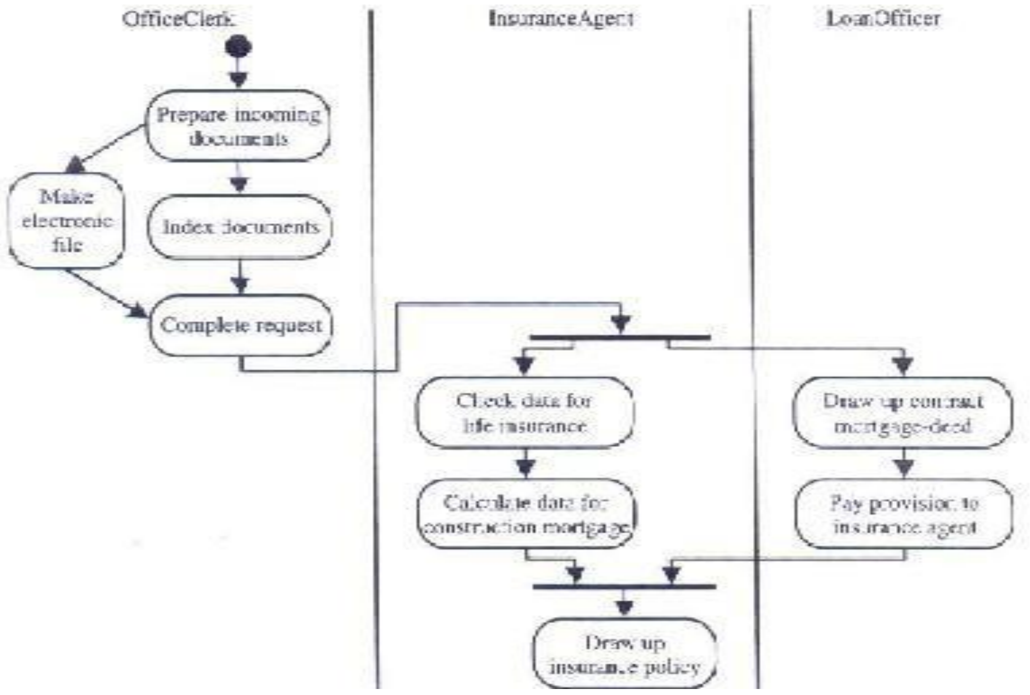

OfficeClerk
InsuranceAgent
LoanOfficer
Prepare incoming documents
Make electronic file
Index documents
Complete request
Check data for life insurance
Calculate data for construction mortgage
Draw up contract mortgage-deed
Pay provision to insurance agent
Draw up insurance policy

Fig. 2.33. Swimlanes in an activity diagram.

UML IMPLEMENTATION DIAGRAMS

Implementation diagrams show the implementation phase of systems development, such as the source code structure and the run-time implementation structure. There are 2 types of implementation diagrams:

*. Component diagrams – Its Show the structure of the code itself.

*. Deployment Diagrams – Its show the structure of the runtime system. These are relatively simple, high-level diagrams compared with other
UML diagrams.

Component Diagram:

Component diagrams model the physical components (such as source code, executable program, user interface) in a design. These high-level physical components mayor may not be equivalent to the many smaller components you use in the creation of your application. For example, a user interface may contain many other off-the-shelf components purchased to put together a graphical user interface.

Another way of looking at components is the concept of **packages**. A package is used to show how you can group together classes, which in essence are smaller scale components. A package usually will be used to group logical components of the application, such as classes, and not necessarily physical components. However, the package could be a first approximation of what eventually will turn into physical grouping. In that case, the package will become a component.

A component diagram is a graph of the design's components connected by dependency relationships. A component is represented by the boxed fig shown in Fig 34 Dependency is shown as a dashed arrow.

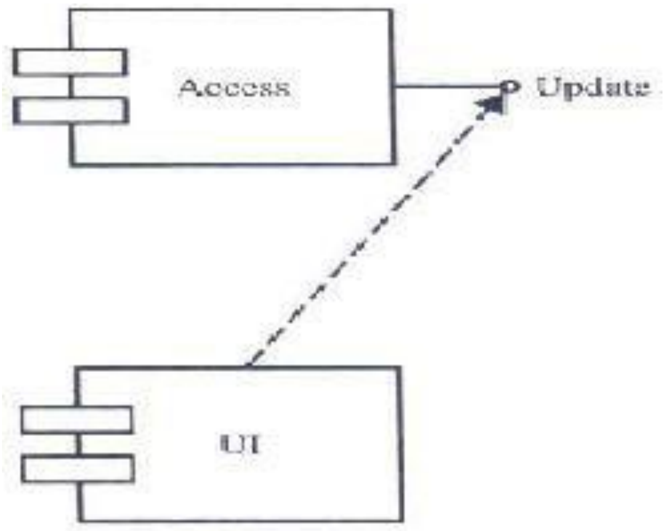


Fig. 2.34. A Component Diagram

Deployment Diagram

Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that live in them. Software component instances represent run-time manifestations of code units. In most cases, component diagrams are used in conjunction with deployment diagrams to show how physical modules of code are distributed on various hardware platforms. In many cases, component and deployment diagrams can be combined.

A deployment diagram is a graph of nodes connected by communication association. Nodes may contain component instances, which mean that the component lives or runs at that node. Components may contain objects; this indicates that the object is part of the component. Components are connected to other components by dashed arrow dependencies, usually through interfaces, which indicate one component uses the services of another. Each node or processing element in the system is represented by a three- dimensional box. Connections between the nodes (or platforms) themselves are shown by solid lines (see Fig 35).

The basic UML notation for a deployment diagram.

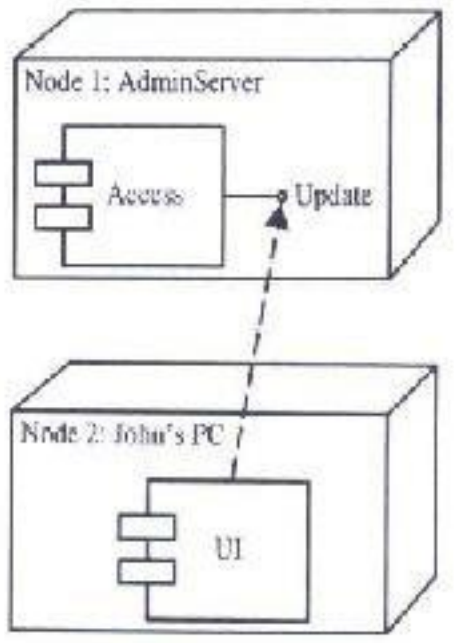


Fig. 2.35. The Basic UML notation for a deployment diagram.

MODEL MANAGEMENT: PACKAGES AND MODEL ORGANIZATION

A **package** is a grouping of model elements. Packages themselves may contain other packages. A package may contain both subordinate packages and ordinary model elements. The entire system can be thought of as a single high-level package with everything else in it. All UML model elements and diagrams can be organized into packages.

A package is represented as a folder, shown as a large rectangle with a tab attachedto its upper left corner. If contents of the package are not shown, then the name of the package is placed within the large rectangle. If contents of the package are shown, then the name of the package may be placed on the tab (see Fig 36). The contents of the package are shown within the large rectangle. Fig shows an example of several packages. This fig shows three packages (Clients, Bank, and Customer) and three classes, (Account class, Savings class, and Checking class) inside the Business Model package.

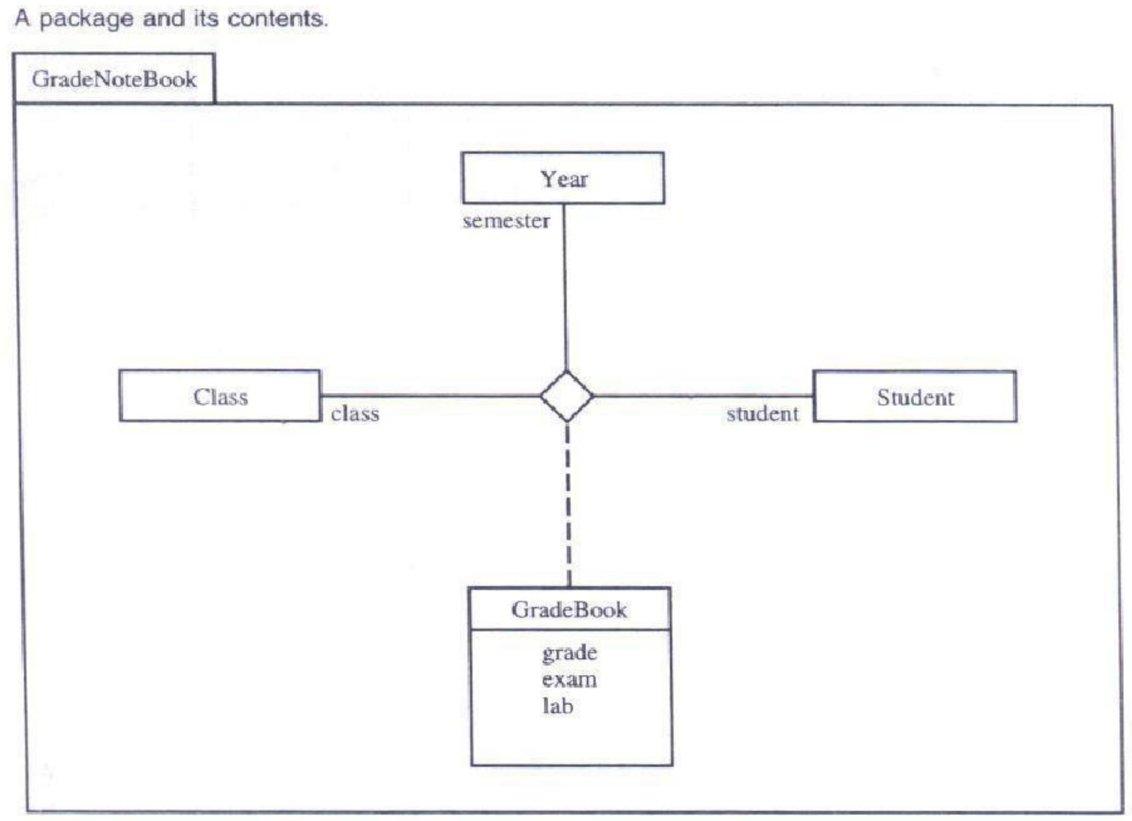


Fig. 2.36. A package and its contents

A real model would have many more classes in each package. The contents might be shown if they are small, or they might be suppressed from higher levels. The entire system is a package. Fig 37 also shows the hierarchical structure, with one packagedependent on other packages. For example, the Customer depends on the package Business Model, meaning that one or more elements within Customer depend on one or more elements within the other packages. The package Business Model is shown partially expanded. In this case, we see that the package Business Model owns the classes Bank, Checking, and Savings as well as the packages Clients and Bank. Ownership may be shown by a graphic nesting of the figs or by the expansion of a package in a separate drawing. Packages can be used to designate not only logical and physical groupings but also use-case groups. A use-case group, as the name suggests, is a package of use cases.

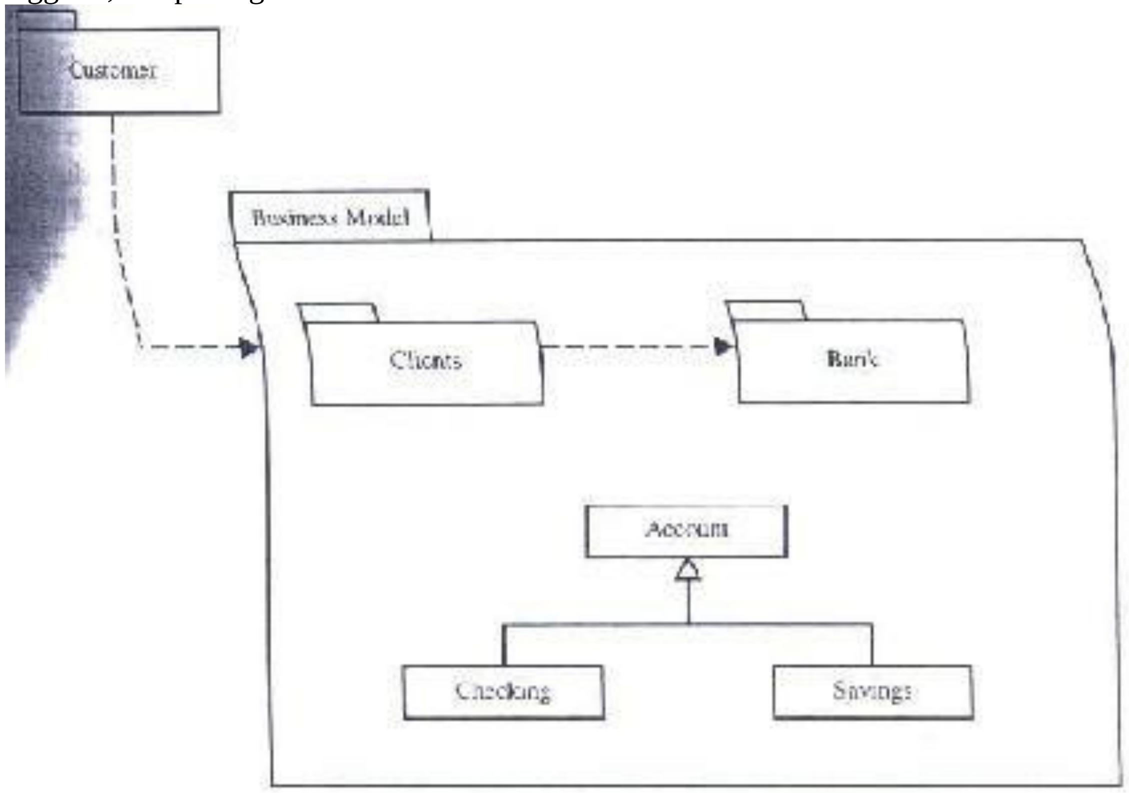


Fig. 2.37. A package and its dependencies

Model dependency represents a situation in which a change to the target element may require a change to the source element in the dependency, thus indicating the relationship between two or more model elements. It relates the model elements themselvesand does not require a set of instances for its meaning. A dependency is shown as a dashed arrow from one model element to another on which the first element is dependent (see Fig 38).

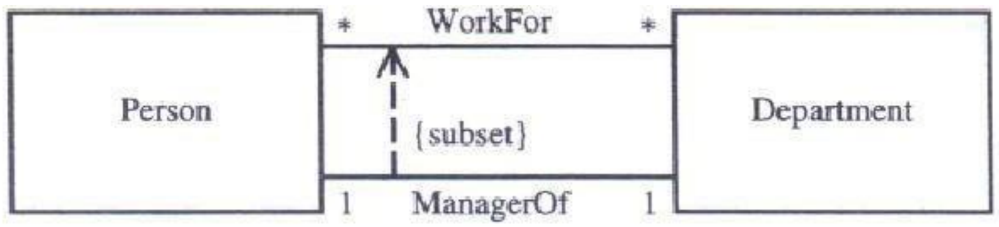


Fig. 2.38. An example of constraints. A person is a manager of people who work for the accounting department.

UML EXTENSIBILITY

1. **Model Constraints and comments**

Constraints are assumptions or relationship among model elements specifyingconditions and propositions that must be maintained as true; otherwise, the system described by the model would be invalid. Some constraints, such as association OR constraints are predefined in the UML; others may be defined by users.

Constraints are shown as text in braces (ref. Fig 38). The UML also provideslanguage for writing constraints in the OCL. The constraints may be written in a natural language.

A constraint may be a “Comment”, in which case it is written in text.

For an element whose notation is a text string such as an attribute, the constraint string may follow the element text string. For a list of elements whose notation is a list of text strings, such as the attributes within class, the constraint string may appear as an element in the list.The constraint applies to all succeeding elements of the list until reaching another constraint string list element or the end of the list. A constraint attached to an individual list element does not supersede the general constraints but may modify individual constraints string maybe placed near the symbol name.

The above example fig 38, shows two classes and two associations. The constraintis shown as a dashed arrow from one element to the other, labeled by the constraints string in brace. The direction of the arrow is relevant information within the constraint.

2. Note

A Note is a graphic symbol containing textual information; it also could contain embedded images. It is attached to the diagram rather than to a model element. A note is shown as a rectangle with “Bent Corner” in the upper right corner. It can contain any length text. (ref. Fig 39).

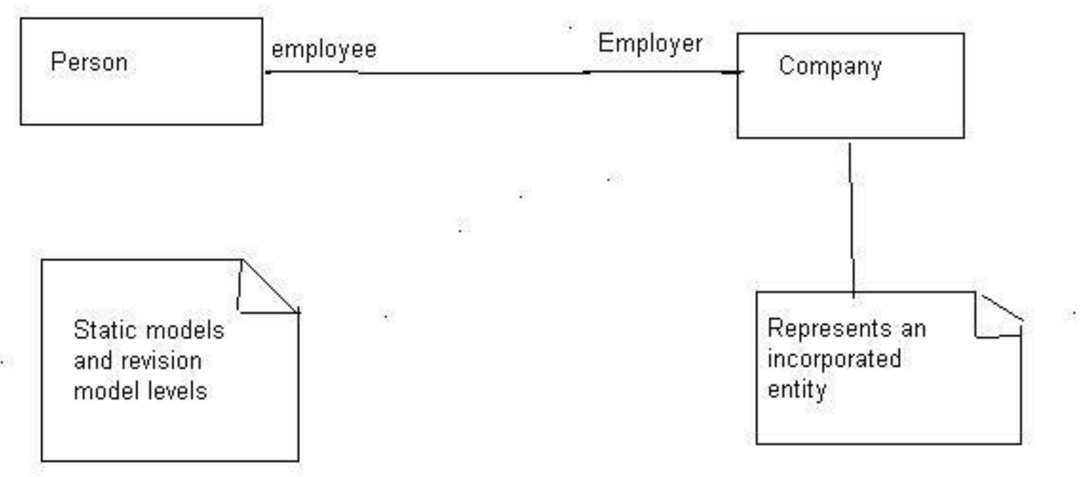

Person
employee
Employer
Company
Static models and revision model levels
Represents an incorporated entity

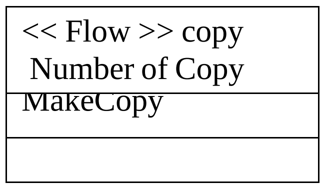
Fig. 2.39. Note

3. Stereotype

Stereotype represent a built-in extensibility mechanism of the UML. User-defined extensions of the UML are enabled through the use of stereotypes and constraints. A stereotype is a new class of modeling element introduced during modeling, It represents a subclass of an existing modeling element with the same form (attributed and relationships) but a different intent. UML stereotype extend and tailor the UML for a specific domain or process.

The general presentation of a stereotype is to use a figure for the base element but place a keyword string above the name of the element (if used, the keyword string is the name of a stereotype within matched guillemets,

“<<”,”>>”, such as <<flow>.. Note that a guillemet looks like a angle-bracket, but it is a single character in most fonts.



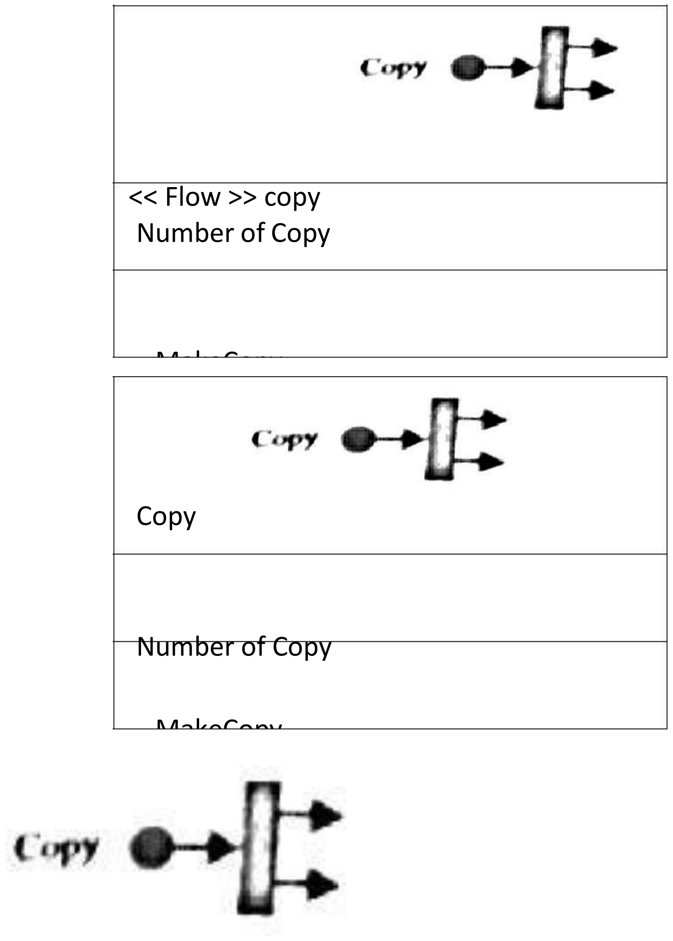


Fig. 2.40. Various forms of Stereotype notation

The stereotype allows extension of UML notation as well as a graphic figure, texture, and color. The figure can be used in one of two ways: (1) instead of or in addition to the stereotype keyword string as part of the symbol for the base model element or (2) as the entire base model element. Other information contained by the base model element symbol is suppressed.

3. UML Meta-model

The UML defined notations as well as a meta model. UML graphic notations can be used not only to describe the system's components but also to describe a model itself. This is known as a **meta-model.**

In other words, **a meta-model** is a model of modeling elements. The purpose of the UML meta-model is to provide a single, common, and definitive statement of the syntax and semantics of the elements of the UML.

The meta model provides us a means to comment different UML diagrams. The connection between the different diagrams is very important, and the UML attempts to make these coupling more explicit through defining the underlying model while imposing no methodology.

The presence of this meta model has made it possible for its developers to agree on semantics and how those semantics would be best rendered. This is an important step forward, since it can assure consistency among diagrams. The meta-model also can serve as a means to exchange data between different case tools. The fig 41 is an example for the UML meta-model that describes relationship with association and generalization; association is depicted as a composition of association roles.

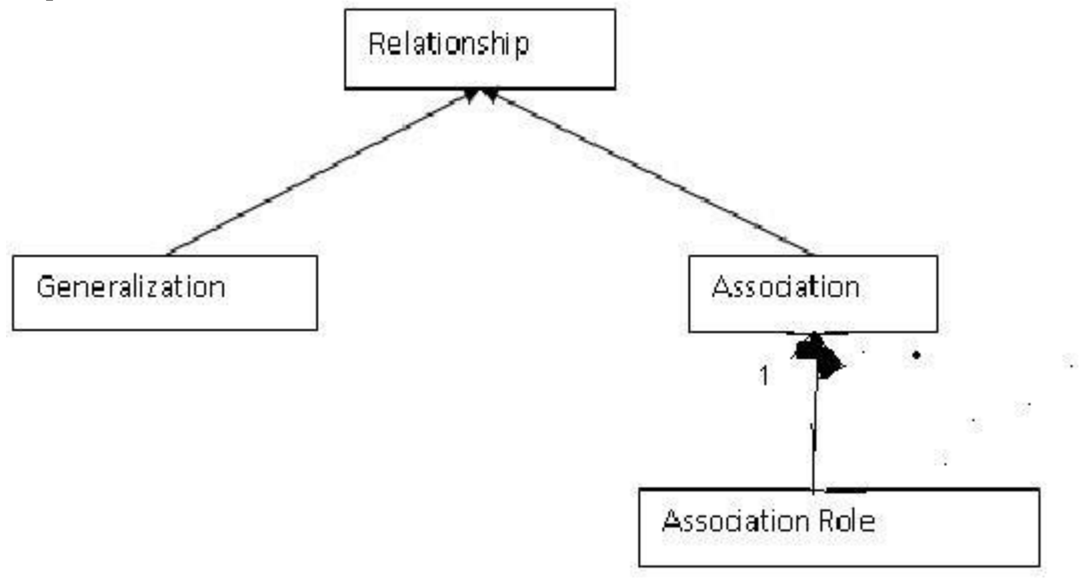


Fig. 2.41. The UML meta-model describing the relationship between association and generalization.

TEXT/ REFERENCE BOOKS:

1. Ali Bahrami, “Object oriented systems development using the unified modelling language”, 1st Edition, McGraw-Hill, 1998.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson, “The Unified Modelling Language User Guide”, 3rd Edition Addison Wesley,2007.
3. John Deacon, "Object Oriented Analysis and Design", 1st Edition, Addison Wesley, 2005.
4. Grady Booch, James Rumbaugh, and Ivar Jacobson, “The Unified Modelling Language User Guide”, 3rd Edition Addison Wesley.
5. John Deacon, “Object Oriented Analysis and Design”, 1st Edition, Addison Wesley.
6. Bernd Oestereich, “Developing Software with UML, Object - Oriented Analysis and Design in Practice”, Addison-Wesley

Questions

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Compare the difference between a method and a process?	Understand	BTL 2
2.	Define an object model? What are the other OMT models?	Knowledge	BTL 1
3.	Sketch the reason for having abstract use cases?	Apply	BTL 3
4.	Describe the difference between patterns and frameworks.	Knowledge	BTL 1
5.	Sketch multiplicity	Apply	BTL 3
6.	Describe the relationships in a use case diagram.	Knowledge	BTL 1
7.	Compare extends and uses relationship?	Understand	BTL 2
8.	Classify the responsibilities of access and view layer?	Understand	BTL 2
9.	Summarize object diagram, binary association, association role, navigability?	Understand	BTL 2
10.	Defend n-ary association, aggregation, composition, generalization, use case diagram?	Evaluate	BTL 5
Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Describe Rumbaugh’s Object Modelling Technique?	Knowledge	BTL 1
2.	Summarize detailed notes about the Booch Methodology?	Understand	BTL 2
3.	Summarize a detailed account of Jacobson methodology?	Understand	BTL 2
4.	Illustrate in detail about the Unified approach?	Analyze	BTL 4

5.	Write static UML diagrams in detail	Create	BTL 6
6.	Distil layered approach to software development.	Analyze	BTL 4
7.	Classify the different types of modeling? Briefly described each.	Understand	BTL 2

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – III - Object Oriented Analysis and Design– SBSA1403

OBJECT ORIENTED ANALYSIS AND DESIGN– SBSA1403

COURSE OBJECTIVES:

- To Understand the fundamentals of Object-Oriented System Development
- To understand the object-oriented methodologies.
- To use UML in requirements elicitation and designing.
- To understand concepts of relationships and aggregations.
- To test the software against its requirements specification

UNIT I - INTRODUCTION

Overview of object-oriented language systems development – Object basics hierarchy – Object and identity –Static and dynamic binding – Object oriented SDLC.

UNIT II - OBJECT ORIENTED METHODOLOGIES

Rumbaugh et al.'s technique – Booch, Jacobson Methodologies – Patterns – Framework – Unified approach –UML – UML diagrams – UML dynamic modelling – UML extensibility – UML meta-model.

UNIT III - OBJECT ORIENTED ANALYSIS

Use case model – Object analysis classification – Approaches for identifying classes – Classes responsibilities and collaborators – Identifying object relationships, attributes and methods.

UNIT IV - OBJECT ORIENTED DESIGN

Design process and design axioms – Designing classes – Access Layer – Object storage and object Interoperability – View layer – Designing interface objects.

UNIT V - SOFTWARE QUALITY

Software quality assurance – Testing strategies – Test cases – Test plan – Myers debugging principle – System usability and measuring user satisfaction.

COURSE OUTCOMES

CO1 - Understand the basics object model for System development.
CO2 - Understand the object-Oriented Methodologies
CO3 - Express software design with UML diagrams
CO4 - Understand the concept of Relationships
CO5 - Design software applications using OO concepts.
CO6 -Understand the various testing methodologies for OO software

UNIT – III

OBJECT ORIENTED ANALYSIS: USE-CASE DRIVEN

Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirement. The goal of object-oriented analysis is to understand the domain of tile problem and the system's responsibilities by understanding how the users use or will use the system. The first step in finding an appropriate solution to a given problem is to understand the problem and its domain.

The main objective of the analysis is to capture a complete, unambiguous, and consistent picture of the requirements of the system and what the system must do to satisfy the users' requirements and needs. This is accomplished by constructing several models of the system that concentrate on describing what the system does rather than how it does it. Separating the behavior of a system from the way that behavior is implemented requires viewing the system from the perspective of the user rather than that of the machine. Analysis is the process of transforming a problem definition from a fuzzy set of facts and myths into a coherent statement of a system's requirements.

3.1 WHY ANALYSIS IS A DIFFICULT ACTIVITY

Analysis is a creative activity that involves understanding the problem, its associated constraints, and methods of overcoming those constraints. This is an iterative process that goes on until the problem is well understood. Norman explains the three most common sources of requirement difficulties:

1. Fuzzy descriptions - such as "fast response time" or "very easy and very secure updating mechanisms." A requirement such as fast response time is open to interpretation, which might lead to user dissatisfaction if the user's interpretation of a fast response is different from the systems analyst's interpretation

2. Incomplete requirements - mean that certain requirements necessary for successful system development are not included for a variety of reasons. These reasons could include the users' forgetting to identify them, high cost, politics within the business,or oversight by the system developer. However, because of the iterative nature of object- oriented analysis and the unified approach most of the incomplete requirements can be identified in subsequent tries.

3. Unnecessary features - When addressing features of the system, keep in mind that every additional feature could affect the performance, complexity, stability, maintenance, and support costs of an application. Features implemented by a small extension to the application code do not necessarily have a proportionally small effect on a user interface.

Analysis is a difficult activity. You must understand the problem in some application domain and then define a solution that can be implemented with software. Experience often is the best teacher. If the first try reflects the errors of an incomplete understanding of the problems, refine the application and try another run.

3.2 BUSINESS OB.JECT ANALYSIS: UNDERSTANDING THE BUSINESS LAYER

Business object analysis is a process of understanding the system's requirements and establishing the goals of an application. The main intent of this activity is to understand users' requirements. The outcome of the business object analysis is to identify classes that make up the business layer and the relationships that play a role in achieving system goals.

3.3 USECASE DRIVEN OBJECT-ORIENTED ANALYSIS: THE UNIFIEDAPPROACH

The object-oriented analysis (OOA) phase of the unified approach uses **actors** and **use cases to describe the system from the users' perspective.** The ***actors*** **are external factors** that interact with the system; ***use cases*** **are scenarios** that describe how

actors use the system. The use cases identified here will be involved throughout the development process.

The OOA process consists of the following steps:

1 Identify the actors:

*Who is using the system?

*Or, in the case of a new system, who will be using the system?

2. Develop a simple business process model using UML activity diagram.

3. Develop the use case:

*What are the users doing with the system?

*Or, in case of the new system, what will users be doing with the system?

*Use cases provide us with comprehensive documentation of the system under study.

4. Prepare interactiondiagrams:

* Determine thesequence.
* Develop collaboration diagrams.

5. Classification -develop a static UML class diagram:

* Identify classes.
* Identify relationships.
* Identify attributes.
* Identify methods.

6. Iterate and refine: If needed, repeat the preceding steps.

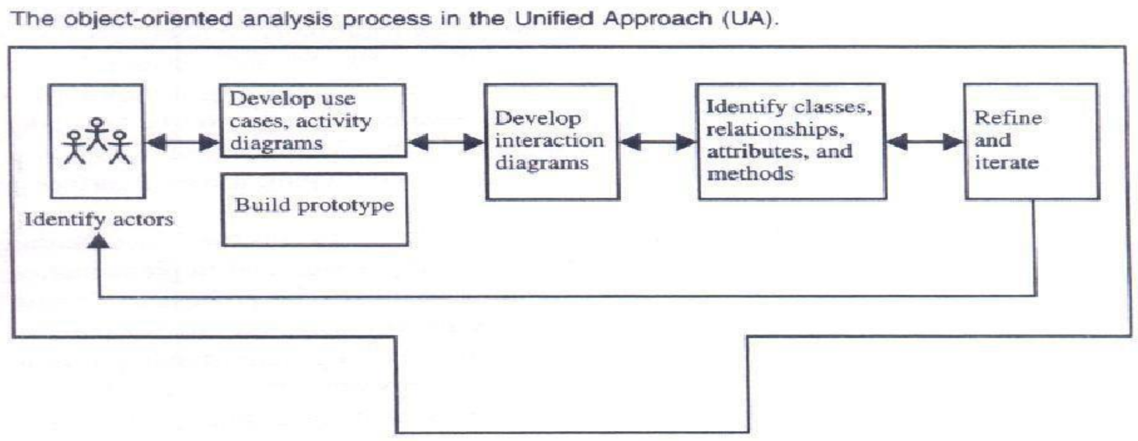

The object-oriented analysis process in the Unified Approach (UA).
Identify actors
Develop use cases, activity diagrams
Build prototype
Develop interaction diagrams
Identify classes, relationships, attributes, and methods
Refine and iterate

Fig 3.1. Object Oriented analysis process in the Unified Approach (UA)

3.4 BUSINESS PROCESS MODELING

This may include modeling as-is processes and the applications that support them and any number of phased, would-be models of reengineered processes or implementation of the system. These activities would be enhanced and supported by using an activity diagram. Business process modeling can be very time consuming, so the main idea should be to geta basic model without spending too much time on the process.

The advantage of developing a business process model is that it makes you more familiar with the system and therefore the user requirements and also aids in developing use cases. For example, let us define the steps or activities involved in using your school library. These activities can be represented with an activity diagram. (See Fig- 3.2) Developing an activity diagram of the business process can give us a better understanding of what sort of activities are performed in a library by a library member.

FIGURE

This activity diagram (AD) shows some activities that can be performed by a library member.

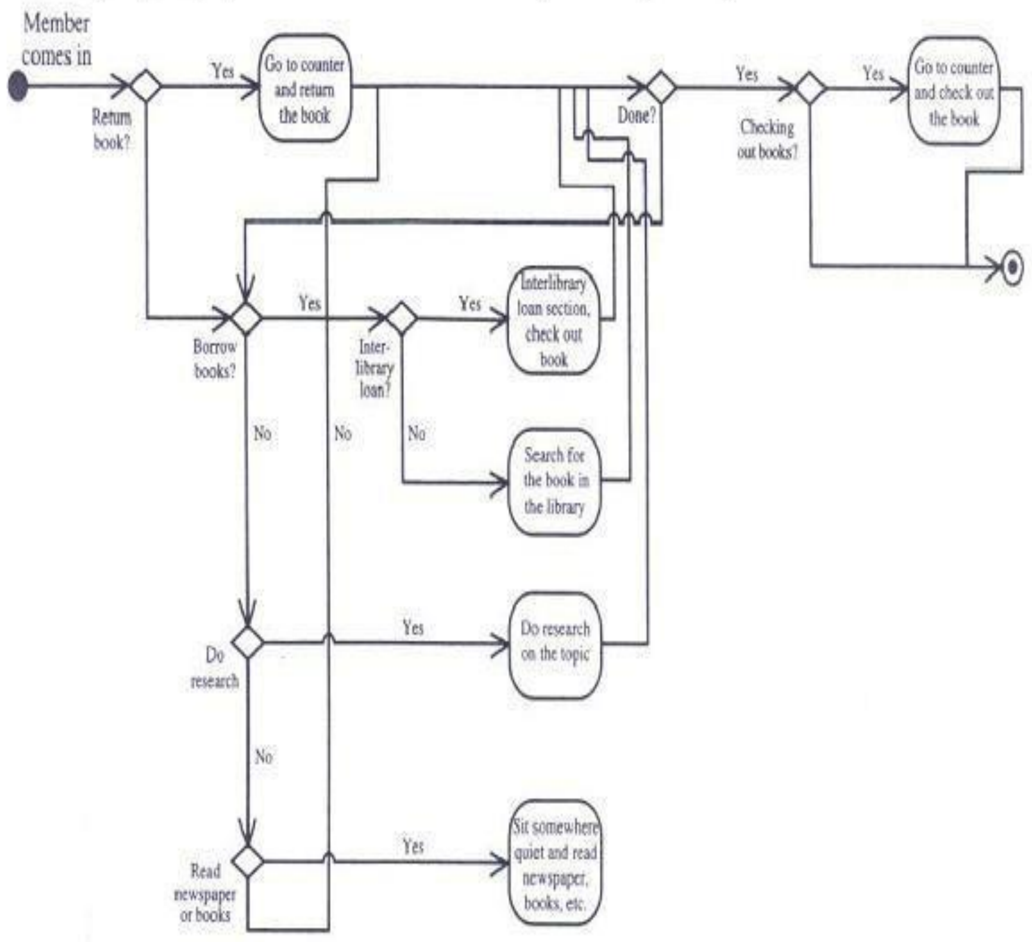

Member comes in
Return book?
Yes
Go to counter and return the book
Done?
Yes
Checking out books?
Yes
Go to counter and check out the book
Borrow books?
Yes
Inter-library loan?
Yes
Interlibrary loan section, check out book
No
No
No
Search for the book in the library
Do research
Yes
Do research on the topic
No
Read newspaper or books
Yes
Sit somewhere quiet and read newspaper, books, etc.

Fig. 3.2. This activity diagram shows some activities that can be performed by a library member.

3.5 USE-CASE MODEL

Use cases are scenarios for understanding system requirements.

- A use-case model can be instrumental in project development, planning, and documentation of systems requirements.
- A use case is an interaction between users and a system; it captures the goal of the users and the responsibility of the system to its users). For example, take a car; typical uses of a car include "take you different places" or "haul your stuff" or a user may wantto use it "off the road."
- The use-case model describes the uses of the system and shows the courses of eventsthat can be performed.
- A use-case model also can discover classes and the relationships among subsystems ofthe systems.
- Use-case model can be developed by talking to typical users and discussing thevarious things they might want to do with the application being prepared.
- Each use or scenario represents what the user wants to do.
- Each use case must have a name and short textual description, no more than a fewparagraphs.
- Since the use-case model provides an external view of a system or application, it is directed primarily toward the users or the "actors" of the systems, not its implementers(see Figure 3.3).
- The use-case model expresses what the business or application will do and not how;that is the responsibility of the UML class diagram

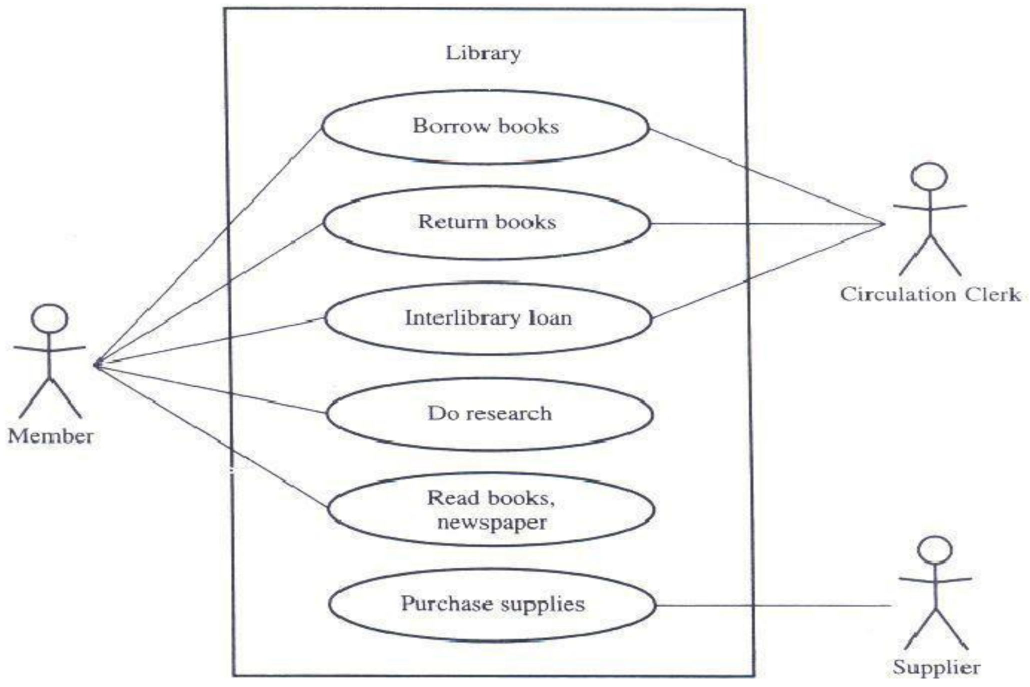


Fig 3.3. Use Case Diagram – Library System

The UML class diagram, also called an object model, represents the static relationships between objects, inheritance, association, and the like. The object model represents an internal view of the system, as opposed to the use-case model, which represents the external view of the system. The object model shows how the business is run. Jacobson, Ericsson, and Jacobson call the use-case model a "what model," in contrast to the object model, whichis a "how model.".

Guidelines for developing Use Case Models:

1. Use Cases under the Microscope
2. Uses and Extends Associations
3. Identifying the Actors
4. Guidelines for Finding Use Cases
5. How Detailed Must a Use Case Be? When to Stop Decomposing and When to Continue
6. Dividing Use Cases into Packages
7. Naming a Use Case

1. Use Cases under the Microscope:

Use cases represent the things that the user is doing with the system, which can be different from the users' goals.

Definition of use case by Jacobson "A Use Case is a sequence of transactions in a system whose task is to yield results of measurable value to an individual actor of the system."

Now let us take a look at the **key words of this definition:**

- **Use case** - Use case is a special flow of events through the system. By definition, many courses of events are possible and many of these are very similar.
- **Actors** - An actor is a user playing a role with respect to the system. When dealing withactors, it is important to think about roles rather than just people and their job titles.
- **Ina system** -. This simply means that the actors communicate with the system's use case.
- **A measurable value**- A use case must help the actor to perform a task that has some identifiable value.
- **Transaction**. - A transaction is an atomic set of activities that are performed either fully or not at all. A transaction is triggered by a stimulus from an actor to the system or by a point in time being reached in the system.

The following are some examples of use cases for the library (see Figure 3.4).

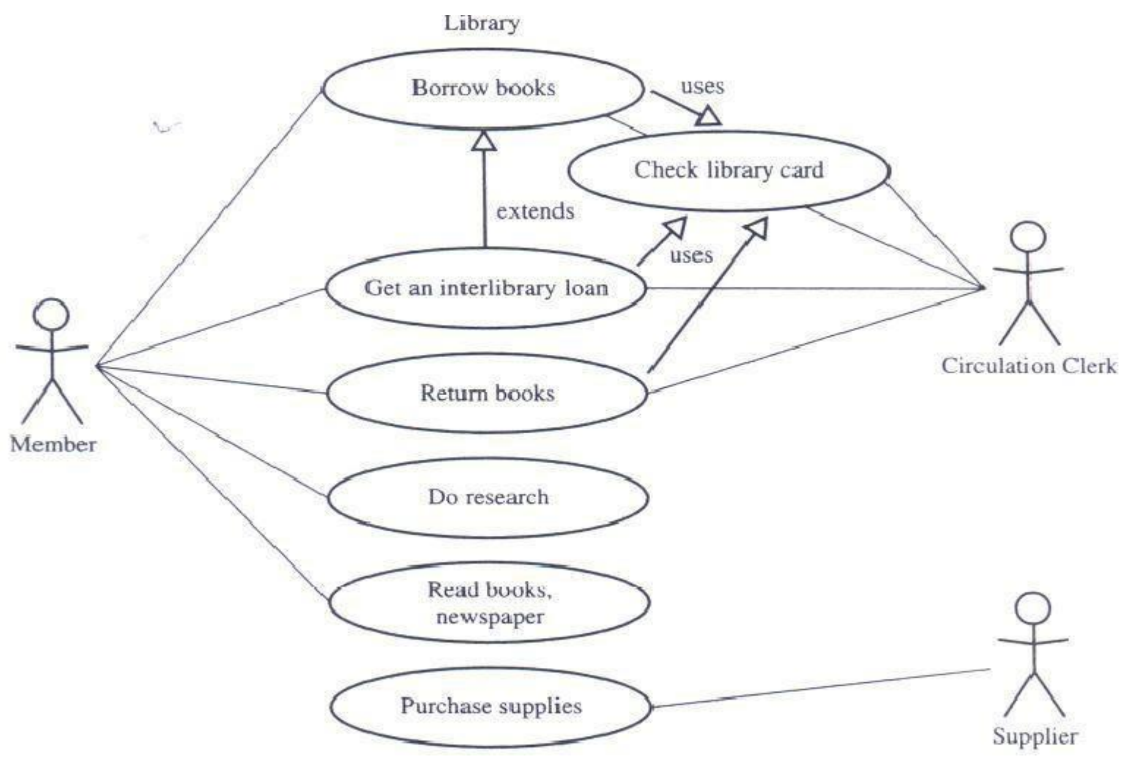


Fig. 3.4. A member, a circulation clerk, and a supplier.

1. **Use-case name: Borrow books**. A member takes books from the library to read at home, registering them at the checkout desk so the library can keep track of its books. Depending on the member's record, different courses of events will follow.
2. **Use-case name: Get an interlibrary loan**. A member requests a book that the library does not have. The book is located at another library and ordered through an interlibrary loan.
3. **Use-case name: Return books**. A member brings borrowed books back to the library.
4. **Use-case name: Check library card.** A member submits his or her library card to the clerk, who checks the borrower's record.
5. **Use-case name: Do research**. A member comes to the library to do research. The member can search in a variety of ways (such as through books, journals, CDROM, WWW) to find information on the subjects of that research.
6. **Use-case name: Read books, newspaper**. A member comes to the library for aquiet place to study or read a newspaper, journal, or book.
7. **Use-case name: Purchase supplies**. The supplier provides the books, journals, and newspapers purchased by the library.

2. Uses and Extends Associations:

A use-case description can be difficult to understand if it contains too many alternatives or exceptional flows of events that are performed only if certain conditions are met as the use-case instance is carried out.

A way to simplify the description is to take advantage of extends and uses associations. The extends association is used when you have one use case that is similar to another use case but does a bit more or is more specialized; in essence, it is like a subclass.

The uses association occurs when you are describing your use cases and notice that some of them have sub flows in common. To avoid describing a sub flow more than once in several use cases, you can extract the common sub flow and make it a use case of its own. This new use case then can be used by other use cases. The relationships among the other use cases and this new extracted use case are called a uses association. The uses association helps us avoid redundancy by allowing a use case to be shared. For example, checking a library card is common among the borrow books, return books, and interlibrary loan use cases (see Figure 3.4).

The similarity between extends and uses associations is that both can be viewed as a kind of inheritance. When you want to share common sequences in several use cases, utilize the uses association by extracting common sequences into a new, shared use case. The extends association is found when you add a bit more specialized, new use case that extends some of the use cases that you have.

Use cases could be viewed as concrete or abstract. An abstract use case is not complete and has no initiation actors but is used by a concrete use case, which does interact with actors. This inheritance could be used at several levels. Abstract use cases also are the use cases that have uses or extends associations.

Fowler and Scott provide us excellent (guidelines for addressing variations in use case modeling : .

1. Capture the simple and normal use case first.
2. For every step in that use case, ask
   *. What could go wrong here?
   *. How might this work out differently?
3. Extract common sequences into a new, shared use case with the uses association. If you are adding more specialized or exceptional uses cases, take advantage of usecases you already have with the extends association.

3. Identifying the Actors:

Identifying the actors is (at least) as important as identifying classes, structures, associations, attributes, and behavior.

The term actor represents the role a user plays with respect to the system. When dealing with actors, it is important to think about roles rather than people or job titles.

A user may play more than one role. For instance, a member of a public library also may play the role of volunteer at the help desk in the library. However, an actor should represent a single user; in the library example, the member can perform tasks some of which can be done by others and others that are unique. However, try to isolate the roles that the users can play. (Fig 3.5)

You have to identify the actors and understand how they will use and interact with the system. In a thought-provoking book on requirement analysis, Gause and Weinberg , explain what is known as the railroad paradox:

*When trying to find all users, we need to beware of the **Railroad Paradox**. When railroads were asked to establish new stops on the schedule, they "studied the requirements," by sending someone to the station at the designated time to see if anyone was waiting for a train. Of course, nobody was there because no stop was scheduled, so the railroad turned down the request because there was no demand.*

Gause and Weinberg concluded that the railroad paradox appears everywhere thereare products and goes like this (which should be avoided):

1. The product is not satisfying the users.
2. Since the product is not satisfactory, potential users will not use it.
3. Potential users ask for a better product.
4. Because the potential users do not use the product, the request is denied.

Therefore, since the product does not meet the needs of some users, they are not identified as potential users of a better product. They are not consulted and the product stays bad. The railroad paradox suggests that a new product actually can create users where none existed before Candidates for actors can be found through the answers to thefollowing questions:

- Who is using the system? Or, who is affected by the system? Or, which groups need help from the system to perform a task?
- Who affects the system? Or, which user groups are needed by the system to perform itsfunctions? These functions can be both main functions and secondary functions, suchas administration.
- Which external hardware or other systems (if any) use the system to perform tasks?
- What problems does this application solve (that is, for whom)? And, finally, howdo users use the system (use case)? What are they doing with the system?

When requirements for new applications are modeled and designed by a group that excludes the targeted users, not only will the application not meet the users' needs, but potential users will feel no involvement in the process and not be committed to giving the application a good try. Always remember Veblen's principle: 'There's no change, no matter how awful, that won't benefit some people; and no change, no matter how good, that won't hurt some.”,

Another issue worth mentioning is that **actors need not be human**, although actors are represented as stick figures within a use case diagram. An actor alsocan be an external system. For example, an accounting system that needs information from a system to update its accounts is an actor in that system.

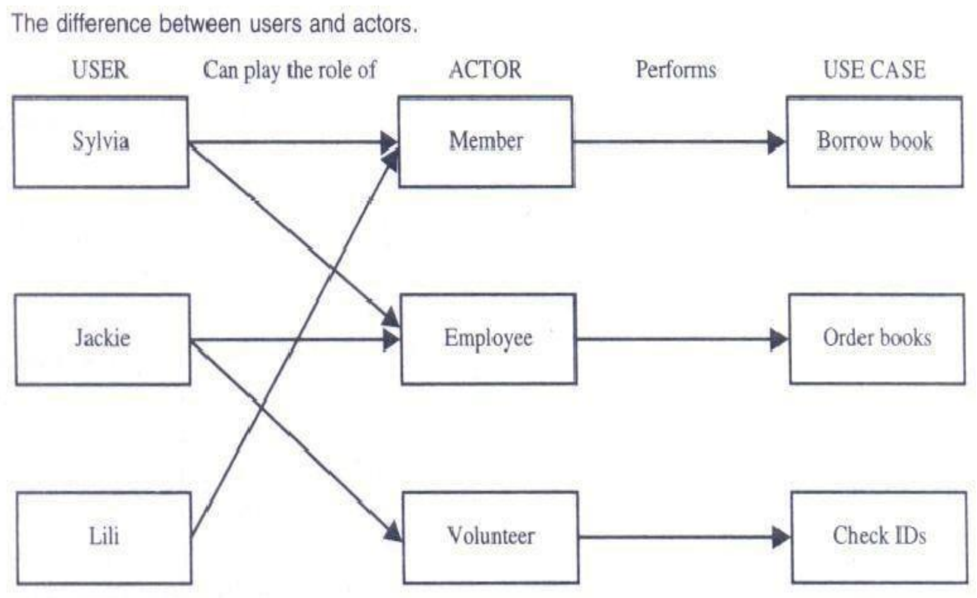


Fig. 3.5. The difference between users and actors

4. Guidelines for Finding Use Cases:

When you have defined a set of actors, it is time to describe the way they interact with the system. This should be carried out sequentially, but an iterated approach may be necessary.

Here are the **steps** for **finding use cases:**

1. For each actor, find the tasks and functions that the actor should be able to perform or that the system needs the actor to perform. The use case should represent a course of events that leads to a clear goal (or, in some cases, several distinct goals that could be alternatives for the actor or for the system).
2. Name the use cases
3. Describe the use cases briefly by applying terms with which the user is familiar. This makes the description less ambiguous.

Once you have identified the use-cases candidates, it may not be apparent that all of these use cases need to be described separately; some may be modeled as variants of others. Consider what the actors want to do.

It is important to separate actors from users. The actors each represent a role that one or several users can play. Therefore, it is not necessary to model different actors that can perform the same use case in the same way. The approach should allow different users to be different actors and play one role when performing a particular actor's use case. Thus, each use case has only one main actor. To achieve this, you have to

- **Isolate users from actors**.
- **Isolate actors from other actors** (separate the responsibilities of each actor).
- **Isolate use cases that have different initiating actors and slightly different behavior** (If the actor had been the same, this would be modeled by a use-case alternativebehavior).

5. How Detailed Must a Use Case Be? When to Stop Decomposing and When to Continue

A use case, as already explained, describes the courses of events that will be carried out by the system. Jacobson et al. believe that, in most cases, too much detail maynot be very useful.

During analysis of a business system, you can develop one use-case diagram asthe system use case and draw packages on this use case to represent the various business domains of the system. For each package, you may create a child usecase diagram. On each child use-case diagram, you can draw all of the use cases of the domain, with actions and interactions. You can further refine the way the use cases are categorized.The extends and uses relationships can be used to eliminate redundant modeling of scenarios.

When should use cases be employed? Use cases are an essential tool in capturing requirements and planning and controlling any software development project.

Capturing use cases is a primary task of the analysis phase. Although most use cases are captured at the beginning of the project, you will uncover more as you proceed.

The UML specification recommends that at least one scenario be prepared for each significantly different kind of use case instance

6. Dividing Use Cases into Packages

Each use case represents a particular scenario in the system.

You may model either how the system currently works or how you want it to work.

A design is broken down into packages.

You must narrow the focus of the scenarios in your system.

For example, in a library system, the various scenarios involve a supplier providing books or a member doing research or borrowing books. In this case, there should be threeseparate packages, one each for Borrow books, Do research, and Purchase books.

Many applications may be associated with the library system and one or more databases used to store the information (see Figure 3.6).

7. Naming a Use Case

Use-case names should provide a general description of the use-case function.

The name should express what happens when an instance of the use case is performed."

Jacobson et al. recommend that the name should be active, often expressed in the form of **a verb** (Borrow) or **verb and noun** (Borrow books).

The naming should be done with care; the description of the use case should be descriptive and consistent.

For example, the use case that describes what happens when a person deposits money into an ATM machine could be named either receive money or deposit money. A library system can be divided into many packages, each of which encompasses multiple use cases.

.

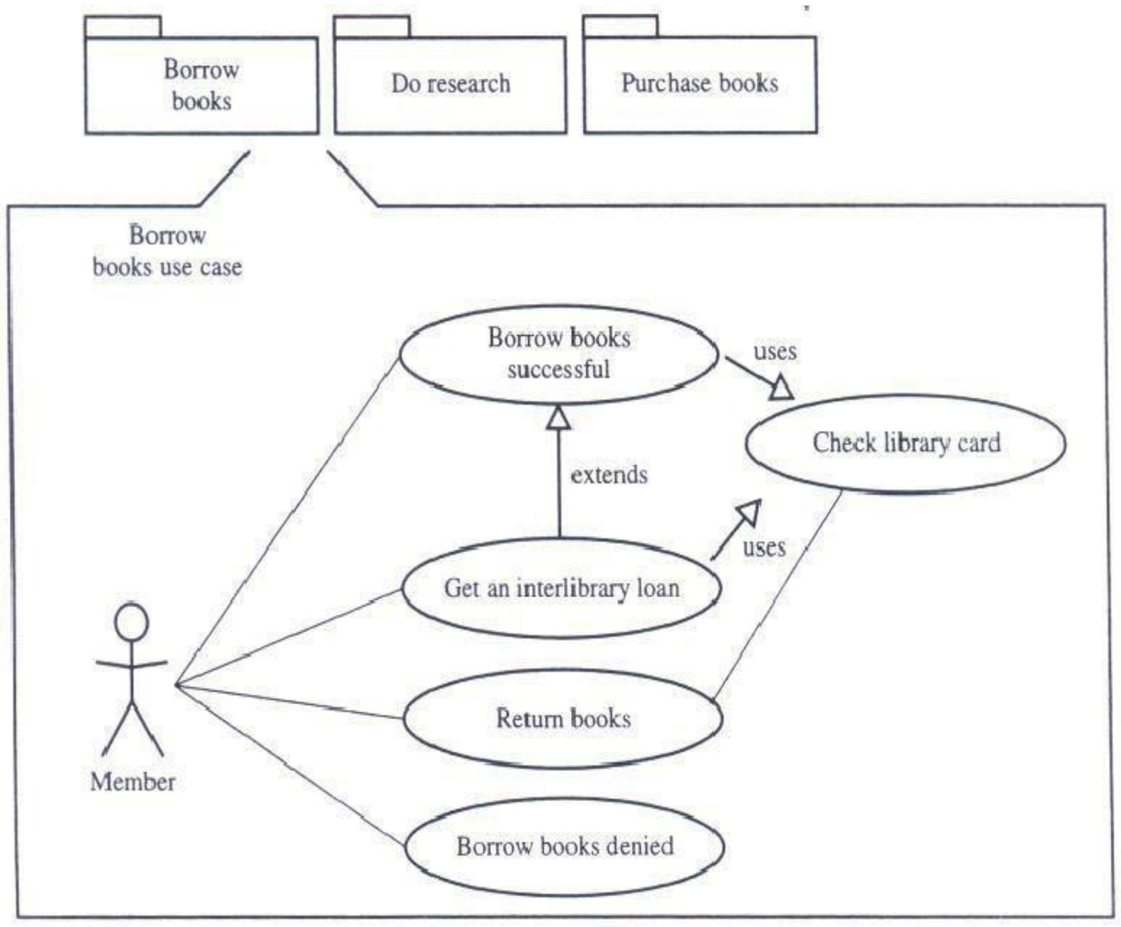


Fig. 3.6. A library system can be divided into many packages, each of whichencompasses multiple use cases.

3.6 DEVELOPING EFFECTIVE DOCUMENTATION

Documenting your project helps reveal issues and gaps in the analysis and design. A document can serve as a communication vehicle among the project's team members, or it can serve as an initial understanding of the requirements. Blum concludes that management has responsibility for resources such as software, hardware, and operational expenses.

In many projects, documentation can be an important factor in making a decision about committing resources. Application software is expected to provide a solution to a problem. It is very difficult, if not impossible, to document a poorly understood problem. The main issue in documentation during the analysis phase is to determine what the system must do. Decisions about how the system works are delayed to the design phase. Blum raises the following questions for determining the importance of documentation: How will a document be used? (If it will not be used, it is not necessary.) What is the objective of the document? What is the management view of the document? Who are the readers of the document?

1. Organization Conventions for Documentation

The documentation depends on the organization's rules and regulations. Most organizations have established standards or conventions for developing documentation. However, in many organizations, the standards border on the nonexistent. In other cases, the standards may be excessive. Too little documentation invites disaster; too much documentation, as Blum put it, transfers energy from the problem-solving tasks to a mechanical and unrewarding activity. Each organization determines what is best for it, and you must respond to that definition and refinement.

Bell and Evans provide us with guidelines and a template for preparing a document that has been adapted for documenting the unified approach's systems development Remember that your modeling effort becomes the analysis, design, and testing documentation. However, this template which is based on the unified approach lifecycle assists you in organizing and composing your models into an effectivedocumentation.

2. Guidelines for Developing Effective Documentation

Bell and Evans provide us the following guidelines for making documents fit theneeds and expectations of your audience:

1. **1.Common cover.** All documents should share a common cover sheet that identifiesthe document, the current version, and the individual responsible for the content. As thedocument proceeds through the life cycle phases, the responsible individual may change. That change must be reflected in the cover sheet.
2. **2.80-20 rule**. As for many applications, the 80-20 rule generally applies for documentation: 80 percent of the work can be done with 20 percent of the documentation. The trick is to make sure that the 20 percent is easily accessible and the rest (80 percent) is available to those (few) who need to know.
3. **Familiar vocabulary.** The formality of a document will depend on how it is used and who will read it. When developing a documentation use a vocabulary that your readers understand and are comfortable with. The main objective here is to communicate with readers and not impress them with buzz words.
4. **Make the document as short as possible**. Assume that you are developing a manual. The key in developing an effective manual is to eliminate all repetition; present summaries, reviews, organization chapters in less than three s; and make chapter headings task oriented so that the table of contents also could serve as an index.
5. **Organize the document**. Use the rules of good organization (such as the organization’s standards, college handbooks, Strunk and White's Elements of Style or the University of Chicago Manual of Style) within each section. Appendix A provides a template for developing documentation for a project. Most CASE tools provide documentation capability by providing customizable reports. The purpose of these guidelines is to assist you in creating an effective documentation.

(Document Name)
For
(Product)
(Version no)

Responsible individual
Name:
Title:

Fig. 3.7. Cover Sheet template.

3.7 Case Study: ANALYSING THE VIANET BANK ATM – THE USE CASE DRIVEN PROCESS

The Following section provides the description of the vianet bank atm system's requirement.

- The Bank client must be able to deposit an amount to and withdraw an amount fromhis or her accounts using the touch screen at the vianet bank atm. Each transaction must be recorded, and the client must be able to review all transactions performed against the given account. Recorded transactions must include the date, time, Transaction type,amount and account balance after the transactions.
- A ViaNet bank client can have two types of accounts: a checking account and savings account. For each checking account, one related savings account can exist.
- Access to the ViaNet bank accounts is Provided by a PIN code consisting of four integerdigits between 0 and 9.
- One PIN code allows access to all accounts held by a bankclient.
- No receipts will be provided for any account transactions.
- The bank application operates for a single banking institution only.
- Neither a checking nor a savings account can have a negative balance. The system should automatically withdraw money from a related savings account if the requested withdrawal amount on the checking account is more than its current balance. If the balance on a savings account is less than the withdrawal amount requested, the transaction will stop and the bank client will be notified.

i) Identifying Actors and Use Cases for the ViaNet Bank ATM System

The bank application will be used by one category of users: bank clients. Notice that identifying the actors of the system is an iterative process and can be modified as you learn more about the system. The actor of the bank system is the bank client. The bank client must be able to deposit an amount to and withdraw an amount from his or her accounts using the bank application. The following scenarios show use-case interactions between the actor (bank client) and the bank. In real life application these use cases are created by system requirements, examination of existing system documentation, interviews, questionnaire, observation, etc.

- Use-case name: Bank ATM transaction. The bank clients interact with the bank system by going through the approval process. After the approval process, the bank clientcan perform the transaction. Here are the steps in the ATM transaction use case:
  1. Insert ATM card.
  2. Perform the approval process.
  3. Ask type of transaction.
  4. Enter type of transaction.
  5. Perform transaction.
  6. Eject card.
  7. Request take card.
  8. Take card.

These steps are shown in the Figure activity diagram. .

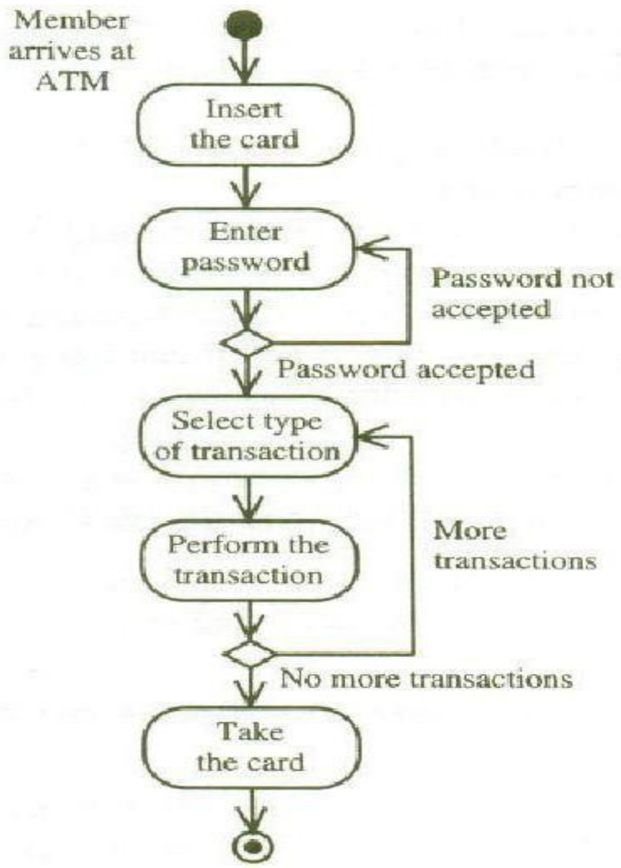


Fig. 3.8. Activities involved in an ATM transaction

- **Use-case name:** Approval process. The client enters a PIN code that consists of 4digits.Activities involved in an ATM transaction.
  1. Request password.
  2. Enter password.
  3. Verify password.

- **Usecase name:** Invalid PIN. If the PIN code is not valid, an appropriate message is displayed to the client. This use case extends the approval process. (See Figure.)
- **Usecase name:** Deposit amount. The bank clients interact with the bank system after the approval process by requesting to deposit money to an account. The client selects the account for which a deposit is going to be made and enters an amount in dollar currency. The system creates a record of the transaction. (See Figure) This usecase extends the bank ATM transaction use case. Here are the steps:

1. Request account type.
2. Request deposit amount.
3. Enter deposit amount.
4. Put the check or cash in the envelope and insert it into ATM.

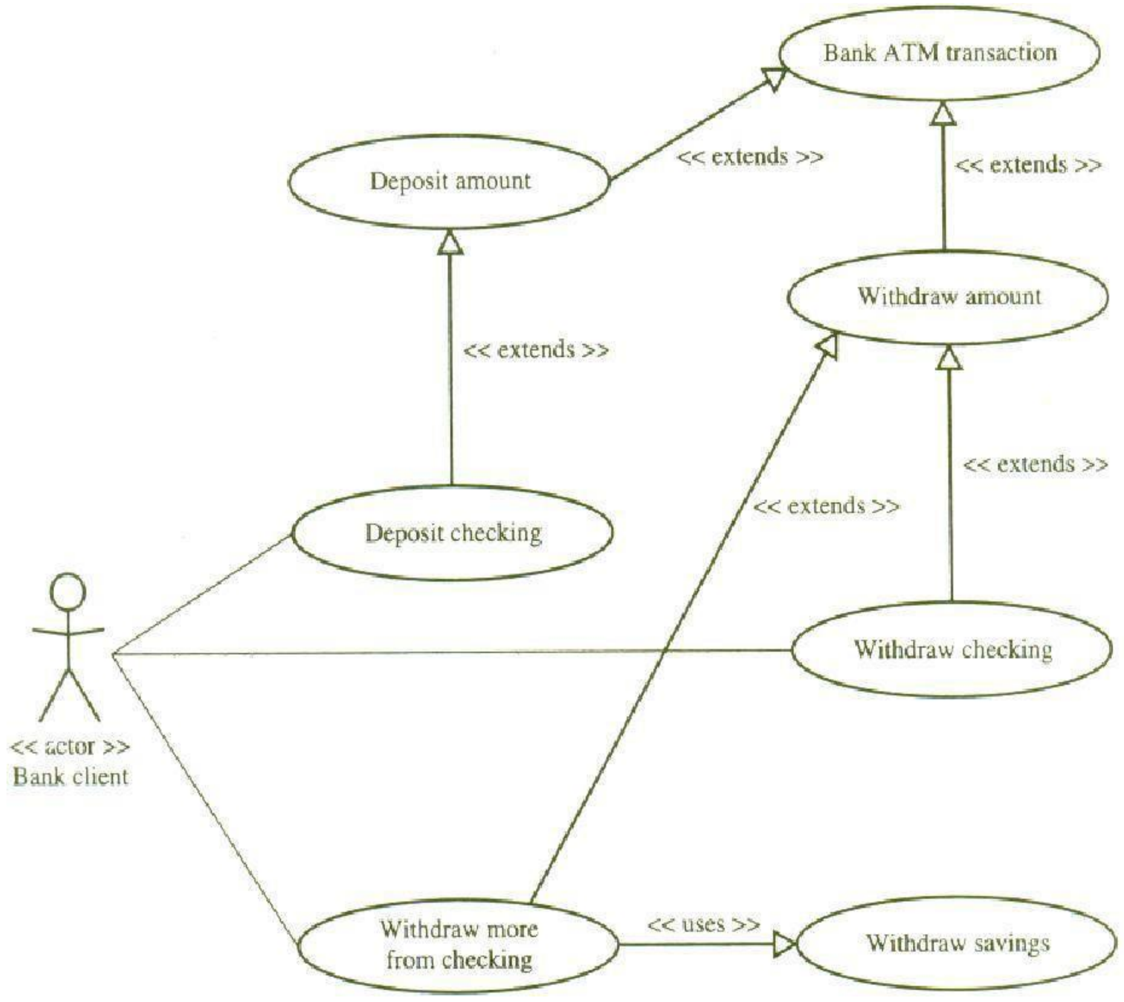


Fig. 3.9. Transaction use cases

- **Usecase name:** Deposit savings. The client selects the savings account for which a deposit is going to be made. All other steps are similar to the deposit amount use case. The system creates a record of the transaction. This use case extends the deposit amount use case. (See Figure 6-11.)

- **Usecase name:** Withdraw checking. The client tries to withdraw an amount from his or her checking account. If the amount is more than the checking account's balance, the insufficient amount is withdrawn from the related savings account. The system creates a record of the transaction and the withdrawal is successful. This use case extends the withdraw checking use case and uses the withdraw savings use case. (See Figure)
- **Usecase name**: Withdraw savings. The client tries to withdraw an amount from a savings account. The amount is less than or equal to the balance and the transaction is performed on the savings account. The system creates a record of the transaction since the withdrawal is successful. This use case extends the withdraw amount use case.

- **Usecase name:** Withdraw savings denied. The client withdraws an amount from a savings account. If the amount is more than the balance, the transaction is halted and a message is displayed. The savings account use-cases package. This use case extends the bank transaction use case. (See Figure 3.10))
- **Usecase name:** Savings transaction history. The bank client requests a history of transactions for a savings account. The system displays the transaction history for the savings account. This use case extends the bank transaction use case. (See Figure)

The use-case list contains at least one scenario of each significantly different kind of use-case instance. Each scenario shows a different sequence of interactions between actors and the system, with all decisions definite. If the scenario consists of an if statement, for each condition create one scenario.

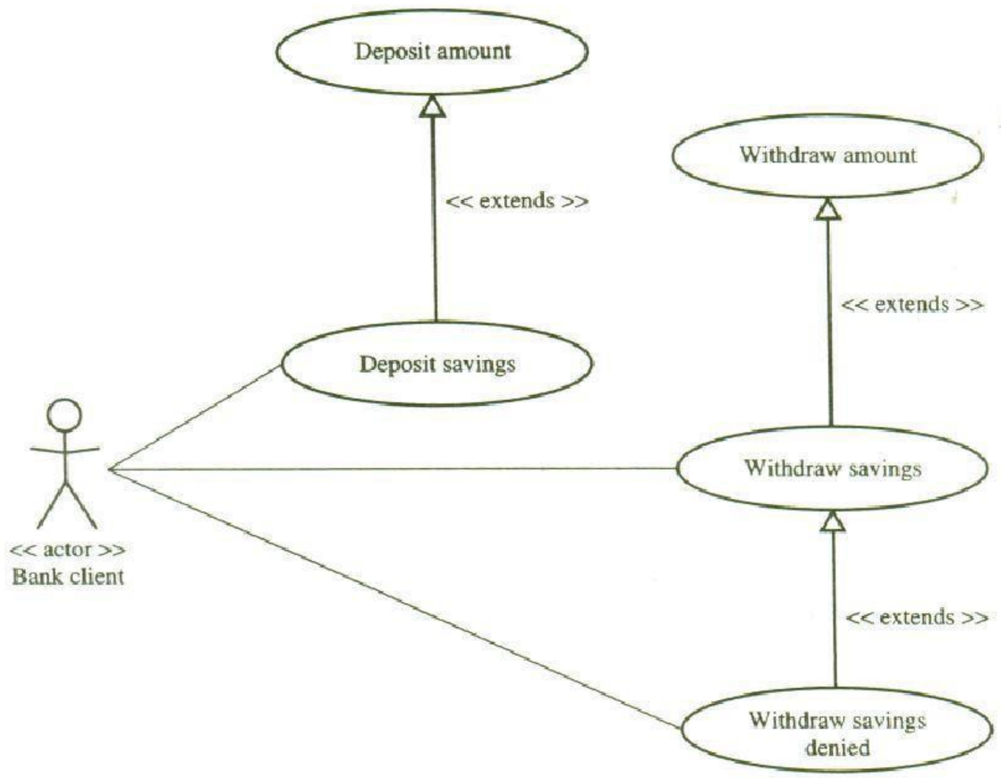


Fig. 3.10. The checking account use-cases

OBJECT ANALYSIS: CLASSIFICATION

3.8 CLASSIFICATIONS THEORY

Classification, the process of checking to see if an object belongs to a category or a class, is regarded as a basic attribute of human nature.

Booch explains that, intelligent classification is part of all good science. Classification guides us in making decisions about modularization. We maychoose to place certain classes and objects together in the same module or in different modules, depending upon the sameness we find among these declarations; coupling and cohesion are simply measures of this sameness. Classification also plays a role in allocating processes to procedures. We place certain processes together in the same processor or different processors, depending upon packaging, performance, or reliability concerns.

Human beings classify information every instant of their waking lives. We recognize the objects around us, and we move and act in relation to them. A human being is sophisticated information system, partly because he or she possesses a superior classification capability. For example, when you see a new model of a car, you have no trouble identifying it as a car. What has occurred here, even though you may never have seen this particular car before, is that you not only can immediately identify it as a car, but you also can guess the manufacturer and model. Clearly, you have some general idea of what cars look like, sound like, do, and are good for-you have a notion of car-kind or, in object-oriented terms, the class car.

Classes are an important mechanism for classifying objects. The chief role of class is to define the attributes, methods, and applicability of its instances. The class car, for example, defines the property color. Each individual car (formally, each instance of the class car) will have a value for this property, such as maroon, yellow, or white. It is early natural to partition the world into objects that have properties (attributes)and methods (behaviors). It is common and useful partitioning or classification, but we also routinely divide the world along a second dimension: We distinguish classes from instances.

A class is a specification of structure, behavior, and the description of an object. Classification is concerned more with identifying the class of an object than the individual objects within a system.

The problem of classification may be regarded as one of discriminating things,not between the individual objects but between classes, via the search for features or invariant attributes or behaviors among members of a class.

Classification can be defined as the categorization of input data (things) into identifiable classes via the extraction of significant features of attributes of the data from a background of irrelevant detail.

Another issue in relationships among classes is studied.

3.9 APPROACHES FOR IDENTIFYING CLASSES

In the following sections, we look at four alternative approaches for identifying classes:

1. The **Noun Phrase** approach;
2. The **Common Class Patterns** approach;
3. The **Usecase Driven, Sequence/Collaboration Modelling** approach;
4. The **Classes, Responsibilities, and Collaborators (CRC)** approach.

The first two approaches have been included to increase your understanding of the subject; the unified approach uses the use-case driven approach for identifying classes and understanding the behavior of objects. However, you always can combine these approaches to identify classes for a given problem.

Another approach that can be used for identifying classes is Classes, Responsibilities, and Collaborators (CRC) developed by Cunningham, Wilkerson, and Beck.

Classes, responsibilities, and Collaborators, more technique than method, is used for identifying classes responsibilities and therefore their attributes and methods.

3.10. NOUN PHRASE APPROACH

The noun phrase approach was proposed by **Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener.** In this method, you read through the requirements or use cases looking for nounphrases. **Nouns in the textual description** are considered to be **classes and verbs.to bemethods of the classes**

All plurals are changed to singular, the nouns are listed, and the list divided into three categories (see Figure 3.11): **relevant classes, fuzzy classes** (the "fuzzyarea," classes we are not sure about), **and irrelevant classes**.

It is safe to scrap the irrelevant classes, which either have no purpose or will be unnecessary. Candidate classes then are selected from the other two categories. Keep in mind that identifying classes and developing a UML class diagram just like other activities is an iterative process. Depending on whether such object modeling is for the analysis or design phase of development, some classes may need to be added or removed from the model and, remember, flexibility is a virtue. You must be able to formulate a statement of purpose for each candidate class; if not, simply eliminate it.

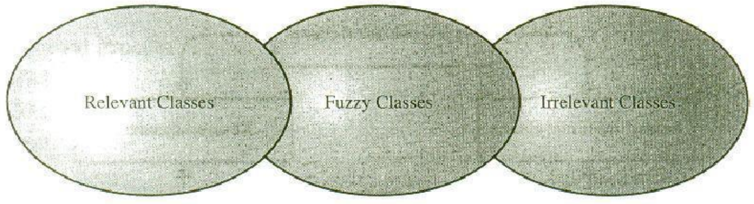


Fig. 3.11. Using the noun phrase strategy, candidate classes can be divided into 3categories.

i) Identifying Tentative Classes

The following are **guidelines** for selecting classes in an application: .

- Look for nouns and noun phrases in the use cases. .
- Some classes are implicit or taken from general knowledge.

- All classes must make sense in the application domain; avoid computer implementation classes-defer them to the design stage.
- Carefully choose and define class names.

ii) Selecting Classes from the Relevant and Fuzzy Categories

The following **guidelines help** in selecting candidate classes from the relevant and fuzzy categories of classes in the problem domain.

- **Redundant classes**. Do not keep two classes that express the same information. If more than one word is being used to describe the same idea, select the one thatis the most meaningful in the context of the system. This is part of building a common vocabulary for the system as a whole. Choose your vocabulary carefully; use the word that is being used by the user of the system.
- **Adjectives classes**. Adjectives can be used in many ways. An adjective can suggest a different kind of object, different use of the same object, or it could be utterly irrelevant. Does the object represented by the noun behave differently when the adjective is applied to it? If the use of the adjective signals that the behavior of the object is different, then make a new class". For example, Adult Members behave differently than Youth Members, so, the two should be classified as different classes.
- **Attribute classes**. Tentative objects that are used only as values should be defined or restated as attributes and not as a class. For example, Client Status and Demographic of Client are not classes but attributes of the Client class.
- **Irrelevant classes**. Each class must have a purpose and every class should be clearly defined and necessary. You must formulate a statement of purpose for each candidate class. If you cannot come up with a statement of purpose, simply eliminate the candidate class.

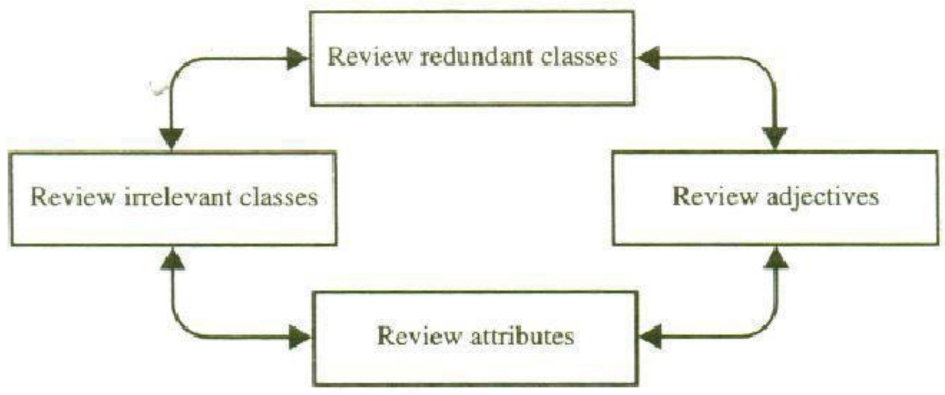


Fig. 3.12. The process of eliminating the redundant classes and refining the remaining classes is not sequential. You can move back and forth among these steps as often as you like.

Example: The ViaNet Bank ATM System: Identifying Classes by Using Noun Phrase Approach

To better understand the noun phrase method, we will go through a case and apply the noun phrase strategy for identifying the classes.

Initial List of Noun Phrases: Candidate Classes

The initial study of the use cases of the bank system produces the following noun phrases (candidate classes-maybe).

Account Balance
AmountApproval
Process ATM
Card
ATM Machine
Bank
Bank Client
Card
Cash
Check
Checking
Checking Account
Client
Client's Account
Currency Dollar
Envelope

Four Digits
Fund

Invalid PIN
Message
Money
Password
PIN

PIN Code
Record
Savings
Savings Account
Step.
System Transaction
History

It is safe to eliminate the irrelevant classes. The candidate classes must be selected from relevant and fuzzy classes. The following **irrelevant classes** can be eliminated because they do not belong to the problem statement: Envelope, Four Digits, and Step. Strikeouts indicate eliminated classes.

Account Balance
AmountApproval
Process ATM
Card
ATM Machine
Bank
BankClient
Card
Cash
Check
Checking
Checking Account
Client
Client's Account
Currency Dollar
~~Envelope~~

~~Four Digits~~
Fund
Invalid PIN
Message
Money,
Password
PIN

PIN Code
Record
Savings
Savings Account

~~Step~~
System
Transaction -
Transaction History

Reviewing the Redundant Classes and Building a Common Vocabulary

We need to review the candidate list to see which classes are redundant. Ifdifferent words are being used to describe the same idea, we must select the one that isthe most **meaningful the context of the system and eliminate the others.** The following are the different class names that are being used to refer to the same concept:

Client, BankClient Account, Client's Account PIN, PIN Code
Checking, Checking Account = BankClient (the term chosen)
Checking Account = Account
Checking Account = PIN

Checking Account = Checking Account
Savings, Savings Account = Savings Account
Fund, Money = Fund
ATM Card, Card = ATM Card
Here is the revised list of candidate classes:
Account

Account Balance
Amount Approval
Process ATM
Card

Bank
BankClient
~~Card~~
Cash
Check
~~Checking~~
Checking Account
~~Client~~
~~Client's account~~
Currency
Dollar
~~Envelope~~
~~Fund digits~~
Fund
Invalid PIN
Message
Message~~M~~
~~oney~~
Password
PIN

~~PIN Code~~
Record
~~Savings~~
Savings Account
~~Step~~
System
Transaction
Transaction History

Reviewing the Classes Containing Adjectives

We again review the remaining list, now with an eye on classes with adjectives. The main question is this: Does the object represented by the noun behave differently when the adjective is applied to it? If an adjective suggests a different kind of class or the class represented by the noun behaves differently when the adjective is applied to it, then we need to make a new class. However (it is a different use of the same object or the classis irrelevant, we must eliminate it) In this example, we have no classes containingadjectives that we can eliminate.

Reviewing the Possible Attributes

The next review focuses on identifying the noun phrases that are attributes, not classes. The noun phrases used only as values should be restated as attributes. This process also will help us identify the attributes of the classes in the system.

Amount: a value, not a class.

Account Balance: An attribute of the Account class.
Invalid PIN: It is only a value, not a class.

Password: An attribute, possibly of the BankClient class.
Transaction History: An attribute, possibly of the Transaction class.
PIN: An attribute, possibly of the BankClientclass.

Here is the revised list of candidate classes. Notice that the eliminated classes are strikeouts (they have a line through them).

Account ~~Account~~
~~Balance Amount~~
Approval Process
ATM Card
Bank
BankClient
Cash
~~Card~~
Check
~~Checking~~
Checking Account
Currency
Dollar

~~Envelope~~
~~Fund digits~~
Fund
Message
MaBey
PIN

~~PIN Code~~
Record
Savings Account
System
~~step~~Transa
ction
~~Transaction History~~

Reviewing the Class Purpose

Identifying the classes that play a role in achieving system goals and requirements is a major activity of object-oriented analysis) Each class must have a purpose. Every class should be clearly defined and necessary in the context of achieving the system's goals. If you cannot formulate a statement of purpose for a class, simply eliminate it. The classes that add no purpose to the system have been deleted from the list. The candidate classes are these:

- **ATM Machine class:** Provides an interface to the ViaNet bank.
  ATMCard class: Provides a client with a key to an account.
- **BankClient class:** A client is an individual that has a checking account and,possibly, a savings account.
- **Bank class:** Bank clients belong to the Bank. It is a repository of accounts and processes the accounts' transactions.
- **Account class:** An Account class is a formal (or abstract) class, it defines the common behaviors that can be inherited by more specific classes such as CheckingAccount and SavingsAccount.
- **CheckingAccount class:** It models a client's checking account and provides more specialized withdrawal service.
- **savingsAccount class:** It models a client's savings account.
- **Transaction class:** Keeps track of transaction, time, date, type, amount, and 'balance.

3.11 COMMON CLASS PATTERN APPROACH

The second method for identifying classes is using *common class patterns*, which is **based on a knowledge base of the common classes that have been proposed by various researchers, such as Shlaer and Mellor [10], Ross[8], and Coad and Yourdon [3].** They have compiled and listed the following patterns for finding the candidate class and object :

- Name.**Concept class**

Context: A concept is a particular idea or understanding that we have of our world. The concept class encompasses principles that are not tangible but used to organize or keep track of business activities or communications. Marin and Odell describe concepts elegantly," Privately held ideas or notions are called **conceptions**. When an understanding is shared by another, it becomes a **concept.** To **communicate** with others, we must share our individually held conceptions and arrive at agreed concepts." Furthermore, Martin and Odell explain that, without concepts, mental life would be total chaos since every item we encountered wouldbe different.
*Example.*Performance is an example of concept class object.

- Name.**Events class**

Context: Events classes are **points in time that must be recorded**. Things happen, usually to something else at a given date and time or as a step in an ordered sequence. Associated with things remembered are attributes (after all, the things to remember are objects) such as who, what, when, where, how, or why.
Example. Landing, interrupt, request, and order are possible events.

- Name. ***Organization class***

Context:. An organization class is a **collection of people, resources, facilities, or groups to which the users belong; their capabilities** havea defined mission, whose existence is largely independent of the individuals.

Example. An accounting department might be considered a potential class.

- Name. ***People class*** (also known as person, roles, and roles played class)

Context. The people class **represents the different roles users play in interacting with the application**. People carry out some function. What roles does a personplay in the system? Coad and Yourdon [3] explain that a class which is represented by a person can be divided into two types: those representing users of the system, such as an operator or clerk who interacts with the system; and those representing people who donot use the system but about whom information is kept by the system.

Example. Employee, client, teacher, and manager are examples of people.

- Name. ***Places class***

Context. Places are **physical locations** that the system must keep information about.
Example. Buildings, stores, sites, and offices are examples of places.

- Name. ***Tangible things and devices class***

Context. This class **includes physical objects or groups of objects** that are tangible and devices with which the application interacts.

Example. Cars are an example of tangible things, and pressure sensors are an example of devices.

3.12 USE-CASE DRIVEN APPROACH: IDENTIFYING CLASSES AND THEIR BEHAVIORS THROUGH SEQUENCE/COLLABORATION MODELING

The use cases are employed to model the scenarios in the system and specify what external actors interact with the scenarios. The scenarios are described in text or through a sequence of steps. Use-case modeling is considered a problem-driven approach to object-oriented analysis, in that the designer first considers the problem at hand and not the relationship between objects, as in a data-driven approach.

Modeling with use cases is a recommended aid in finding the objects of a system and is the technique used by the unified approach. Once the system has been described in terms of its scenarios, the modeler can examine the textual description or steps of each scenario to determine what objects are needed for the scenario to occur. However, this is not a magical process in which you start with use cases, develop a sequence diagram, and voila, classes appear before your eyes.

The process of creating sequence or collaboration diagrams is a systematic way to think about how a use case (scenario) can take place; and by doing so, it forces you to think about objects involved in your application.

When building a new system, designers model the scenarios of the way the system of business should work. When redesigning an existing system, many modelers choose to first model the scenarios of the current system, and then model the scenarios of the way the system should work.

i) Implementation Of Scenarios

The UML specification recommends that at least one scenario be prepared for each significantly different use-case instance. Each scenario shows a different sequenceof interaction between actors and the system, with all decisions definite. In essence, this process helps us to understand the behavior of the system's objects.

When you have arrived at the lowest use-case level, you may create a child sequence diagram or accompanying collaboration diagram for the use case. With the sequence and collaboration diagrams, you can model the implementation of the scenario.

Like use-case diagrams, sequence diagrams are used to model scenarios in the systems. Whereas use cases and the steps or textual descriptions that define them offer a high-level view of a system, the sequence diagram enables you to model a more specific analysis and also assists in the design of the system by modeling the interactions between objects in the system.

As explained in a sequence diagram, the objects involved are drawn on thediagram as a vertical dashed line, with the name of the objects at the top. Horizontal lines corresponding to the events that occur between objects are drawn between the vertical object lines. The event lines are drawn in sequential order, from the top of the diagram to the bottom. They do not necessarily correspond to the steps defined for a usecase scenario.

CASE STUDY: THE VIANET BANK ATM SYSTEM: DECOMPOSING

Scenario with a Sequence Diagram: Object Behavior Analysis A sequence diagram represents the sequence and interactions of a given use case or scenario. Sequence diagrams are among the most popular UML diagrams and, if used with an object model or class diagram, can capture most of the information about a system. Most object-to-object interactions and operations are considered events, and events include signals, inputs, decisions, interrupts, transitions, and actions to or from users or external devices. An event also is considered to be any action by an object that sends information. The event line represents a message sent from one object to another, in which the "from" object is requesting an operation be performed by the "to" object. The "to" object performs the operation using a method that its class contains. Developing sequence or collaboration diagrams requires us to think about objects that generate these events and therefore will help us in identifying classes.

To identify objects of a system, we further analyze the lowest level use cases with a sequence and collaboration diagram pair (actually, most CASE tools such as SA/Object allow you to create only one, either a sequence or a collaboration diagram, and the system generates the other one). Sequence and collaboration diagrams represent the order in which things occur and how the objects in the system send messages to one another.

These diagrams provide a macro-level analysis of the dynamics of a system. Once you start creating these diagrams, you may find that objects may need to be added to satisfy the particular sequence of events for the given use case.

You can draw sequence diagrams to model each scenario that exists when a BankClient withdraws, deposits, or needs information on an account. By walking through the steps, you can determine what objects are necessary for those steps to take place. Therefore, the process of creating sequence or collaboration diagrams can assist you in Identifying Classes or objects of the system. This approach can be combined with noun phrase and class categorization for the best results. We identified the use cases for the bank system. The following are the low level (executable) use cases:

Deposit Checking
Deposit Savings
Invalid PIN
Withdraw Checking
Withdraw More fromChecking
Withdraw Savings
Withdraw Savings Denied
Checking Transaction History
Savings Transaction History

Let us create a sequence/collaboration diagram for the following use cases:

- Invalid PIN use case
- Withdraw Checking use case
- Withdraw More from Checking use case

Sequence/collaboration diagrams are associated with a use case. For example, to model the sequence/collaboration diagrams in SA/Object, you must first select a use case, such as the Invalid PIN use case, then associate a sequence or collaboration child process.

To create a sequence, you must think about the classes that probably will be involved in a use-case scenario. Keep in mind that use case refers to a process, not a class. However, a use case can contain many classes, and the same class can occur in many different use cases. Point of caution: you should defer the interfaces classes to the design phase and concentrate on the identifying business classes here. Consider how we would prepare a sequence diagram for the Invalid PIN use case. Here, we need to think about the sequence of activities that the actor BankClient performs:

- Insert ATM Card.
- Enter PIN number.
- Remove the ATM Card.

Based on these activities, the system should either grant the access right to the account or reject the card. Next, we need to more explicitly define the system. With what are we interacting? We are interacting with an ATMMachine and the BankClient. So, the other objects of this use case are ATMMachine and BankClient.

Now that we have identified the objects involved in the use case, we need to list them in a line along the top of a and drop dotted lines beneath each object (see Figure 3.13). The client in this case is whoever tries to access an account through the ATM, and may or may not have an account. The BankClient on the other hand has an account.

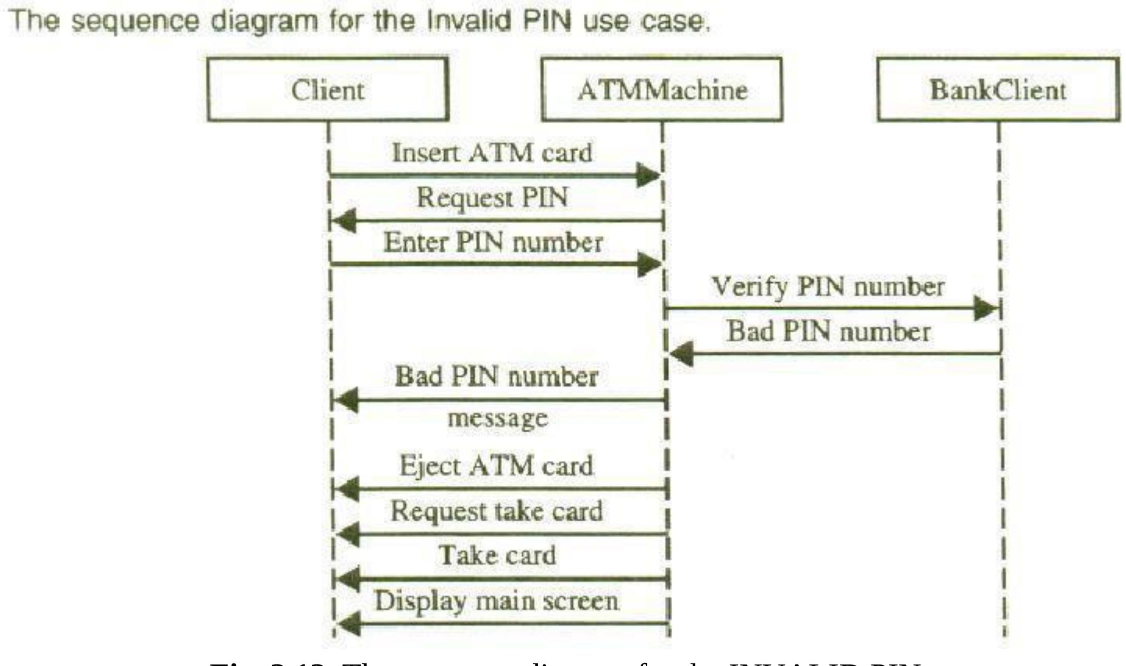


Fig. 3.13. The sequence diagram for the INVALID PIN use case

The dotted lines are the lifelines. The line on the right represents an actor, in this case the BankClient, or an event that is outside the system boundary. An event arrow connects objects. In effect, the event arrow suggests that a message is moving between those two objects. An example of an event message is the request for a PIN. An event line can pass over an object without stopping at that object. Each event must ha"\'e'a descriptive name. In some cases, several objects are active simultaneously, even if they are only waiting for another object to return information to them. In other cases, an object becomes active when it receives a message and then becomes inactive as soon as it responds. Similarly, we can develop sequence diagrams for other use cases (as in Figures 3.14 and 3.16). Collaboration diagrams are just another view of the sequence diagrams and therefore can be created automatically; most UML modeling tools automatically create them (see Figures 3.15)

The following classes have been identified by modeling the UML sequence / collaboration diagrams: Bank, BankClient, ATMMachine, Account, Checking Account, and Savings Account. Similarly other classes can be identified by developing the remaining sequence/ collaboration diagrams.

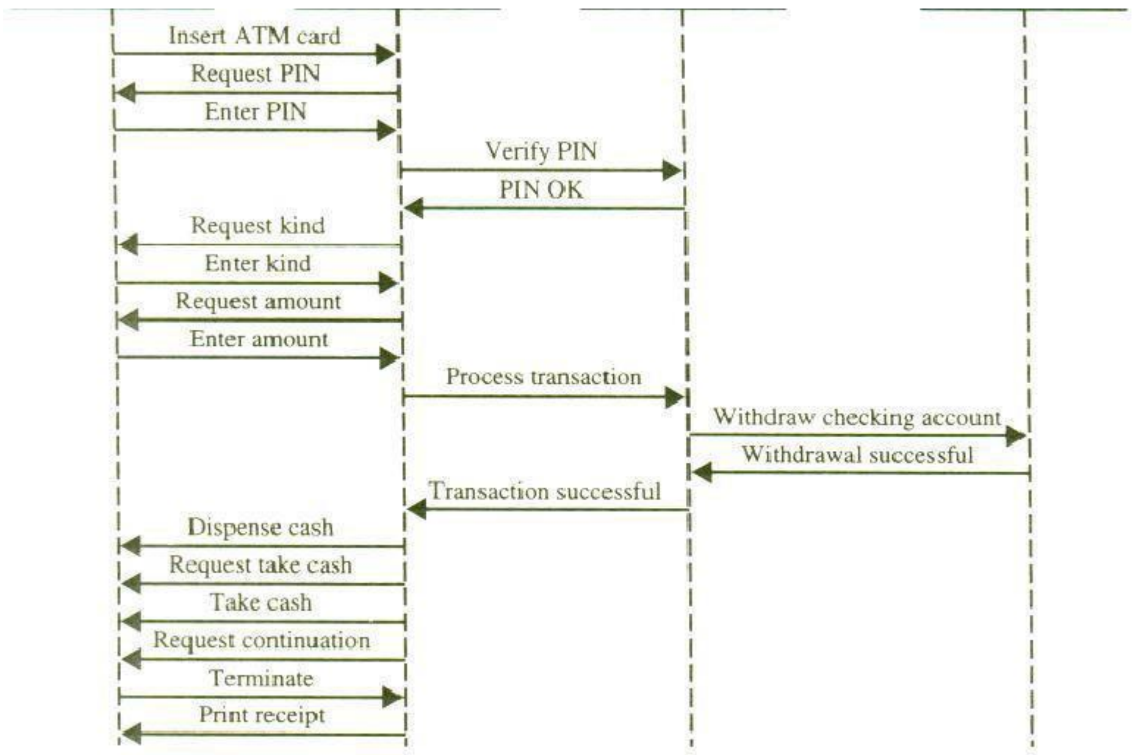


Fig. 3.14. Sequence Diagram for the Withdraw Checking use case

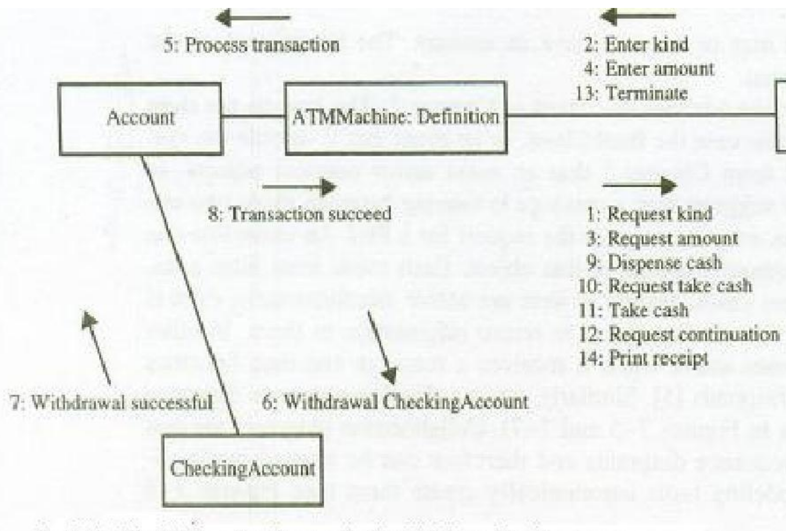
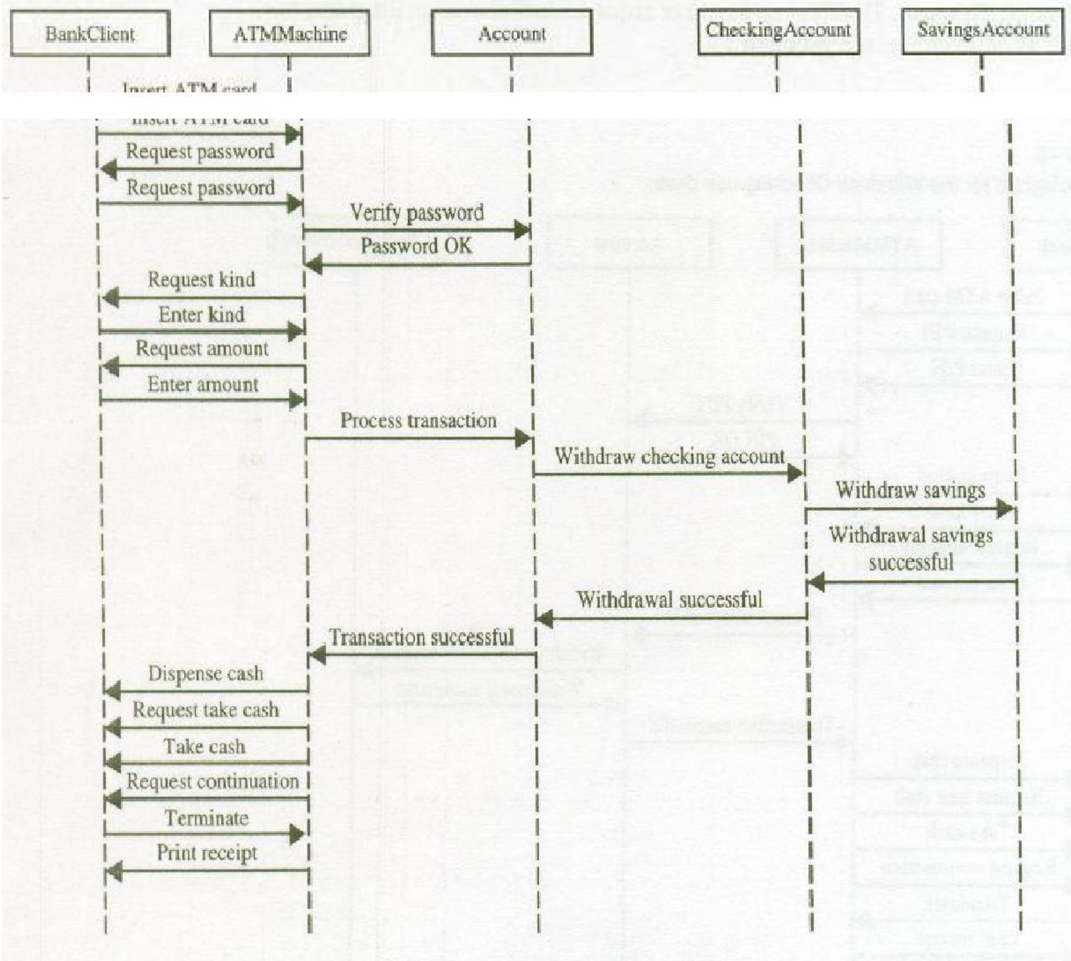


Fig 3.15 : The Collaboration diagram for the Withdraw Checking use case

Fig 3.16
The sequence diagram for the Withdraw More from Checking use case.



3.13 CLASSES, RESPONSIBILITIES, AND COLLABORATORS (CRC) APPROACH

Classes, responsibilities, and collaborators (CRC), developed by Cunningham, Wilkerson, and Beck, was first presented as a way of teaching the basic concepts of object-oriented development.

Classes, Responsibilities, and Collaborators is a technique used for identifying classes' responsibilities and therefore their attributes and methods.

Furthermore, Classes, Responsibilities, and Collaborators can help us identify classes. Classes, Responsibilities, and Collaborators is more a teaching technique than a method for identifying classes.

Classes, Responsibilities, and Collaborators is based on the idea that an object either can accomplish a certain responsibility itself or it may require the assistance of other objects. It requires the assistance of other objects; it must collaborate with those objects to fulfill its responsibility. By identifying **an object's responsibilities** and **collaborators** (cooperative objects with which it works) you can identify its attributes andmethods.

Classes, Responsibilities, and Collaborators cards are 4" X 6" index cards. All the information for an object is written on a card, which is cheap, portable, readily available, and familiar. Figure 3.17 shows an idealized card.

The class name should appear in the upper left-hand corner, a bulleted list of responsibilities should appear under it in the left two thirds of the card, and the list of collaborators should appear in the right third.

Classes, Responsibilities, and Collaborators cards place the designer's focus on the motivation for collaboration by representing (potentially) many messages as phrases of English text.

.

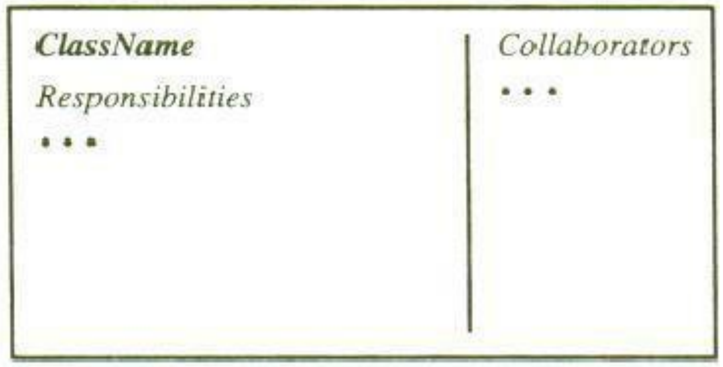


Fig. 3.17. A Classes, Responsibilities and Collaborators (CRC) Index Card

CRC starts with only one or two obvious cards. If the situation calls for a responsibility not already covered by one of the objects: Add, or create a new object to address that responsibility.

– Finding classes is not easy.

– The more practice you have, the better you get at identifying classes.

– There is no such thing as the ―right set of classes.

– Finding classes is an incremental and iterative process.

i) Classes, Responsibilities, And Collaborators Process

The Classes, Responsibilities, and Collaborator's process consists of three steps(Figure 3.17)
1. Identify classes' responsibilities (and identify classes).
2. Assign responsibilities.
3. Identify collaborators.

Classes are identified and grouped by common attributes, which also provides candidates for super classes. The class names then are written onto Classes, Responsibilities, and Collaborators cards. The card also notes sub- and super classes to show the class structure. The application's requirements then are examined for actions and information associated with each class to find the responsibilities of each class.

Next, the responsibilities are distributed; they should be as general as possible and placed as high as possible in the inheritance hierarchy. The idea in locating collaborators is to identify how classes interact. Classes (cards) that have a close collaboration are grouped together physically.

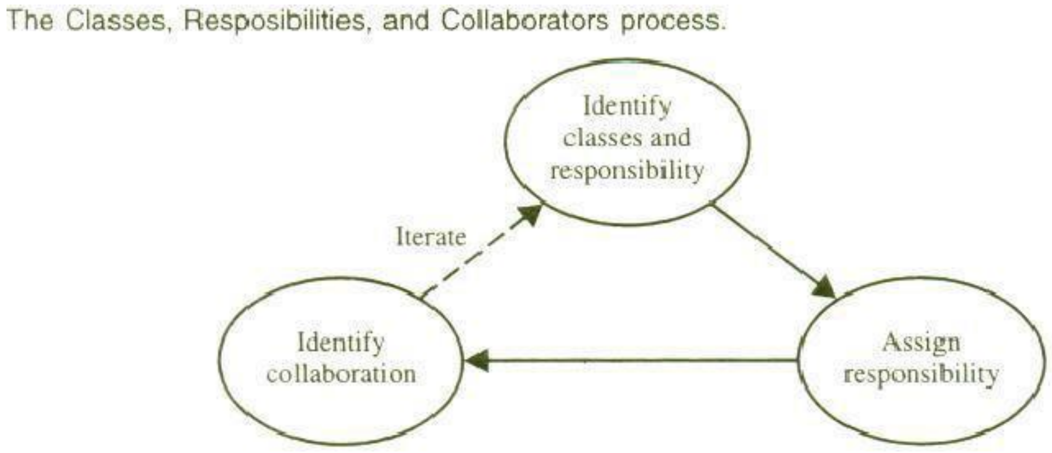

The Classes, Resposibilities, and Collaborators process.
Identify classes and responsibility
Assign responsibility
Identify collaboration
Iterate

Fig. 3.18. The Classes, Responsibilities and Collaborators process.

CASE STUDY: The ViaNet Bank ATM System: Identifying Classes by Using Classes, Responsibilities, and Collaborators

We already identified the initial classes of the bank system. The objective of this example is to identify objects' responsibilities such as attributes and methods in that system. Account and Transaction provide the banking model. Note that Transaction assumes an active role while money is being dispensed and a passive role thereafter. The class Account is responsible mostly to the BankClient class and it collaborates with several objects to fulfill its responsibilities. Among the responsibilities of the Account class to the BankClient class is to keep track of the BankClient balance, account number, and other data that need to be remembered. These are the attributes of the Account class. Furthermore, the Account class provides certain services or methods, such as means for

BankClient to deposit or withdraw an amount and display the account's Balance (see Figure 3.18).

Classes, Responsibilities, and Collaborators for the Account object.

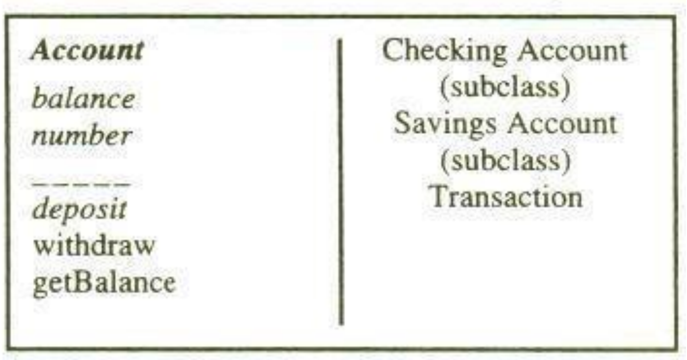


Fig. 3.19. Classes, Responsibilities and Collaborators for the Account Object.

Classes, Responsibilities, and Collaborators encourages team members to pick up the card and assume a role while "executing" a scenario. It is not unusual to see a designer with a card in each hand, waving them about, making a strong identificationwith the objects while describing their collaboration.

As you can see from Figure, this process is iterative.

Start with few cards (classes) then proceed to play "what if." If the situation calls for a responsibility not already covered by one of the objects, either add the responsibility to an object or create a new object to address that responsibility.

3.14 NAMING CLASSES

Naming a class in an important activity.

Guidelines for Naming Classes

• The class should describe a single object, so it should be the singular form of noun.

• Use names that the users are comfortable with.

• The name of a class should reflect its intrinsic nature.

• By the convention, the class name must begin with an upper-case letter.

• For compound words, capitalize the first letter of each word - for example, Loan Window.

IDENTIFYING OBJECT, RELATIONSHIPS, ATTRIBUTES, & METHODS

In an object-oriented environment, objects take on an active role in a system. All objects stand in relationship to others on whom they rely for services and control. The relationship among objects is based on the assumption each makes about the otherobjects, including what operations can be performed and what behavior results. **Three types of relationships among objects are:**

- **Association**: How are objects associated? This information will guide using designing classes.
- **Super-sub structure** (also known as generalization hierarchy): How are objects organized into super classes and subclasses? This information provides us the direction of inheritance.
- **Aggregation and a-part-of structure:** What is the composition of complex classes? This information guides us in defining mechanisms that properly manage object within-object.

Generally, the relationships among objects are known as **associations.**

The **hierarchical or super-sub relation** allows the sharing of properties or inheritance.

A-part-of structure is a familiar means of organizing components of a bigger object.

.

3.15 ASSOCIATIONS

Association represents a physical or conceptual connection between two or more Objects. For example, if an object has the responsibility for telling another object that a credit card number is valid or invalid, the two classes have an association.

we learnt that the **binary associations are shown as lines connecting two class symbols. Ternary and higher-order associations are shown as diamonds connecting to a class symbol by lines, and the association name is written above or below the line. The association name can be omitted if the relationship is obvious.**

Basic association. See Chapter 5 for a detailed discussion of association.

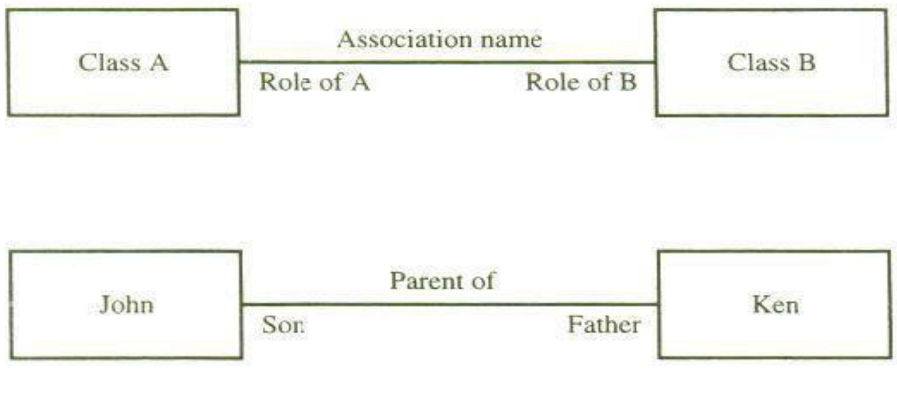


Fig. 3.20. Basic Associations

i) Identifying Associations

Identifying associations begins by analysing the interactions between classes. After all, **any dependency relationship between two or more classes is an association**.

You must examine the responsibilities to determine dependencies.

In other words, **if an object is responsible for a specific task (behavior) and lacks all the necessary knowledge needed to perform the task, then the object must delegate the task to another object that possesses such knowledge.**

Wirfs-Brock, Wilkerson,"'arid Wiener provide the following questions that can help us to identify associations:

- As the class capable of fulfilling the required task by itself?
- If not, what does it need?
- From what other class can it acquire what it needs?

Answering these questions helps us identify association. The approach you should take to identify association is flexibility. **First, extract all candidates' associations from the problem statement and get them down on paper.**

ii) Guidelines For Identifying Association

The Following are general guidelines for identifying the tentative associations:

- A dependency between two or more classes may be an association. Association often corresponds to a verb or prepositional phrase, such as part of, next to, works for, or contained in.
- A reference from one class to another is an association. Some associations are implicit or taken from general knowledge.

iii) Common Association Patterns

The common association patterns are based on some of the common associations defined by researchers and practioners: Rumbaugh et al. Coad and Yourdon, and others.

These include.**Location association** - -next to, part of, contained in. For example, consider a soup object, cheddar cheese is a-part-of soup. The a-part-of relation is a special type of association.

Communication association –talk to, order to. For example, a customer places an order (communication association) with an operator person (see Figure 3.20).

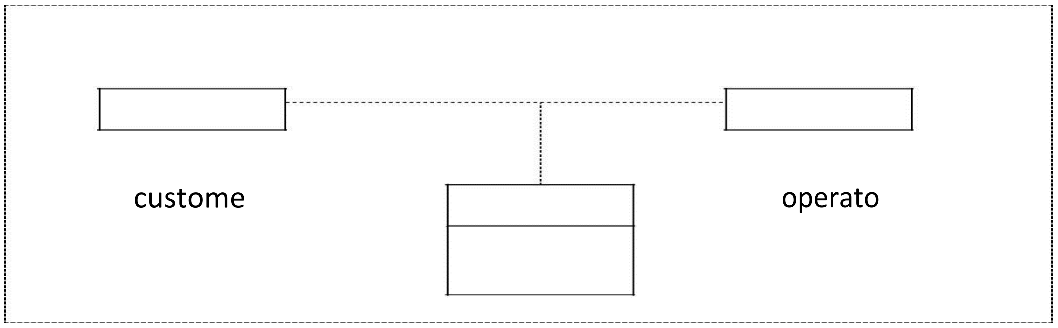


Fig. 3.21. A customer places an order (communication association) with an

operator person.

These association patterns and similar ones can be stored in the repository and added to as more patterns are discovered.

IV) Eliminate Unnecessary Associations

- **Implementation association.** Defer implementation-specific associations to the design phase. Implementation associations are concerned with theimplementation or design of the class within certain programming or developmentenvironments and not relationships among business objects.
- **Ternary associations.** Ternary or n-ary association is an association among more than two classes. Ternary associations complicate the representation. When possible, restate ternary associations as binary associations.

□ **Directed actions (or derived) association**. Directed actions (derived)associations can be defined in terms of other associations. Since they are redundant, avoid these types of association. For example, Grandparent of can be defined in terms ofthe parent of association (see Figure 3.21).

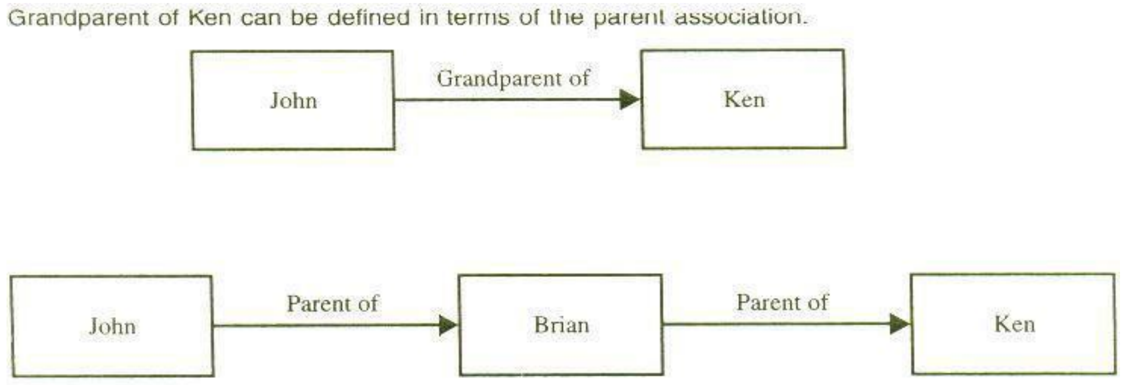

Grandparent of Ken can be defined in terms of the parent association.
John
Grandparent of
Ken
John
Parent of
Brian
Parent of
Ken

Fig. 3.22. Grandparent of Ken can be defined in terms of the parent association.

3.16 SUPER-SUB CLASS RELATIONSHIPS

The other aspect of classification is identification of super-sub relations among classes. For the most part, **a class is part of a hierarchy of classes, where the top classis the most general one and from it descend all other, more specialized classes**.

The super-sub class relationship represents the **inheritance relationshipsbetween related classes, and the class hierarchy determines the lines of inheritance between classes.**

Class inheritance is useful for a number of reasons. For example, in some cases, **you want to create a number of classes that are similar in all but a few characteristics**. In other cases, someone already has developed a class that you can use, but you need to modify that class.

Subclasses are more specialized versions of their superclasses. Superclass-subclass relationships, also known as generalization hierarchy, allow objects to be built from other objects. Such relationships allow us to explicitly take advantage of the commonality of objects when constructing new classes.

The super-sub class hierarchy is a relationship between classes, where one class is the parent class of another (derived) class. The **parent class also is known as the base or super class or ancestor.**

The super-sub class hierarchy is **based on inheritance**, which is programming by extension as opposed to programming by reinvention. The real **advantage of using this technique is that we can build on what we already have and, more important, reuse what we already have. Inheritance allows classes to share and reuse behaviors and attributes. Where the behavior of a class instance is defined in that class's methods**, **a class also inherits the behaviors and attributes of all of its super classes.**

i) <u>Guidelines For Identifying Super-Sub Relationship, A Generalization</u>

The following are guidelines for identifying super-sub relationships in the application:

- **Top-down.** Look for noun phrases composed of various adjectives in a classname. Often, you can discover additional special cases. Avoid excessive refinement. Specialize only when the subclasses have significant behavior. For example, a phone operator employee can be represented as a cook as well as a clerk or manager because they all have similar behaviors.
- **Bottom-up.** Look for classes with similar attributes or methods. In most cases, you can group them by moving the common attributes and methods to an abstract class. You may have to alter the definitions a bit; this is acceptable as long as generalization truly applies.
- **Reusability.** Move attributes and behaviors (methods) as high as possible in the hierarchy. At the same time, do not create very specialized classes at the top of the hierarchy. This is easier said than done. The balancing act can be achieved through several iterations. This process ensures that you design objects that can be reused in another application.
- **Multiple inheritance**. Avoid excessive use of multiple inheritances. Multiple inheritance brings with it complications such as how to determine which behavior to get from which class, particularly when several ancestors define the same method. It also is more difficult to understand programs written in a multiple inheritance system. One way of achieving the benefits of multiple inheritance is to inherit from the most appropriate class and add an object of another class as an attribute. (See fig 3.22)

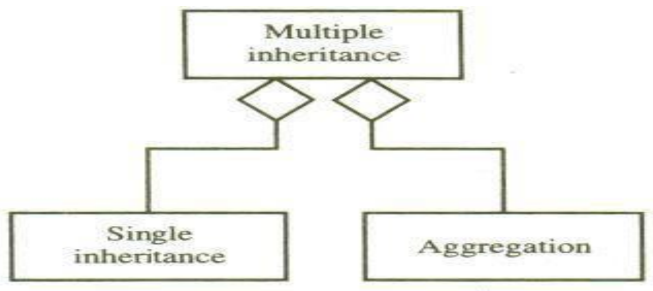


Fig. 3.23. One way of achieving the benefits of multiple inheritance from the most appropriate class.

3.17 A PART OF RELATIONSHIPS-AGGREGATION

A-part-of relationship, also called **aggregation**, represents the situation where class consists of several component classes. A class that is composed of other classes does not behave like its parts; actually, it behaves very differently.

For example, a car consists of many other classes, one of which is a radio, but a car does not behave like a radio (see Figure 3.23).

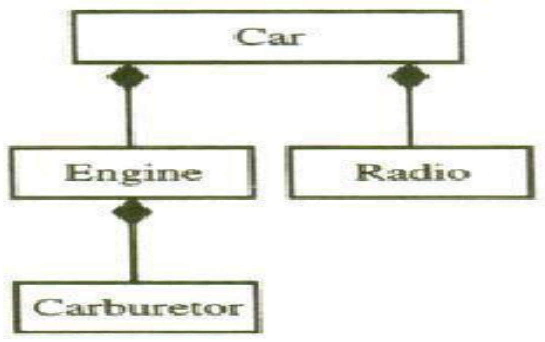


Fig. 3.24. A-part-of composition. A carburetor is a part of an engine and an engine and a radio are parts of a car.

Two major properties of a-part-of relationship are transitivity and antisymmetric,

- **Transitivity**. The property where, if A is part of Band B is part of C, then A is part of C. For example, a carburetor is part of an engine and an engine is part of a car; therefore, a carburetor is part of a car. Figure3.23 shows a-part-of structure.
- **Antisymmetric.** The property of a-part-of relation where, if A is part of B, then . B is not part of A. For example, an engine is part of a car, but acar is not part of an engine.

A clear distinction between the part and the whole can help us determine where responsibilities for certain behavior must reside. This is done mainly by asking the following questions:

- Does the part class belong to a problem domain?
- Is the part class within the system's responsibilities?
- □ Does the part class capture more than a single value? (If it captures only a single value, then simply include it as an attribute with the whole class.)
- Does it provide a useful abstraction in dealing with the problem domain?

We saw that the UML uses hollow or filled diamonds to represent aggregations. A filled diamond signifies the strong form of aggregation, which is composition. For example, one might represent aggregation such as container and collection as hollow diamonds (see Figures 3.24, 3.25) and use a solid diamond to represent composition, which is a strong form of aggregation (see Figure 3.23).

A house is a container.

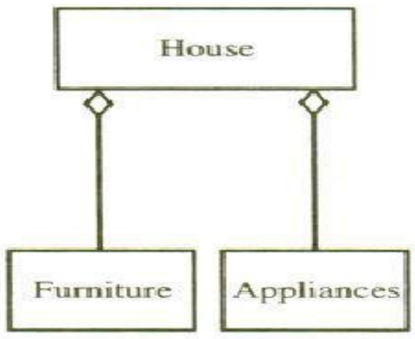


Fig. 3.24. A house is a container

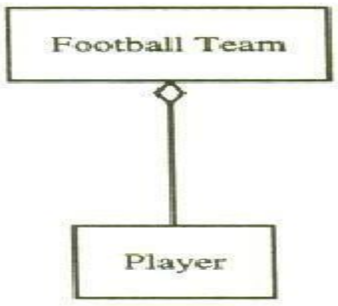


Fig. 3.25. A football team is a collection of players.

i) <u>Part-Of Relationship Patterns</u>

To identify a-part-of structures, Coad and Yourdon provide the following guidelines:

- **Assembly.** An assembly is constructed from its parts and an assembly- part situation physically exists; for example, a French onion soup is an assembly of onion, butter, flour, wine, French bread, cheddar cheese, and so on.

- **Container.** A physical whole encompasses but is not constructed fromphysical parts; for example, a house can be considered as a container for furniture and appliances (see Figure 3.24).
- **Collection-member**. A conceptual whole encompasses parts that may be physical or conceptual; for example, a football team is a collection of players. (fig.3.25).

CASE STUDY: RELATIONSHIP ANALYSIS FOR THE VIANET BANK ATM SYSTEM

1) Identifying Classes' Relationships

One of the strengths of object-oriented analysis is the ability to model objects as they exist in the real world. To accurately do this, you must be able to model more than just an object's internal workings. You also must be able to model how objects relate to each other. Several different relationships exist in the ViaNet bank ATM system, so we need to define them.

2) Developing a UML Class Diagram Based on the Use-Case Analysis

The UML class diagram is the main static analysis and design diagram of a system. The analysis generally consists of the following class diagrams.

- **One class diagram** for the system, which shows the identity and definition of classes in the system, their interrelationships, and various packages containing groupings of classes.

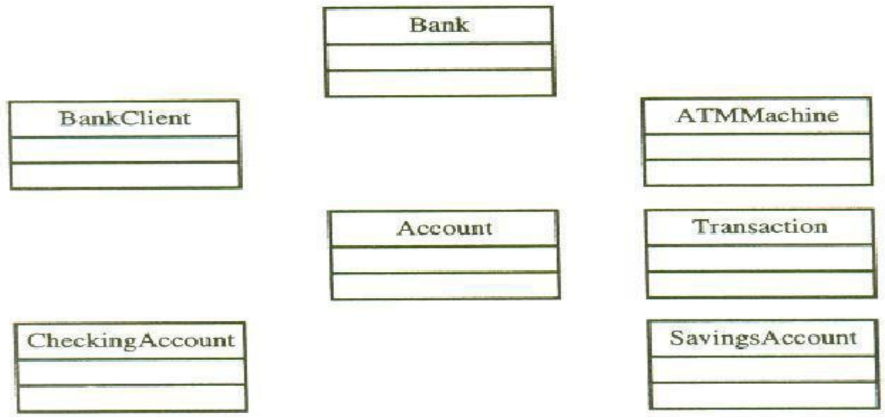


Fig 3.26. UML class diagram for the ViaNet bank ATM system. Some CASE tools such as the SA/Object Architect can automatically define classes and draw them from use cases or collaboration/ sequence diagrams. However, presently, it cannot identifyall the classes. For this example, S/A Object was able to identify only the BankClient class.

- **Multiple class diagrams** that represent various pieces, or views, of the systemclass diagram.
- **Multiple class diagrams,** that show the specific static relationshipsbetween various classes.

we will add relationships later (see Figure 3.26).

3) Defining Association Relationships

Identifying association begins by analyzing the interactions of each class. Remember that any dependency between two or more classes is an association.
The following are general guidelines for identifying the tentative associations:

- Association often corresponds to verb or prepositional phrases, such as part of, next to, works for, or contained in.
- A reference from one class to another is an association. Some associations are implicit or taken from general knowledge.

Some common patterns of associations are these:

- **Location association**. For example, next to, part of, contained in (noticethat apart- of relation is a special type of association).
- **Directed actions association**.
- □ **Communication association**. For example, talk to, order from.

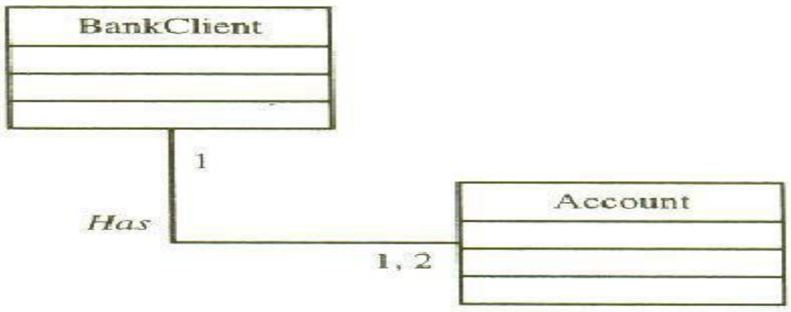


Fig. 3.27. Defining the BankClient-Account association multiplicity. One Client can have one or more Accounts (checking and savings accounts).

The first obvious relation is that each account belongs to a bank client since each BankClient has an account. Therefore, there is an association between the BankClient and Account classes. We need to establish cardinality among these classes.

By default, in most CASE tools such as SNObject Architect, all associations are considered one to one (one client can have only one account and vice versa). However, since each BankClient can have one or two accounts we need to change the cardinality of the association (see Figure 3.27). Other associations and their cardinalities are defined in Table 8-1 and demonstrated in Figure 3.28.

Table 8.1

SOME ASSOCIATIONS AND THEIR CARDINALITIES IN THE BANK SYSTEM

Class	Related class	Association name	Cardinality
Account	BankClient	Has	One
BankClient	Account		One or two
SavingsAccount	CheckingAccount	Savings-Checking	One
CheckingAccount	SavingsAccount		Zero or one
Account	Transaction	Account-Transaction	Zero or more
Transaction	Account		One

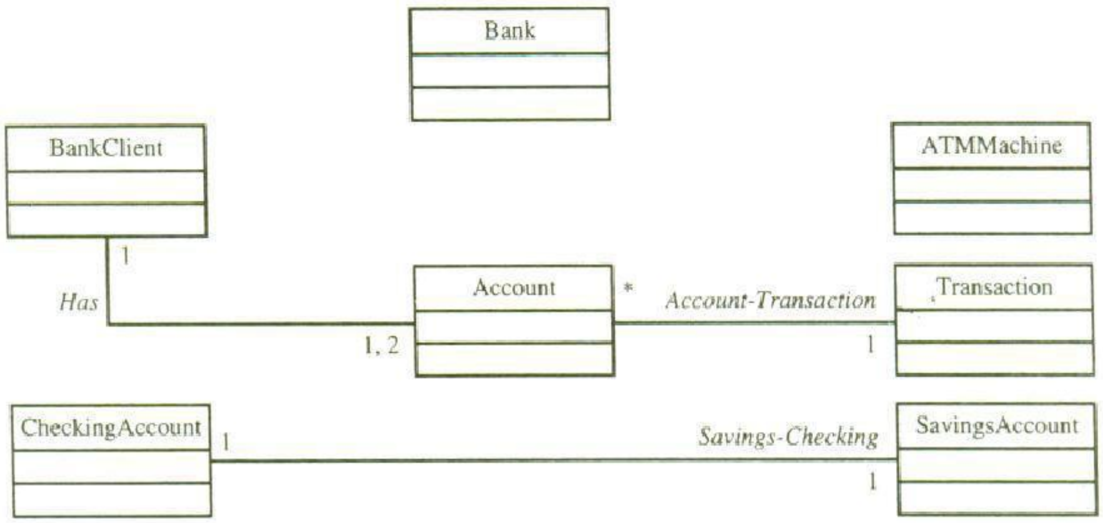


Fig. 3.28. Associations among the ViaNet bank ATM system classes.

4) Defining Super-Sub Relationships

Let us review the guidelines for identifying super-sub relationships:

*. **Top-down**. Look for noun phrases composed of various adjectives in the class name.
*. **Bottom-up**. Look for classes with similar attributes or methods. In most cases, you can group them by moving the common attributes and methods to an abstract class.
*. **Reusability.** Move attributes and behaviors (methods) as high as possible in the hierarchy.
***. Multiple inheritance.** Avoid excessive use of multiple inheritance.

CheckingAccount and SavingsAccount both are types of accounts. They can be defined as specializations of the Account class. When implemented, class will define attributes and services common to all kinds of accounts, with CheckingAccount and SavingsAccount each defining methods that make them more specialized. Fig 3.29.

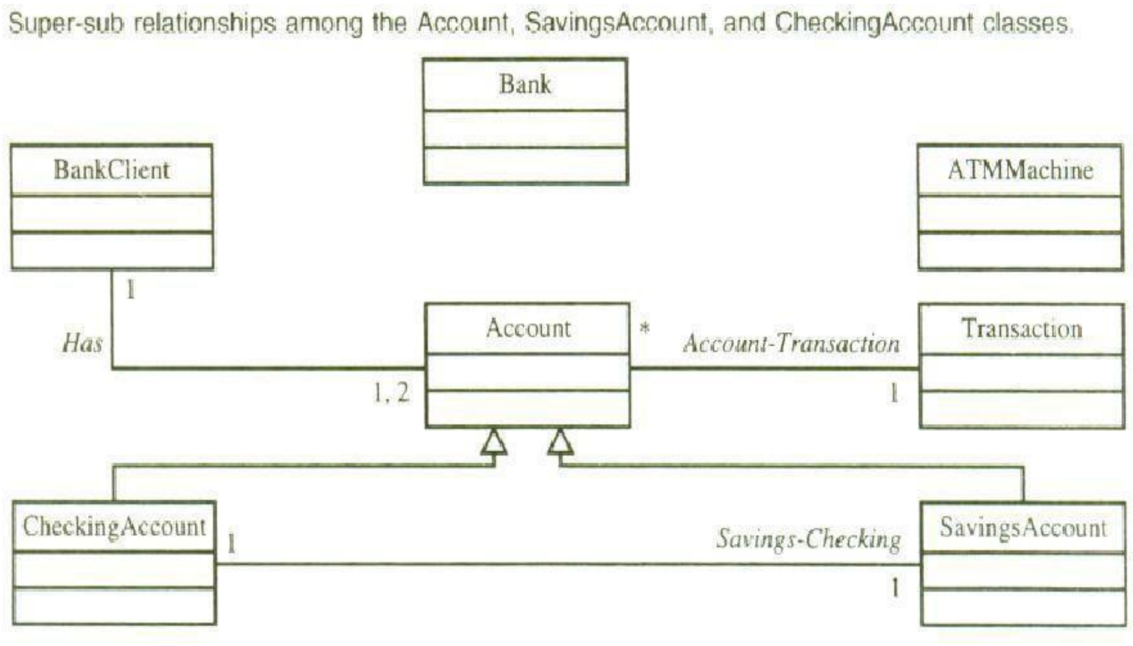


Fig 3.29- Super-sub relationships among the Account, SavingsAccount and CheckingAccount Classes

5) Identifying the Aggregation/a-Part-of Relationship

To identify a-part-of structures, we look for the following clues:

- Assembly. A physical whole is constructed from physical parts.
- Container. A physical whole encompasses but is not constructed from physical parts.
- Collection-Member. A conceptual whole encompasses parts that may be physical or conceptual.

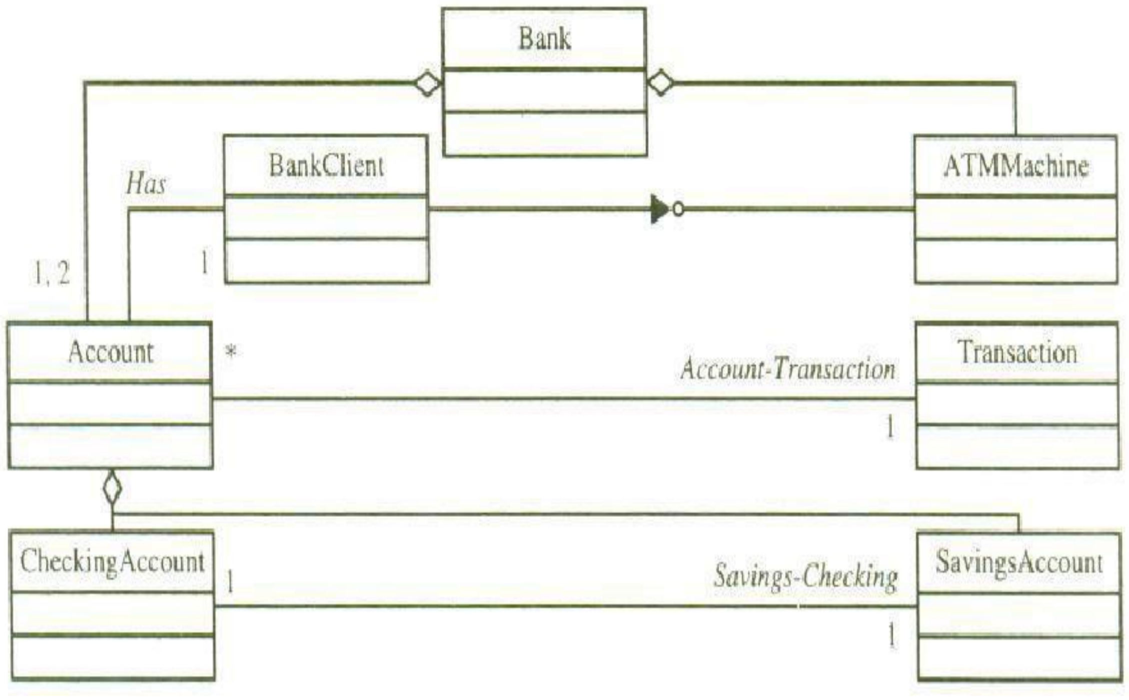


Fig 8.30. Association, generalization, and aggregation among the ViaNet bank classes. Notice that the super-sub arrows for CheckingAccount and SavingsAccount have merged. The relationship between BankClient and ATMMachine is an interface.

A bank consists of ATM machines, accounts, buildings, employees, and so forth. However, since buildings and employees are outside the domain of this application, we define the Bank class as an aggregation of ATM Machine and Account classes. Aggregation is a special type of association. Figure 3.30 depicts the association, generalization, and aggregation among the bank systems classes. If you are wondering what is the relationship between the BankClient and ATM Machine, it is an interface. Identifying a class interface is a design activity of object-oriented system development.

3.18 CLASS RESPONSIBILITY : IDENTIFYING ATTRIBUTES AND METHODS

Identifying attributes and methods is like finding Classes, still a difficult activity and an iterative process.
Responsibilities identify problems to be solved.

Attributes are things an object must remember such as color, cost and manufacturer. Identifying attributes of a system's classes starts with understanding the system' s responsibilities.

The following questions help in identifying the responsibilities of classes and deciding what data elements to keep track of:

# What information about an object should we keep track of?
# What services must a class provide?

3.19CLASS RESPONSIBILITY: DEFING ATTRIBUTES BY ANALYZING USE CASES AND OTHER UML DIAGRAMS

Attributes can be derived from scenario testing; therefore, by analyzing the use cases and sequence/collaboration, activity and state diagrams, you can begin to understand classes responsibilities and how they must interact to perform their tasks.

The main goal is to understand what the class is responsible for knowledge. Responsibility is the issue.

What kind of questions what kind of question would you like ask;

- How am I going to be used?
- How am I going to collaborate with other classes?
- How am I described in the context of this system's responsibility?

Guidelines for Defining Attributes.

Attributes usually correspond to nouns followed by prepositional phrases such as cost of the soup. Attributes also may correspond to adjectives or adverbs. Keep the class simple; state only enough attributes to define the object state. Attributes are less to be fully described in the problem statement. Omit derived attributes. Do not carry discovery of attributes to excess. You can add more attributes in subsequentiterations.

Important point to remember is that you may think of many attributes that can be associated with a class. You must careful to add only those attributes necessary to the design at hand.

3.20 OBJECT RESPONSIBILITY: METHODS AND MESSAGES.

Objects not only describe abstract data but also must provide some services.
Methods and messages are the workhorses of object-oriented systems.

In an object-oriented environment, every piece of data or object is surrounded by a rich set of routines called **methods.**

These methods do everything from printing the object to initializing its variables.
Every class is responsible for storing certain information from the domain knowledge. It also is logical to assign the responsibility for performing any operation necessary on that information.

Operations (methods or Behavior) in the o-o-system usually correspond to queries about attributes.

Methods are responsible for managing the value of attributes such as query, updating, reading and writing;

CASE STUDY : DEFINING ATTRIBUTES FOR VIANET BANK OBJETCTS.

1. **Defining Attributes for the BankClient Class**

   By analyzing the use cases, the sequence/collaboration diagrams and activity diagram of bank atm process, it is apparent that, for the BankClient Class, theproblem domain and system dictate certain attributes. In essence, what does thesystem need to know about the BankClient?

By looking at the activity diagram (See Fig 3.9) we notice that the BankClient must have a PIN number and CardNumber. Therefore, the PIN number and CardNumber are appropriate attributes for the BankClient.

The Attributes of the BankClient are
firstName

lastName

pinNumber

cardNumbe
account:Account

At this stage of the design, we are concerned with the functionality of the BankClient object and not with implementation attributes.

2. **Defining Attributes for the AccountClass**.

   Similarly, what information does the system need to know about an account? Based on the ATM Usecase diagram, Sequence/Collaboration diagram and activity diagram, BankClient can interact with its account by entering the account number and they could deposit money, get an account history, or get the balance. Therefore, we have defined the following attributes for the Account Class: number, balance.

CASE STUDY: DEFINING METHODS BY ANALYZING UML DIAGRAMS AND USE CASES.

We know that, in a sequence diagram, the objects involved are drawn on the diagram as vertical dashed lines. Furthermore, the events that occur between objects are drawn between the vertical object lines. An Event is considered tobe an action that transmits information.

For example, to define methods for the Account class, we look at sequencediagrams for the following use cases.

Deposit Checking

Deposit Savings
Withdraw Checking

Withdraw More from
Checking Withdraw Savings

Withdraw Savings Denied
Checking Transaction History
Savings Transaction History

CASE STUDY: DEFINING METHODS BY BANK OBJECTS

Operations (methods or behavior) in the object-oriented system usually correspond to events or actions that transmit information in the sequence diagram or queries about attributes of the objects. In other words, methods are responsible for managing the value for attributes such as query, updating, reading and writing.

1) Defining Account Class operations.

Deposit and withdrawal operations are available to the Client through the bank application, but they are provided as services by the Account Class, since the account objects must be able to manipulate their internal attributes. Account objects also must be able to create transaction records of any deposit or withdrawal they perform.

Here are the methods that we need to define for the Account Class:

deposit
Withdraw create
Transaction.

The services added to the Account class are those that apply to all subclasses of Account; namely, CheckingAccount and SavingsAccount. The subclass will either inherit these generic services without charge or enhance them to suit their own needs.

2) Defining BankClient Class Operation

Analyzing the sequence diagram (fig 3.13), it is apparent that the BankClient requires a method to validate client's passwords.

3) Defining CheckingAccount Class Operations

The requirement specification states that, when a checking account has insufficient funds to cover a withdrawal, it must try to withdraw the insufficient amount from its related saving account. To provide the service, the CheckingAccount class needs awithdrawal service that enables the transfer. Similarly, we must add the withdrawal service to the CheckingAccount class.

TEXT/ REFERENCE BOOKS:

1. Ali Bahrami, “Object oriented systems development using the unified modelling language”, 1st Edition, McGraw-Hill, 1998.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson, “The Unified Modelling Language User Guide”, 3rd Edition Addison Wesley,2007.
3. John Deacon, "Object Oriented Analysis and Design", 1st Edition, Addison Wesley, 2005.
4. Grady Booch, James Rumbaugh, and Ivar Jacobson, “The Unified Modelling Language User Guide”, 3rd Edition Addison Wesley.
5. John Deacon, “Object Oriented Analysis and Design”, 1st Edition, Addison Wesley.
6. Bernd Oestereich, “Developing Software with UML, Object - Oriented Analysis and Design in Practice”, Addison-Wesley

Questions

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Illustrate the tools available for extracting information about a system?	Analyze	BTL 4
2.	Describe the purpose of analysis? Why do we need analysis?	Knowledge	BTL 1
3.	Describe use-case model?	Knowledge	BTL 1
4.	Illustrate how would you identify actors?	Analyze	BTL 4
5.	Illustrate how would you name classes?	Analyze	BTL 4
6.	Explain why is identifying class hierarchy important In object-oriented analysis?	Knowledge	BTL 1
7.	Sketch generalization?	Apply	BTL 3
8.	Illustrate how would you identify a super-subclass structure?	Analyze	BTL 4
9.	Classify association different from an a-part-of relation?	Understand	BTL 2
10.	Sketch repeating attributes	Apply	BTL 3
	Part-B		
Q.No	**Questions**	**Competence**	**BT Level**
1.	Explain the guidelines for finding use cases and developing effective documentation.	Knowledge	BTL 1
2.	Illustrate the CRC approach and naming classes.	Analyze	BTL 4
3.	Classify detailed notes about the Noun phrase approach.	Understand	BTL 2

4.	Explain common class pattern approach with example.	Knowledge	BTL 1
5.	Explain use case driven approach with example.	Knowledge	BTL 1
6.	Conclude developing a sequence collaboration diagram a useful activity in identifying classes?	Analyze	BTL 4
7.	Explain how we can identify attributes and methods for classes?	Knowledge	BTL 1

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – IV - Object Oriented Analysis and Design– SBSA1403

OBJECT ORIENTED ANALYSIS AND DESIGN– SBSA1403

COURSE OBJECTIVES:

- To Understand the fundamentals of Object-Oriented System Development
- To understand the object-oriented methodologies.
- To use UML in requirements elicitation and designing.
- To understand concepts of relationships and aggregations.
- To test the software against its requirements specification

UNIT I - INTRODUCTION
Overview of object-oriented language systems development – Object basics hierarchy – Object and identity –Static and dynamic binding – Object oriented SDLC.

UNIT II - OBJECT ORIENTED METHODOLOGIES
Rumbaugh et al.'s technique – Booch, Jacobson Methodologies – Patterns – Framework – Unified approach –UML – UML diagrams – UML dynamic modelling – UML extensibility – UML meta-model.

UNIT III - OBJECT ORIENTED ANALYSIS
Use case model – Object analysis classification – Approaches for identifying classes – Classes responsibilities and collaborators – Identifying object relationships, attributes and methods.

UNIT IV - OBJECT ORIENTED DESIGN
Design process and design axioms – Designing classes – Access Layer – Object storage and object Interoperability – View layer – Designing interface objects.

UNIT V - SOFTWARE QUALITY
Software quality assurance – Testing strategies – Test cases – Test plan – Myers debugging principle – System usability and measuring user satisfaction.

COURSE OUTCOMES
CO1 - Understand the basics object model for System development.
CO2 - Understand the object-Oriented Methodologies
CO3 - Express software design with UML diagrams
CO4 - Understand the concept of Relationships
CO5 - Design software applications using OO concepts.
CO6 -Understand the various testing methodologies for OO software

UNIT IV
OBJECT ORIENTED DESIGN

4.1 **OO Design Process**

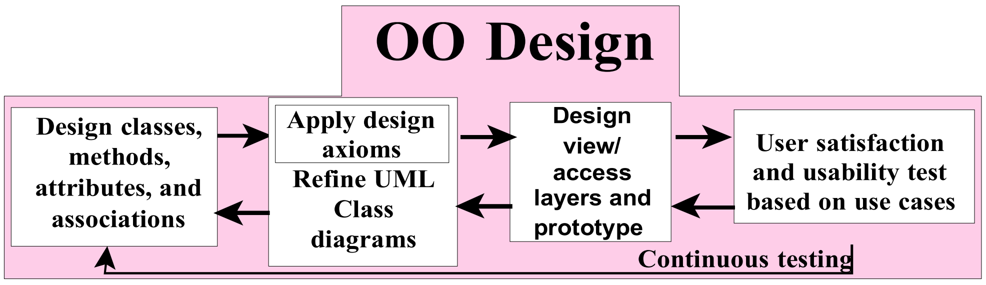

OO Design
Design classes, methods, attributes, and associations
Apply design axioms
Refine UML Class diagrams
Design view/ access layers and prototype
User satisfaction and usability test based on use cases
Continuous testing

Fig. 4.1. OO Design Process in the unified approach.

1. Apply design axioms to design classes, their attributes, methods, associations, structures, and protocols.
   1.1. Refine and complete the static UML class diagram (object model) by adding details to the UML class diagram. This step consists of the following activities:
   1.1.1. Refine attributes.
   1.1.2. Design methods and protocols by utilizing a UML activity diagram to represent the method's algorithm.
   1.1.3. Refine associations between classes (if required).
   1.1.4. Refine class hierarchy and design with inheritance (if required).
   1.2. Iterate and refine again.
2. Design the access layer
   2.1. Create mirror classes. For every business class identified and created, create one access class.
   2.2. define relationships among access layer classes.
   2.3. Simplify the class relationships. The main goal here is to eliminate redundant classes and structures.
   2.3.1. Redundant classes: Do not keep two classes that perform similar translate request and translate results activities. Simply select one and eliminate the other.
   2.3.2. Method classes: Revisit the classes that consist of only one or two methods to see if they can be eliminated or combined with existing classes.
   2.4. Iterate and refine again.
3. Design the view layer classes.
   3.1. Design the macro level user interface, identifying view layer objects.
   3.2. Design the micro level user interface, which includes these activities:
   3.2.1. Design the view layer objects by applying the design axioms and corollaries.
   3.2.2. Build a prototype of the view layer interface.
   3.3. Test usability and user satisfaction (Chapters 13 and 14).
   3.4. Iterate and refine.

4. Iterate and refine the preceding steps. Reapply the design axioms and, if needed, repeat the preceding steps.

4.1 Axioms, Theorems and Corollaries

An axiom is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception. A theorem is a proposition that may not be self-evident but can be proven from accepted axioms. A Corollary is a proposition that follows from an axiom or another proposition that has been proven.

Design Axioms

- Axiom 1 deals with relationships between system components (such as classes, requirements, software components).
- Axiom 2 deals with the complexity of design.

Axioms

Axiom 1. The independence axiom. Maintain the independence of components.

Axiom 2. The information axiom. Minimize the information content of the design. Occam's Razor

The best theory explains the known facts with a minimum amount of complexity and maximum simplicity and straightforwardness.

Corollaries

- **Corollary 1.** Uncoupled design with less information content.
- **Corollary 2.** Single purpose. Each class must have single, clearly defined purpose.
- **Corollary 3.** Large number of simple classes. Keeping the classes simple allows reusability.
- **Corollary 4.** Strong mapping. There must be a strong association between the analysis's object and design's object.
- **Corollary 5.** Standardization. Promote standardization by designing interchangeable components and reusing existing classes or components.
- **Corollary 6.** Design with inheritance. Common behavior (methods) must be moved to super classes.
- The superclass-subclass structure must make logical sense.

Coupling and Cohesion

- Coupling is a measure of the strength of association among objects.
- Cohesion is interactions within a single object or software component.
- Tightly Coupled Object

Corollary 1- Uncoupled Design with Less Information Content

The main goal here is to maximize objects (or software components) cohesiveness.

Corollary 2 - Single Purpose

Each class must have a purpose, as was explained!

When you document a class, you should be able to easily explain its purpose in a sentence or two.

Corollary 3- Large Number of Simpler Classes, Reusability

A great benefit results from having a large number of simpler classes.

The less specialized the classes are, the more likely they will be reused.

Corollary 4. Strong Mapping

As the model progresses from analysis to implementation, more detail is added, but it remains essentially the same. A strong mapping links classes identified during analysis and classes designed during the design phase.

Corollary 5. Standardization

The concept of design patterns might provide a way for standardization by capturing the design knowledge, documenting it, and storing it in a repository that can be shared and reused in different applications.

Corollary 6. Designing with Inheritance

Say we are developing an application for the government that manages the licensing procedure for a variety of regulated entities. Let us focus on just two types of entities: motor vehicles and restaurants.

Designing With Inheritance

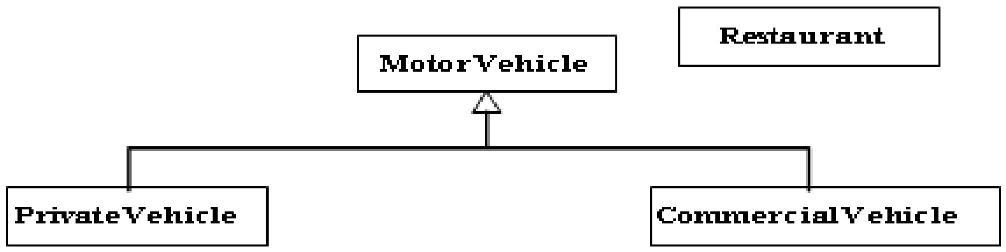


Fig 4.2 The initial single inheritance design.

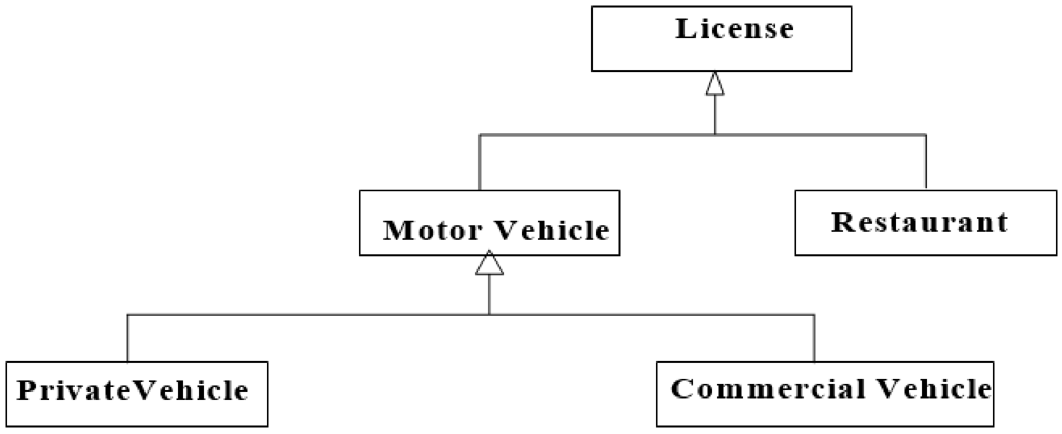


Fig 4.3. The single inheritance design modified to allow licensing food trucks.

Designing With Inheritance Weak Formal Class

- MotorVehicle and Restaurant classes do not have much in common.
- For example, of what use is the gross weight of a diner or the address of a truck?

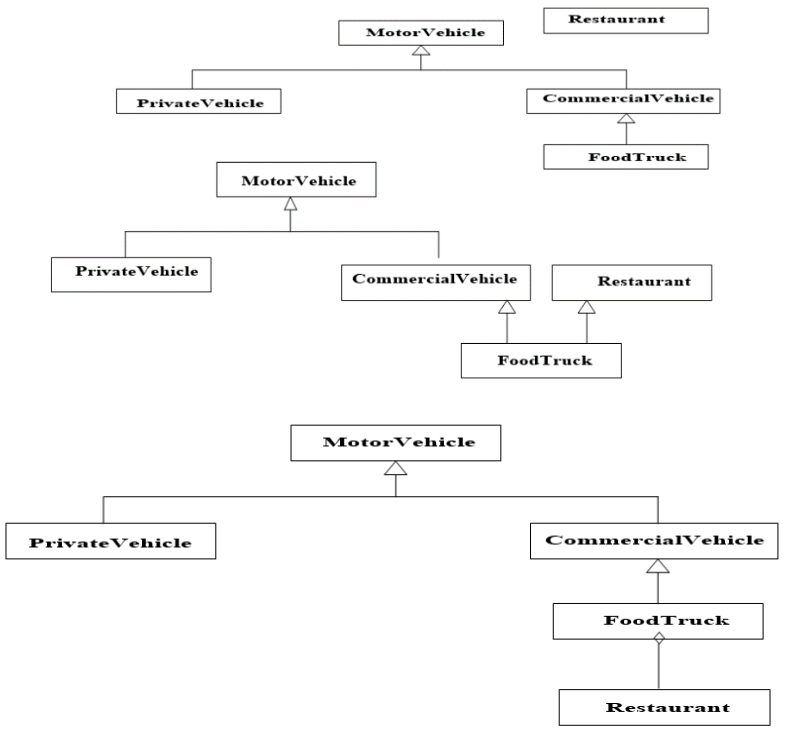


Fig 4.4. Achieving Multiple Inheritance using Single Inheritance Approach

Avoid Inheriting Inappropriate Behaviors
You should ask the following questions:

I. Is the subclass fundamentally similar to its superclass? or,
II. Is it entirely a new thing that simply wants to borrow some expertise from its superclass?

4.2 Designing Classes

OO Design Philosophy
The first step in building an application should be to design a set of classes, each of which has a specific expertise and all of which can work together in useful ways.

Designing Class: The Process

1. Apply design axioms to design classes, their attributes, methods, associations, structures, and protocols.

1.1. Refine and complete the static UML class diagram (object model) by adding details to that diagram.

   1.1.1. Refine attributes.
   1.1.2. Design methods and the protocols by utilizing a UML activity diagram to represent the method's algorithm.
   1.1.3. Refine the associations between classes (if required).
   1.1.4. Refine the class hierarchy and design with inheritance (if required).

1.2. Iterate and refine again.

Classes
In analysis modeling only the following needs to be shown:
Class name;
key attributes;
key operations
stereotypes (if they have any business significance)

Name compartment

- Class name is in UpperCamelcase
- Special symbols such as punctuation marks, dashes, underscores, ampersands, hashes, and slashes are always avoided and can lead to unexpected consequences when code or Html/Xml documentation is generated from the model.
- Avoid abbreviations at all costs
- Domain specific acronyms can be used (e.g. CRM-Customer Relationship Management)

Attribute Component
Attributes are named in lowerCamelCase-Starting with a lowercase letter and then mixed upper-and lowercase.
visibilityname:type[multiplicity]=initialValue

Visibility
Visibility controls access to the features of a class.
+ ->Public visibility
- ->Private visibility
# ->protected visibility
~ -> package visibility

Object Construction and destruction

- Constructors are a design consideration and are generally not shown on analysis models.

Destructors

- Destructors are special operations that “clean up” when objects are destroyed.
- Different languages follow different algorithms for clean-up.
- In java “Garbage Collection”

Class Visibility

- In designing methods or attributes for classes, you are confronted with two issues.
    - One is the protocol, or interface to the class operations and its visibility;
    - and how it should be implemented.
- Public protocols define the functionality and external messages of an object, while private protocols define the implementation of an object.

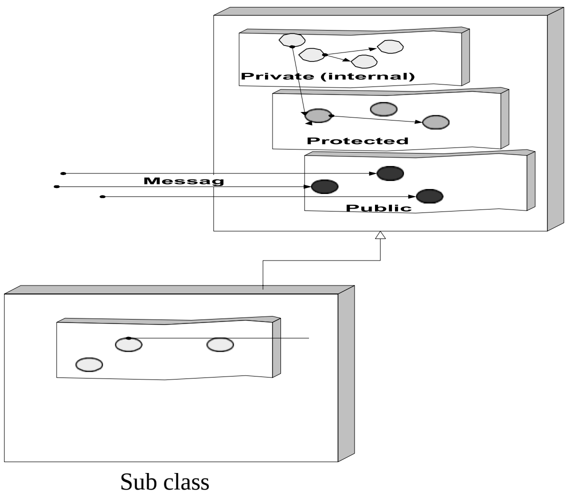


Fig 4.5. Class Visibility

Private Protocol (Visibility)

- A set of methods that are used only internally.
- Object messages to itself.
- Define the implementation of the object (Internal).
- Issues are: deciding what should be private.
    - What attributes
    - What methods
- In a protected protocol, subclasses can use the method in addition to the class itself.
- In private protocols, only the class itself can use the method.

Public Protocol (Visibility)

- Defines the functionality of the object
- Decide what should be public (External). Guidelines for Designing Protocols
- Good design allows for polymorphism.
- Not all protocols should be public, again apply design axioms and corollaries.
- The following key questions must be answered:
    - What are the class interfaces and protocols?
    - What public (external) protocol will be used or what external messages must the system understand?
    - What private or protected (internal) protocol will be used or what internal messages or messages from a subclass must the system understand?

Attribute Types
The three basic types of attributes are:

1. Single-value attributes.
2. Multiplicity or multivalued attributes.
3. Reference to another object, or instance connection.

Designing Methods and Protocols
A class can provide several types of methods:

- Constructor. Method that creates instances (objects) of the class.
- Destructor. The method that destroys instances.
- Conversion method. The method that converts a value from one unit of measure to another.
- Copy method. The method that copies the contents of one instance to another instance.
- Attribute set. The method that sets the values of one or more attributes.
- Attribute get. The method that returns the values of one or more attributes
- I/O methods. The methods that provide or receive data to or from a device.
- Domain specific. The method specific to the application.

Five Rules for Identifying Bad Design

- If it looks messy then it's probably a bad design.
- If it is too complex then it's probably a bad design.
- If it is too big then it's probably a bad design.
- If people don't like it then it's probably a bad design.
- If it doesn't work then it's probably a bad design.

Avoiding Design Pitfalls

- Keep a careful eye on the class design and make sure that an object's role remains well defined.
- If an object loses focus, you need to modify the design.
- Apply Corollary 2 (single purpose).
- Move some functions into new classes that the object would use.
- Apply Corollary 1 (uncoupled design with less information content).
- Break up the class into two or more classes.
- Apply Corollary 3 (large number of simple classes).

Access Layer
Introduction
A database management system (DBMS) is a collection of related data and associated programs that access, manipulate, protect and manage data.

What’s the purpose of DBMS?
The main purpose of a DBMS is to provide a reliable, persistent data storage facility, and mechanism for efficient and convenient data access and retrieval.

Persistence (review)
Persistence is defined as objects that outlive the programs which created them.

• Persistent object stores do not support query or interactive user interface facilities, as found in a fully supported DBMS or OODBMS.

Object Storage and Persistence

Atkinson et al. describe six broad categories for the lifetime of data:

1. Transient results to the evaluation of expressions.
2. Variables involved in procedure activation (parameters and variables with a localized scope).
3. Global variables and variables that are dynamically allocated.
4. Data that exist between the executions of a program.
5. Data that exist between the versions of a program.
6. Data that outlive a program.

Database Management Systems

- A DBMS is a set of programs that enable the creation and maintenance of a collection of related data.
- DBMS have a number of properties that distinguish them from the file-based data management approach.

Database system Vs. File System

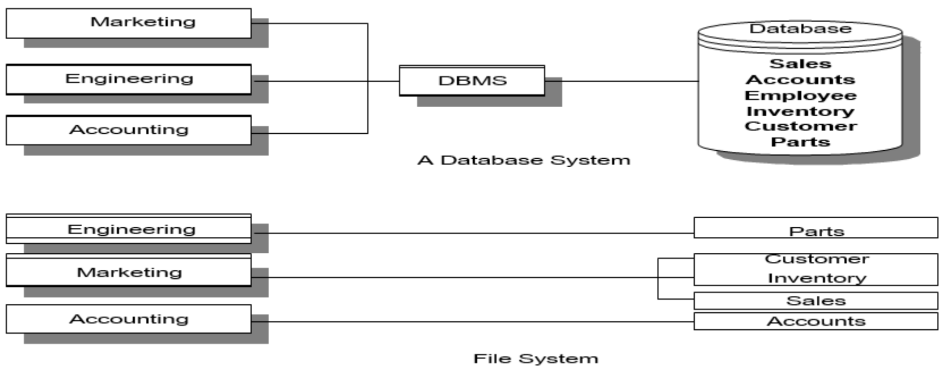


Fig 4.6. Database system Vs. File System

Database Models

A database model is a collection of logical constructs used to represent the data structure and data relationships within the database.

- Hierarchical Model
- Network Model
- Relational Model

Hierarchical model

The hierarchical model represents data as a single-rooted tree

Network model

A network database model is similar to a hierarchical database, with one distinction.
Unlike the hierarchical model, a network model's record can have more than one parent.

Relational Model
Of all the database models, the relational model has the simplest, most uniform structure and is the most commercially widespread.

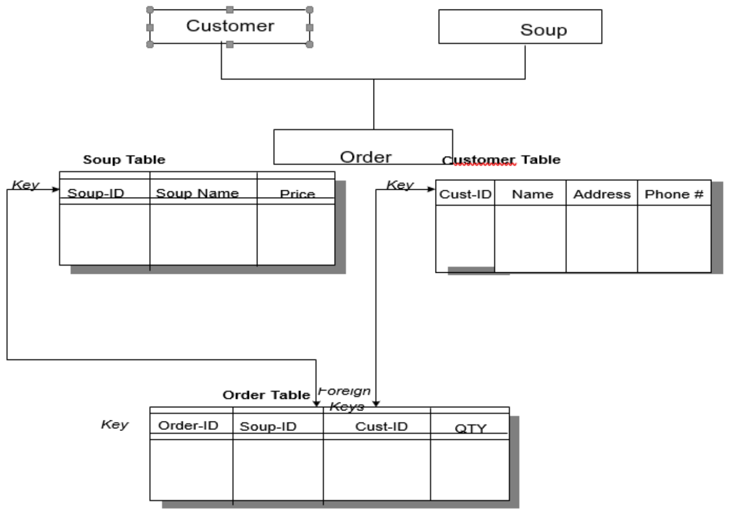


Fig 4.6. Relational Model

What is a schema and metadata?
- The schema, or metadata, contains a complete definition of the data formats, such as the data structures, types, and constraints.
- In an object-oriented DBMS, the schema is the collection of class definitions.
- The relationships among classes (such as super/sub) are maintained as part of the schema.

Database Definition Language (DDL)
- A database definition language (DDL) is used to describe the structure of and relationships between objects stored in a database.

Data Manipulation Language (DML)
- Once data is stored in a database, there must be a way to get it, use it, and manipulate it.
- DML is a language that allows users to access and manipulate (such as: creation, saving and destruction of) data organization.
- The Structured Query Language (SQL) is the standard DML for relational DBMS.
- In a relational DBMS, the DML is independent from a host programming language.
- For example, a host language such as C or COBOL would be used to write the body of the application.
- SQL statements are then typically embedded in C or COBOL applications to manipulate data.

Shareability
- Data in the database often needs to be accessed and shared by different applications.
- The database then must detect and mediate the conflicts and promote the greatest amount of sharing possible without sacrificing the integrity of data.

Transaction
- A transaction is a unit of change, in which either all changes to objects within a transaction will be applied or not at all.
- A transaction is said to commit if all changes can be successfully made to the database and to abort if all changes cannot be successfully made to the database.

Concurrency Control

- Programs will attempt to read and write the same pieces of information simultaneously and, in doing so, create a contention for data.
- The concurrency control mechanism is thus established to mediate these conflicts.
- It does so by making policies that dictate how read and write conflicts will be handled.
- The most conservative way is to allow a user to lock all records or objects when they are accessed and to release the locks only after a transaction commits.
- By distinguishing between reading the data, and writing it (which is achieved by qualifying the type of lock placed in the data-read lock or write lock) somewhat greater concurrency can be achieved.
- This policy allows many readers of a record or an objective, but only one writer.

Distributed Databases

In distributed databases portions of the database reside on different nodes (computers) in the network.

Client/Server Computing

Client/server computing allows objects to be executed in different memory spaces or even different machines. The calling module becomes the "client" (which requests a service), and the called module becomes the "server" (which provides the service).

Client programs usually manage:

- The user-interface
- Validate data entered by the user
- Dispatch requests to server programs, and sometimes
- Execute business logic.
- The Business layer contains all of the objects that represent the business such as:
  - Order
  - Customer
  - Line Item
  - Inventory, etc.
- A server process (program) fulfills the client request by performing the task requested.
- Server programs generally receive requests from client programs, execute database retrieval and updates, manage data integrity, and dispatch responses to client requests.

A Two-Tier Architecture

- A two-tier architecture is one where a client talks directly to a server, with no intervening server.
- This type of architecture is typically used in small environments with less than 50 users.

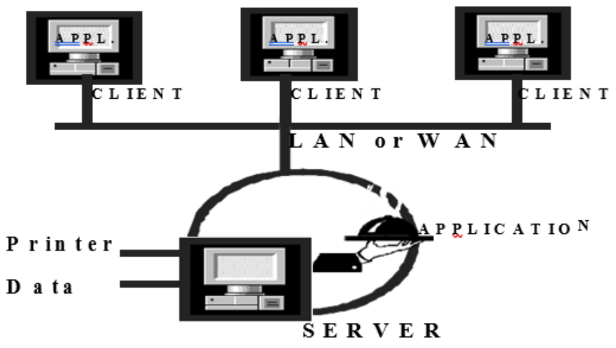


Fig 4.7. A Two-Tier Architecture

A Three-Tier Architecture

- A three-tier architecture introduces another server (or an "agent") between the client and the server.
- The role of the agent is many fold.
- It can provide translation services as in adapting a legacy application on a mainframe to a client/server environment.

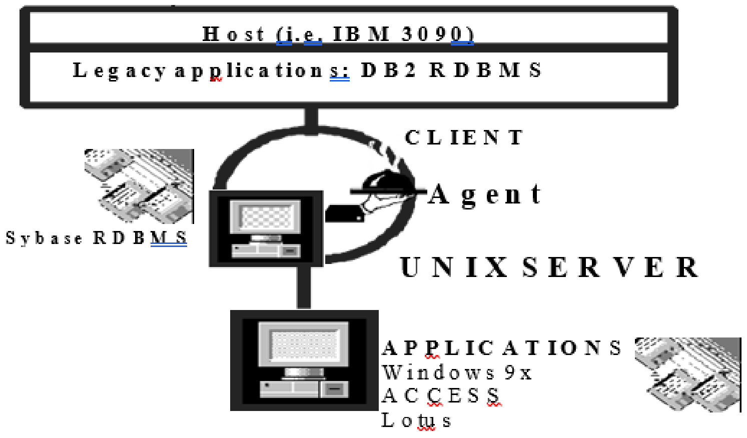


Fig 4.8. A Three-Tier Architecture

Basic Characteristics of Client/Server Architectures

1. The client or front-end interacts with the user, and a server or back-end interacts with the shared resource.
2. The front-end task and back-end task have fundamentally different requirements for computing resources.
3. Resources such as processor speeds, memory, disk speeds and capacities, and input/output devices.
4. The environment is typically heterogeneous and multi-vendor.
5. Client-server systems can be scaled horizontally or vertically.
6. Horizontal scaling means adding or removing client workstations with only a slight performance impact.
7. Vertical scaling means migrating to a larger and faster server machine or multi servers.

Distributed Processing

- Distributed processing implies that processing will occur on more than one processor in order for a transaction to be completed.

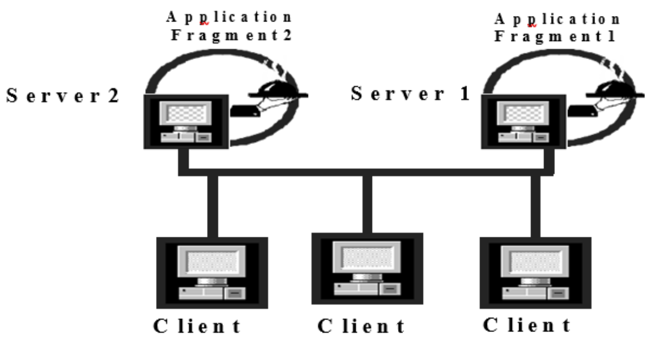


Fig 4.9. Distributed Processing

For example, in processing an order from our client, the client information may process at one machine and the account information will then be processed next on a different machine.

Cooperative processing

Cooperative processing is a form of distributed computing where two or more distinct processes are required to complete a single business transaction.

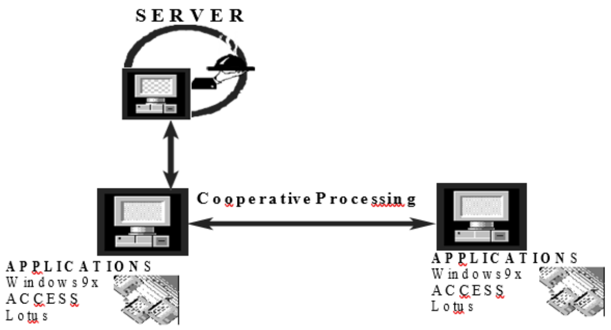


Fig 4.10. Cooperative processing

Client/Server Components

1. **User Interface Layer:** This is one of the major components of the client/server application. It interacts with users, screens, Windows, Window management, keyboard, and mouse handling.
2. **Business Processing Layer:** This is a part of the application that uses the user interface data to perform business tasks.
3. **Database Processing Layer:** This is a part of the application code that manipulates data within the application.

Object-Oriented Database Systems

The object-oriented database management system is a marriage of object-oriented programming and database technology to provide what we now call object-oriented databases.

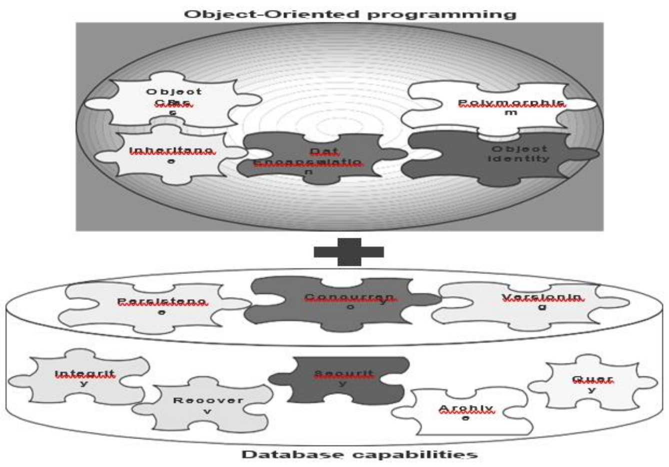


Fig 4.11. Object-Oriented Database Systems

Object-Oriented Database System Manifesto

Malcolm Atkinson et al. described the necessary characteristics that a system must satisfy to be considered an object-oriented database.

These categories can be broadly divided into object-oriented language properties and database requirements.

First, the rules that make it an object-oriented system are as follows:

1. The system must support complex objects.
2. Object identity must be supported.
3. Objects must be encapsulated.
4. The system must support types or classes.
5. The system must support inheritance.
6. The system must avoid premature binding.
7. The system must be computationally complete.
8. The system must be extensible.

Second, these rules make it a DBMS:

9. It must be persistent, able to remember an object state.
10. It must be able to manage very large databases.
11. It must accept concurrent users.
12. It must be able to recover from hardware and software failures.
13. Data query must be simple.

Object-Oriented Databases versus Traditional Databases

- The objects are an "active" component in an object-oriented database.
- Relational database systems do not explicitly provide inheritance of attributes and methods.
- Each object has its own identity, or object-ID (as opposed to the purely value-oriented approach of traditional databases).
- Object identity allows objects to be related as well as shared within a distributed computing network.
- Object-Relational Systems: The Practical World
- In practice, even though many applications increasingly are developed in an object- oriented environment, chances are good that the data those applications need to access live in a very different universe—a relational database.

Object-Relation Mapping

- For a tool to be able to define how relational data maps to and from application objects, it must have at least the following mapping capabilities:
- Table-class mapping.
- Table-multiple classes mapping.
- Table-inherited classes mapping.
- Tables-inherited classes mapping.

Table-Class Mapping

- Each row in the table represents an object instance and each column in the table corresponds to an object attribute.

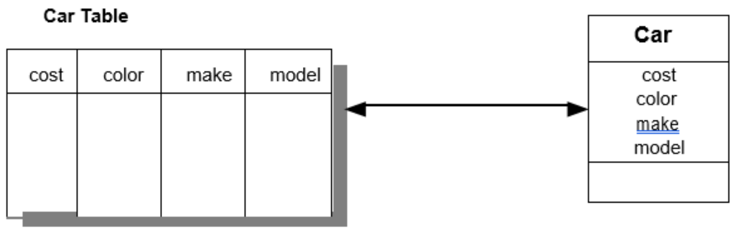


Fig 4.12. Table-Class Mapping

Table-Multiple Classes Mapping

The custID column provides the discriminant. If the value for custID is null, an employee instance is created at run time; otherwise, a Customer instance is created.

Table-Inherited Classes Mapping
Instances of SalariedEmployee can be created for any row in the Person table that has a non null value for the salary column. If salary is null, the row is represented by an HourlyEmployee instance.

Tables-Inherited Classes Mapping
Instances of Person are mapped directly from the Person table. However, instances of Employee can be created only for the rows in the Employee table (the joins of the Employee and Person tables on the ssn key). The ssn is used both as a primary key on the Person table and as a foreign key on the Person table and a primary key on the Employee table for activating instances of type Employee.

Keys for Instance Navigation
The departmentID property of Employee uses the foreign key in column Employee.departmentID. Each Employee instance has a direct reference of class Department (association) to the department object to which it belongs.

Multidatabase System
A different approach for integrating object-oriented applications with relational data environments is multidatabase systems, or heterogeneous database systems, which facilitate the integration of heterogeneous databases and other information sources.

Federated Multidatabase Systems
Federated multidatabase systems provide a solution to the problem of interoperating heterogeneous data systems, provide uniform access to data stored in multiple databases that involve several different data models.

MDBS
A multidatabase system (MDBS) is a database system that resides on top of, say existing relational and object databases and file systems (called local database systems) and presents a single database illusion to its users.

Characteristics of MDBS
- Automatic generation of a unified global database schema from local databases.
- Provision of cross-database functionality (global queries, updates, and transactions) by using unified schemata.
- Integration of a heterogeneous database system with multiple databases.
- Integration of data types other than relational data through the use of such tools as driver generators.
- Provision of a uniform but diverse set of interfaces (e.g., a SQL-style interface, browsing tools, and C++) to access and manipulate data stored in local databases.

Open Database Connectivity
Open database connectivity (ODBC), provides a mechanism for creating a virtual DBMS.

Designing Access Layer Classes
- The main idea behind creating an access layer is to create a set of classes that know how to communicate with data source, whether it be a file, relational database, mainframe, Internet, DCOM, or via ORB.
- The access classes must be able to translate any data-related requests from the business layer into the appropriate protocol for data access.
- The business layer objects and view layer objects should not directly access the database. Instead, they should consult with the access layer for all external system connectivity.

4.3. Designing Access Layer Classes

- The main idea behind creating an access layer is to create a set of classes that know how to communicate with data source, whether it be a file, relational database, mainframe, Internet, DCOM, or via ORB.
- The access classes must be able to translate any data-related requests from the business layer into the appropriate protocol for data access.
- The business layer objects and view layer objects should not directly access the database. Instead, they should consult with the access layer for all external system connectivity.

The access layer performs two major tasks:

1. Translate the request.
2. Translate the results.

Benefits of Access Layer Classes

- Access layer classes provide easy migration to emerging distributed object technology, such as CORBA and DCOM.
- These classes should be able to address the (relatively) modest needs of two-tier client/server architectures as well as the difficult demands of fine-grained, peer-to- peer distributed object architectures.

Process

The access layer design process consists of these following activities:
If a class interacts with a nonhuman actor, such as another system, database or the web

1. For every business class identified, mirror the business class package.
2. Define relationships. The same rule as applies among business class objects also applies among access classes.
3. Simplify classes and relationships. The main goal here is to eliminate redundant or unnecessary classes or structures.
   3.1. Redundant classes. If you have more than one class that provides similar services, simply select one and eliminate the other(s).
   3.2. Method classes. Revisit the classes that consist of only one or two methods to see if they can be eliminated or combined with existing classes.
4. Iterate and refine.
   Another approach is to let the methods be stored in a program (ex: A compiled c++ program stored on a file)
   1. For every business class identified, determine if the class has persistent data.
   2. Mirror the business class package.
   3. Define relationships. The same rule as applies among business class objects also applies among access classes.
   4. Simplify classes and relationships. The main goal here is to eliminate redundant or unnecessary classes or structures.
      4.1. Redundant classes. If you have more than one class that provides similar services, simply select one and eliminate the other(s).
      4.2. Method classes. Revisit the classes that consist of only one or two methods to see if they can be eliminated or combined with existing classes.
5. Iterate and refine

4.5. Designing View Layer Classes

- Interface objects
  - The only exposed objects of an application with which users can interact
  - Represent the set of operations in the business that users must perform to complete their tasks

The view layer classes are responsible for two major aspects of the applications:

- Input-Responding to user interaction
- User interface must be designed to translate an action by the user
- Output-Displaying business objects

Design of the view layer classes are divided into the following activities:

I. Macro Level UI Design Process- Identifying View Layer Objects.

II. Micro Level UI Design Activities.
   - Designing the view layer objects by applying design axioms
   - Prototyping the view layer interface

III. Usability and User Satisfaction Testing.

IV. Refine and iterate.

Macro-Level Process – By analyzing usecases

1. For Every Class Identified
   - 1.1 Determine If the Class Interacts With Human Actor: If yes, do next step otherwise move to next class.
     - 1.1.1 Identify the View (Interface) Objects for The Class.
     - 1.1.2 Define Relationships Among the View (Interface) Objects.
2. Iterate and refine.

Relationships Among Business, Access and View Classes

In some situations, the view class can become a direct aggregate of the access object, as when designing a web interface that must communicate with application/Web server through access objects.

View Layer Micro Level

- Better to design the view layer objects user driven or user centered
- Process of designing view objects

1. For Every Interface Object Identified in the Macro UI Design Process.
   1.1 Apply Micro Level UI Design Rules and Corollaries to Develop the UI.
2. Iterate and refine.

UI Design Rules

- Rule 1- Making the Interface Simple
- Rule 2- Making the Interface Transparent and Natural
- Rule 3- Allowing Users to Be in Control of the Software

UI Design Rule 1

- Making the interface simple: application of corollary 2.
- Keep It Simple.
- Simplicity is different from being simplistic.
- Making something simple requires a good deal of work and code.
- Every additional feature potentially affects performance, complexity, stability, maintenance, and support costs of an application.
- A design problem is harder to fix after the release of a product because users may adapt, or even become dependent on, a peculiarity in the design.

UI Design Rule 2

- Making the interface transparent and Natural: application of corollary 4.
- Corollary 4 implies that there should be strong mapping between the user's view of doing things and UI classes.

Making The Interface Natural

- The user interface should be intuitive so users can anticipate what to do next by applying their previous knowledge of doing tasks without a computer.

Using Metaphors

- Metaphore, relates two unrelated things by using one to denote the other
- Metaphors can assist the users to transfer their previous knowledge from their work environment to your application interface.
- For example, question mark to label a Help button.

UI Design Rule 3

- Allowing users to be in control of the software: application of corollary 1.
- Users should always feel in control of the software, rather than feeling controlled by the software.

Allowing Users Control of the Software

- Some of the ways to put users in control are:
  - Making the interface forgiving.
  - Making the interface visual.
  - Providing immediate feedback.
  - Making the interface consistent.

Making the Interface Forgiving

- Users should be able to back up or undo their previous action.
- They should be able to explore without fear of causing an irreversible mistake.

Making the Interface Visual

- You should make your interface highly visual so users can see, rather than recall, how to proceed.
- Whenever possible, provide users with a list of items from which they can choose.

Providing Immediate Feedback

- Users should never press a key or select an action without receiving immediate visual feedback, audible feedback, or both.

Making the Interface Consistent

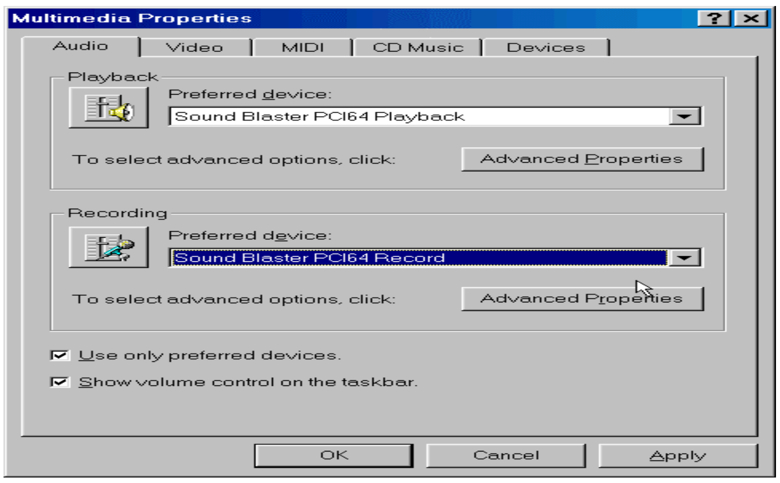
- User Interfaces should be consistent throughout the applications.
- For example, keeping button locations consistent make users feel in control.

Purpose of a View Layer Interface

- Data Entry Windows: Provide access to data that users can retrieve, display, and change in the application.
- Dialog Boxes: Display status information or ask users to supply information.
- Application Windows (Main Windows): Contain an entire application that users can launch.

Guidelines For Designing Data Entry Windows

- You can use an existing paper form such as a printed invoice form as the starting point for your design.
- If the printed form contains too much information to fit on a screen:
- Use main window with optional smaller Windows that users can display on demand, or
- Use a window with multiple pages.
- Users scan a screen in the same way they read a page of a book, from left to right, and top to bottom.
- An example of a dialog box with multiple pages in the Microsoft multimedia setup.



Multimedia Properties

Audio | Video | MIDI | CD Music | Devices

Playback

Preferred device: Sound Blaster PCI64 Playback

To select advanced options, click: Advanced Properties

Recording

Preferred device: Sound Blaster PCI64 Record

To select advanced options, click: Advanced Properties

☑ Use only preferred devices.

☑ Show volume control on the taskbar.

OK | Cancel | Apply

Fig 4.13. Designing Data Entry Windows

- Orient the controls in the dialog box in the direction people read.
- Usually left to right, top to bottom.
- Required information should be put toward the top and left side of the form, entering optional or seldom entered information toward the bottom.

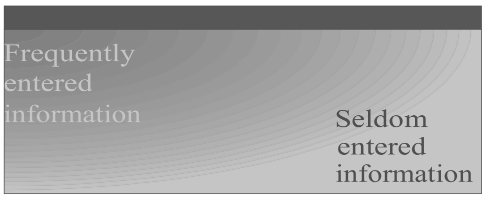


Fig 4.14. Dialog box

Place text labels to the left of text box controls, align the height of the text with text displayed in the text box.

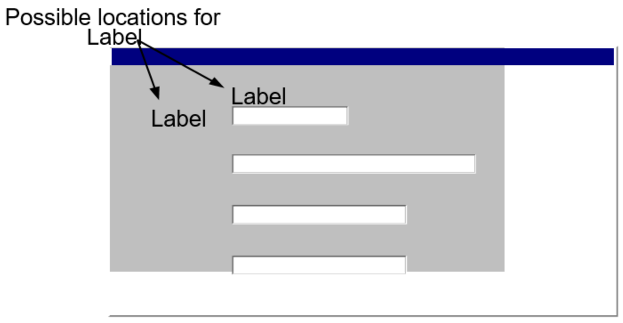


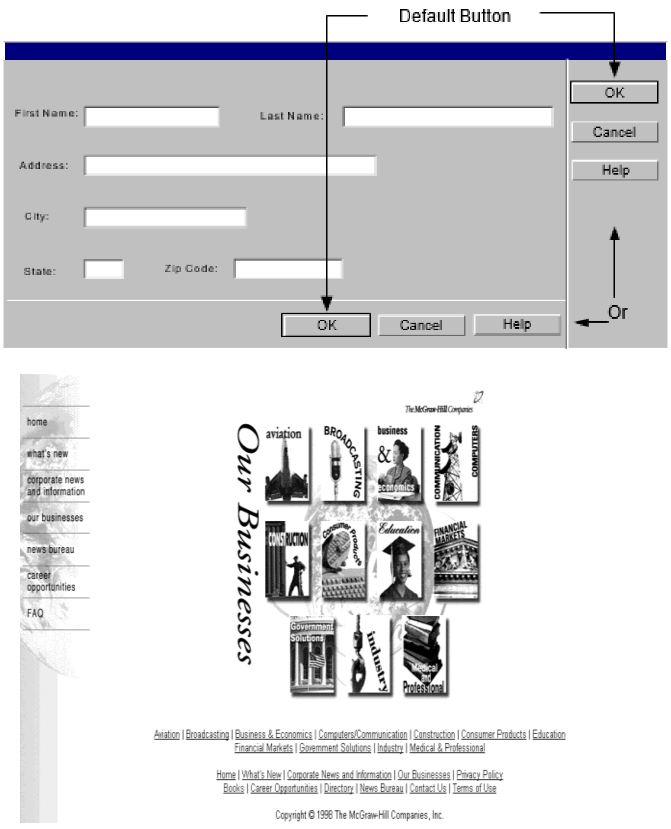
Fig 4.15. Text box controls

Guidelines For Designing Dialog Boxes

- If the dialog box is for an error message, use the following guidelines:
- Your error message should be positive.
- For example, instead of displaying "You have typed an illegal date format," display this message "Enter date format mm/dd/yyyy."
- Your error message should be constructive, brief and meaningful.
- For example, avoid messages such as
  "You should know better! Use the OK button"
- instead display
  "Press the Undo button and try again."

Guidelines for the Command Buttons Layout

- Arrange the command buttons either along the upper-right border of the form or dialog box or lined up across the bottom.
- Positioning buttons on the left or centre is popular in Web interfaces.



Aviation | Broadcasting | Business & Economics | Computers/Communication | Construction | Consumer Products | Education
Financial Markets | Government Solutions | Industry | Medical & Professional

Home | What's New | Corporate News and Information | Our Businesses | Privacy Policy
Books | Career Opportunities | Directory | News Bureau | Contact Us | Terms of Use

Copyright © 1998 The McGraw-Hill Companies, Inc.

Fig 4.16. Command Buttons Layout

Guidelines For Designing Application Windows

A typical application window consists of a frame (or border) which defines its extent: title bar, scroll bars, menu bars, toolbars, and status bars.

Guidelines For Using Colors

- Use identical or similar colors to indicate related information.
- Use different colors to distinguish groups of information from each other.
- For example, checkout and in-stock tapes could appear in different colors.
- For an object background, use a contrasting but complementary color.
- For example, in an entry field, make sure that the background color contrasts with the data color
- Use bright colors to call attention to certain elements on the screen.
- Use dim colors to make other elements less noticeable.
- For example, you might want to display the required field in a brighter color than optional fields.
- Use colors consistently within each window and among all Windows in your application.
- For example the colors for Pushbuttons should be the same throughout.
- Using too many colors can be visually distracting, and will make your application less interesting.
- Allow the user to modify the color configuration of your application.

Guidelines For Using Fonts

- Use commonly installed fonts, not specialized fonts that users might not have on their machines.
- Use bold for control labels so they will remain legible when the object is dimmed.
- Use fonts consistently within each form and among all forms in your application.
- For example, the fonts for check box controls should be the same throughout.
- Consistency is reassuring to users, and psychologically makes users feel in control.
- Using too many font styles, sizes and colors can be visually distracting and should be avoided.

Prototyping the User Interface

- Rapid prototyping encourages the incremental development approach, “grow, don’t build.”

Three General Steps

- Create the user interface objects visually.
- Link or assign the appropriate behaviors or actions to these user interface objects and their events.
- Test, debug, then add more by going back to step 1.

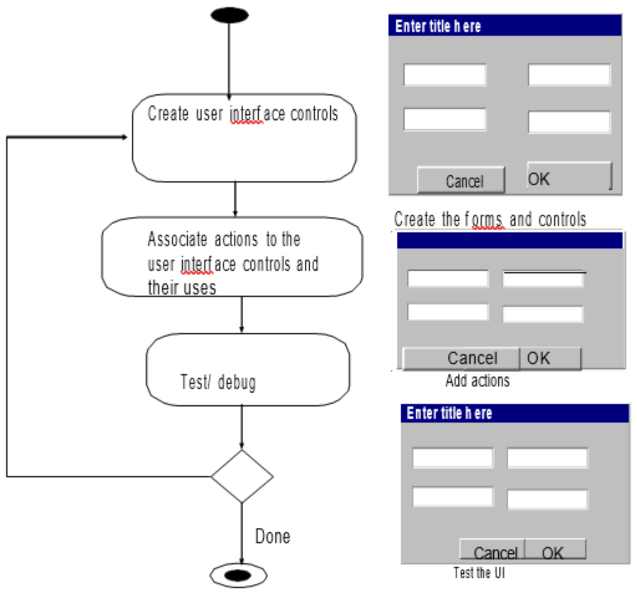

Create user interface controls
Associate actions to the user interface controls and their uses
Test/ debug
Done
Enter title here
Cancel
OK
Create the forms and controls
Cancel
OK
Add actions
Enter title here
Cancel
OK
Test the UI

Fig 4.17. Prototyping the User Interface

TEXT/ REFERENCE BOOKS:

1. Ali Bahrami, "Object oriented systems development using the unified modelling language", 1st Edition, McGraw-Hill, 1998.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson, "The Unified Modelling Language User Guide", 3rd Edition Addison Wesley,2007.
3. John Deacon, "Object Oriented Analysis and Design", 1st Edition, Addison Wesley, 2005.
4. Grady Booch, James Rumbaugh, and Ivar Jacobson, "The Unified Modelling Language User Guide", 3rd Edition Addison Wesley.
5. John Deacon, "Object Oriented Analysis and Design", 1st Edition, Addison Wesley.
6. Bernd Oestereich, "Developing Software with UML, Object - Oriented Analysis and Design in Practice", Addison-Wesley

Question

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Compare Coupling and Cohesion.	Analysis	BTL 4
2.	List out the types of Database models.	Remember	BTL 1
3.	Define Corollaries.	Remember	BTL 1
4.	List out the types of attributes.	Remember	BTL 1
5.	Define DDL and DML.	Remember	BTL 1
6.	Define Axiom.	Remember	BTL 1
7.	What is meant by Coupling?	Remember	BTL 1
8.	What is OCL?	Remember	BTL 1
9.	Analyze the purpose of DBMS.	Analysis	BTL 4
10.	List out the various Visibility modes.	Remember	BTL 1
Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Elaborate Object-Oriented Design process in detail.	Remember	BTL 1
2.	Explain the steps involved in designing the access layer classes.	Remember	BTL 1
3.	Explain the activities involved in the macro and micro level processes while designing the view layer classes.	Remember	BTL 1

4.	Explain about Corollaries in detail.	Remember	BTL 1
5.	Discuss in detail about (i)Client Server computing (ii)Distributed Database	Understand	BTL 2
6.	(i)Compare Traditional Database and Object-Oriented Database. (ii)Analyze the Characteristics of OOD	Analysis	BTL 4
7.	Explain OODBMS in detail.	Remember	BTL 1

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – V - Object Oriented Analysis and Design– SBSA1403

OBJECT ORIENTED ANALYSIS AND DESIGN– SBSA1403

COURSE OBJECTIVES:

- To Understand the fundamentals of Object-Oriented System Development
- To understand the object-oriented methodologies.
- To use UML in requirements elicitation and designing.
- To understand concepts of relationships and aggregations.
- To test the software against its requirements specification

UNIT I - INTRODUCTION

Overview of object-oriented language systems development – Object basics hierarchy – Object and identity –Static and dynamic binding – Object oriented SDLC.

UNIT II - OBJECT ORIENTED METHODOLOGIES

Rumbaugh et al.'s technique – Booch, Jacobson Methodologies – Patterns – Framework – Unified approach –UML – UML diagrams – UML dynamic modelling – UML extensibility – UML meta-model.

UNIT III - OBJECT ORIENTED ANALYSIS

Use case model – Object analysis classification – Approaches for identifying classes – Classes responsibilities and collaborators – Identifying object relationships, attributes and methods.

UNIT IV - OBJECT ORIENTED DESIGN

Design process and design axioms – Designing classes – Access Layer – Object storage and object Interoperability – View layer – Designing interface objects.

UNIT V - SOFTWARE QUALITY

Software quality assurance – Testing strategies – Test cases – Test plan – Myers debugging principle – System usability and measuring user satisfaction.

COURSE OUTCOMES

CO1 - Understand the basics object model for System development.
CO2 - Understand the object-Oriented Methodologies
CO3 - Express software design with UML diagrams
CO4 - Understand the concept of Relationships
CO5 - Design software applications using OO concepts.
CO6 -Understand the various testing methodologies for OO software

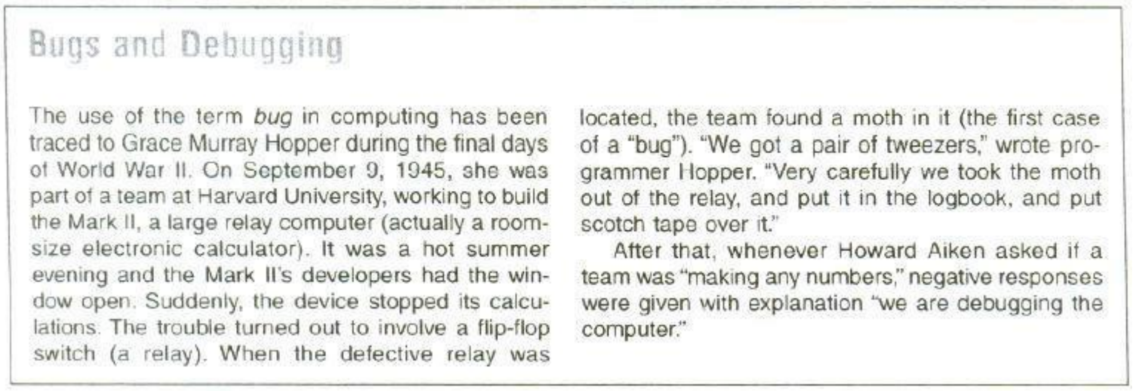
UNIT V
SOFTWARE QUALITY

Software quality assurance – Testing strategies – Test cases – Test plan – Myers debugging principle – System usability and measuring user satisfaction.

5.1. SOFTWARE QUALITY ASSURANCE

In the early history of computers, live bugs could be a problem (see Bugs and Debugging). Moths and other forms of natural insect life no longer trouble digital computers. However, bugs and the need to debug programs remain. In a 1966 article in Scientific American, computer scientist Christopher Strachey wrote,

Although programming techniques have improved immensely since the early years, the process of finding and correcting errors in programming-"debugging" still remains a most difficult, confused and unsatisfactory operation. The chief impact of this state of affairs is psychological.



Bugs and Debugging

The use of the term *bug* in computing has been traced to Grace Murray Hopper during the final days of World War II. On September 9, 1945, she was part of a team at Harvard University, working to build the Mark II, a large relay computer (actually a room-size electronic calculator). It was a hot summer evening and the Mark II's developers had the window open. Suddenly, the device stopped its calculations. The trouble turned out to involve a flip-flop switch (a relay). When the defective relay was located, the team found a moth in it (the first case of a "bug"). "We got a pair of tweezers," wrote programmer Hopper. "Very carefully we took the moth out of the relay, and put it in the logbook, and put scotch tape over it."

After that, whenever Howard Aiken asked if a team was "making any numbers," negative responses were given with explanation "we are debugging the computer."

Fig. 5.1. Bugs and Debugging

The elimination of the syntactical bug is the process of debugging, whereas the detection and elimination of the logical bug is the process of testing. Gruenberger writes, the logical bugs can be extremely subtle and may need a great deal of effort to eliminate them. It is commonly accepted that all large software systems (operating or application) have bugs remaining in them. The number of possible paths through a large computer program is enormous, and it is physically impossible to explore all of them. The single path containing a bug may not be followed in actual production runs for a long time (if ever) after the program has been certified as correct by its author or others.

QUALITY ASSURANCE TESTS

One reason why quality assurance is needed is because computers are infamous for doing what you tell them to do, not necessarily what you want them to do. To close this gap, the code must be free of errors or bugs that cause unexpected results, a process called debugging.

Scenario-based testing, also called usage-based testing, concentrates on what the user does, not what the product does. This means capturing use cases and the tasks users perform, then performing them and their variants as tests. These scenarios also can identify interaction bugs. They often are more complex and realistic than error-based tests. Scenario-based tests tend to exercise multiple subsystems in a single test, because that is what users do. The tests will not find everything, but they will cover at least the higher visibility system interaction bugs.

5.2. TESTING STRATEGIES

The extent of testing a system is controlled by many factors, such as the risks involved, limitations on resources, and deadlines. In light of these issues, we must deploy a testing strategy that does the "best" job of finding defects in a product within the given constraints. There are many testing strategies, but most testing uses a combination of these: black box testing, white box testing, top-down testing, and bottom-up testing. However, no strategy or combination of strategies truly can prove the correctness of a system; it can establish only its "acceptability."

Black Box Testing

The concept of the black box is used to represent a system whose inside workings are not available for inspection. In a black box, the test item is treated as "black," since its logic is unknown; all that is known is what goes in and what comes out, or the input and output (see Figure 13-1). Weinberg describes writing a user manual as an example of a black box approach to requirements. The user manual does not show the internal logic, because the users of the system do not care about what is inside the system.
In black box testing, you try various inputs and examine the resulting output; you can learn what the box does but nothing about how this conversion is implemented. Black box testing works very nicely in testing objects in an object-oriented environment. The black box testing technique also can be used for scenario-based tests, where the system's inside may not be available for inspection but the input and output are defined through use cases or other analysis information.

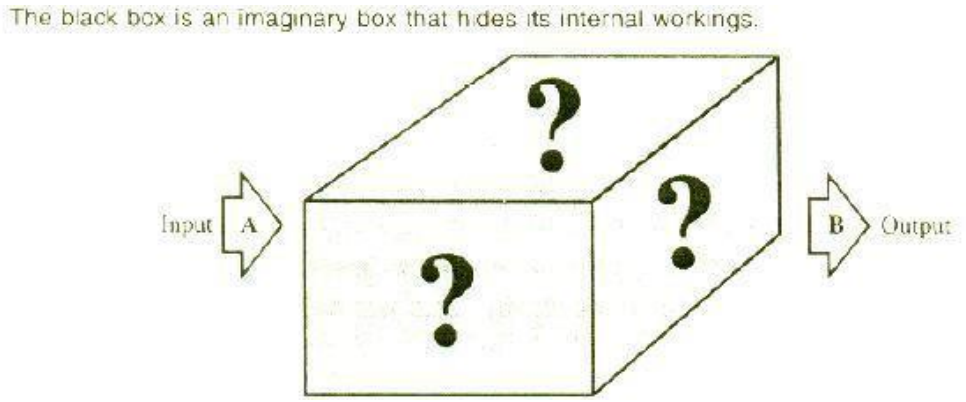

The black box is an imaginary box that hides its internal workings.
Input
A
?
?
?
B
Output

Fig. 5.2. Black Box Testing

White Box Testing

White box testing assumes that the specific logic is important and must be tested to guarantee the system's proper functioning. The main use of the white box is in error-based testing, when you already have tested all objects of an application and all external or public methods of an object that you believe to be of greater importance (see Figure).

In white box testing, you are looking for bugs that have a low probability of execution, have been carelessly implemented, or were overlooked previously. One form of white box testing, called path testing, makes certain that each path in a object's method is executed at least once during testing. Two types of path testing are statement testing coverage and branch testing coverage: Statement testing coverage. The main idea of statement testing coverage is to test every statement in the object's method by executing it at least once. Murray states, "Testing less than this for new software is unconscionable and should be criminalized" [quoted in 2]. However, realistically, it is impossible to test a program on every single input, so you never can be sure that a program will not fail on some input. Branch testing coverage. The main idea behind branch testing coverage is to perform enough tests to ensure that every branch alternative has been executed at least once under some test. As in statement testing coverage, it is unfeasible to fully test any program of considerable size. Most debugging tools are excellent in statement and branch testing coverage. White box testing is useful for error-based testing.

Top-Down Testing

Top-down testing assumes that the main logic or object interactions and systems messages of the application need more testing than an individual object's methods or supporting logic. A top-down strategy can detect the serious design flaws early in the implementation.

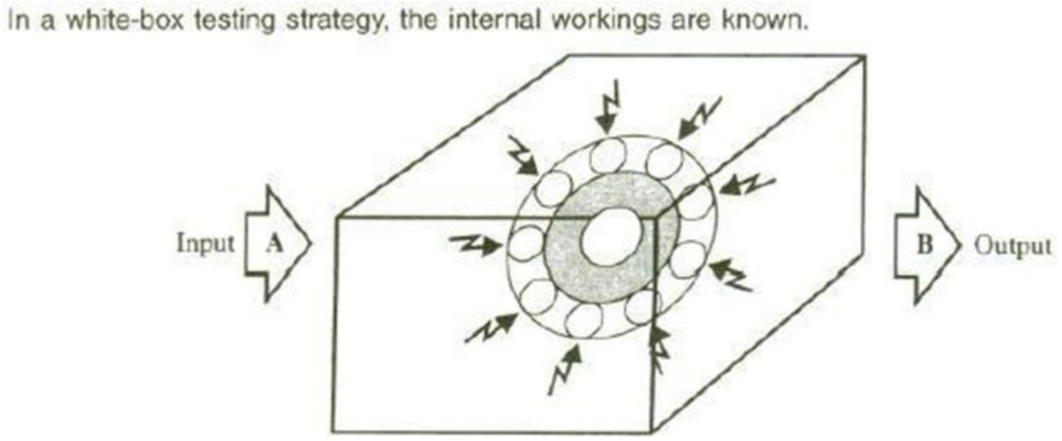

In a white-box testing strategy, the internal workings are known.
Input
A
B
Output

Fig. 5.3. Top-down Testing

In theory, top-down testing should find critical design errors early in the testing process and significantly improve the quality of the delivered software because of the iterative nature of the test. A top-down strategy supports testing the user interface and event-driven systems. Testing the user interface using a top-down approach means testing interface navigation. This serves two purposes, according to Conger. First, the top-down approach can test the navigation through screens and verify that it matches the requirements. Second, users can see, at an early stage, how the final application will look and feel. This approach also is useful for scenario-based testing. Top-down testing is useful to test subsystem and system integration.

Bottom-Up Testing

Bottom-up testing starts with the details of the system and proceeds to higher levels by a progressive aggregation of details until they collectively fit the requirements for the system. This approach is more appropriate for testing the individual objects in a system. Here, you test each object, then combine them and test their interaction and the messages passed among objects by utilizing the top-down approach.

In bottom-up testing, you start with the methods and classes that call or rely on no others. You test them thoroughly. Then you progress to the next level up: those methods and classes that use only the bottom level ones already tested. Next, you test combinations of the bottom two layers. Proceed until you are testing the entire program. This strategy makes sense because you are checking the behavior of a piece of code before it is used by another. Bottom-up testing leads to integration testing, which leads to systems testing.

5.3. TEST CASES

To have a comprehensive testing scheme, the test must cover all methods or a good majority of them. All the services of your system must be checked by at least one test. To test a system, you must construct some test input cases, then describe how the output will look. Next, perform the tests and compare the outcome with the expected output. The good news is that the use cases developed during analysis can be used to describe the usage test cases. After all, tests always should be designed from specifications and not by looking at the product! Myers describes the objective of testing as follows.

Testing is the process of executing a program with the intent of finding errors. A good test case is the one that has a high probability of detecting an as-yet undiscovered error. A successful test case is the one that detects an as-yet undiscovered error.

Guidelines for Developing Quality Assurance Test Cases

Gause and Weinberg provide a wonderful example to highlight the essence of a test case. Say, we want to test our new and improved "Superchalk": Writing a geometry lesson on a blackboard is clearly normal use for Superchalk. Drawing on clothing is not normal, but is quite reasonable to expect. Eating Superchalk may be unreasonable, but the design will have to deal with this issue in some way, in order to prevent lawsuits. No single failure of requirements work leads to more lawsuits than the confident declaration.

Basically, a test case is a set of what-if questions. Freedman and Thomas have developed guidelines that have been adapted for the UA: Describe which feature or service (external or internal) your test attempts to cover. If the test case is based on a use case (i.e., this is a usage test), it is a good idea to refer to the use-case name. Remember that the use cases are the source of test cases. In theory, the software is supposed to match the use cases, not the reverse. As soon as you have enough of use cases, go ahead and write the test plan for that piece. Specify what you are testing and which particular feature (methods). Then, specify what you are going to do to test the feature and what you expect to happen. Test the normal use of the object's methods. Test the abnormal but reasonable use of the object's methods. Test the abnormal and unreasonable use of the object's methods.

Test the boundary conditions. For example, if an edit control accepts 32 characters, try 33, then try 40. Also specify when you expect error dialog boxes, when you expect some default event, and when functionality still is being defined. Test objects' interactions and the messages sent among them. If you have developed sequence diagrams, they can assist you in this process. When the revisions have been made, document the cases so they become the starting basis for the follow-up test.

The internal quality of the software, such as its reusability and extendibility, should be assessed as well. Although the reusability and extend ability are more difficult to test, nevertheless they are extremely important. Software reusability rarely is practiced effectively. The organizations that will survive in the 21st century will be those that have achieved high levels of reusability-anywhere from 70-80 percent or more. Griss argues that, although reuse is widely desired and often the benefit of utilizing object technology, many object-oriented reuse efforts fail because of to narrow a focus on technology rather than the policies set forth by an organization. He recommends an institutionalized approach to software development, in which software assets intentionally are created or acquired to be reusable. These assets then are consistently used and maintained to obtain high levels of reuse, thereby optimizing the organization's ability to produce high-quality software products rapidly and effectively. Your test case may measure what percentage of the system has been reused, say, measured in terms of reused lines of code as opposed to new lines of code written. Specifying results is crucial in developing test cases. You should test cases that are supposed to fail. During such tests, it is a good idea to alert the person running them that failure is expected. Say, we are testing a File Open feature. We need to specify the result as follows:

1. Drop down the File menu and select Open.
2. Try opening the following types of files:

- A file that is there (should work).
- A file that is not there (should get an Error message).
- A file name with international characters (should work).
- A file type that the program does not open (should get a message or conversion dialog box).

5.4. TEST PLANS

On paper, it may seem that everything will fall into place with no preparation and a bug- free product will be shipped. However, in the real world, it may be a good idea to use a test plan to find bugs and remove them. A dreaded and frequently overlooked activity in software development is writing the test plan. A test plan offers a road map for testing activities, whether usability, user satisfaction, or quality assurance tests. It should state the test objectives and how to meet them. The test plan need not be very large; in fact, devoting too much time to the plan can be counterproductive.

The following steps are needed to create a test plan:

1. **Objectives of the test.** Create the objectives and describe how to achieve them. For example, if the objective is usability of the system, that must be stated and also how to realize it.
2. **Development of a test case.** Develop test data, both input and expected output, based on the domain of the data and the expected behaviors that must be tested (more on this in the next section).
3. **Test analysis.** This step involves the examination of the test output and the documentation of the test results. If bugs are detected, then this is reported and the activity centers on debugging. After debugging, steps 1 through 3 must be repeated until no bugs can be detected.

All passed tests should be repeated with the revised program, called regression testing, which can discover errors introduced during the debugging process. When sufficient testing is believed to have been conducted, this fact should be reported, and testing for this specific product is complete.

According to Tamara Thomas, the test planner at Microsoft, a good test plan is one of the strongest tools you might have. It gives you the chance to be clear with other groups or departments about what will be tested, how it will be tested, and the intended schedule. Thomas explains that, with a good, clear test plan, you can assign testing features to other people in an efficient manner. You then can use the plan to track what has been tested, who did the testing, and how the testing was done. You also can use your plan as a checklist, to make sure that you do not forget to test any features.

Who should do the testing? For a small application, the designer or the design team usually will develop the test plan and test cases and, in some situations, actually will perform the tests. However, many organizations have a separate team, such as a quality assurance group, that works closely with the design team and is responsible for these activities (such as developing the test plans and actually performing the tests). Most software companies also use beta testing, a popular, inexpensive, and effective way to test software on a select group of the actual users of the system. This is in contrast to alpha testing, where testing is done by inhouse testers, such as programmers, software engineers, and internal users. If you are going to perform beta testing, make sure to include it in your plan, since it needs to be communicated to your users well in advance of the availability of your application in a beta version.

GUIDELINES FOR DEVELOPING TEST PLANS

As software gains complexity and interaction among programs is more tightly coupled, planning becomes essential. A good test plan not only prevents overlooking a feature (or features), it also helps divide the work load among other people, explains Thomas.

The following guidelines have been developed by Thomas for writing test plans :

- You may have requirements that dictate a specific appearance or format for your test plan. These requirements may be generated by the users. Whatever the appearance of your test plan, try to include as much detail as possible about the tests. The test plan should contain a schedule and a list of required resources. List how many people will be needed, when the testing will be done, and what equipment will be required.
- After you have detained what types of testing are necessary (such as black box, white box, top-down, or bottom-up testing), you need to document specifically what you are going to do. Document every type of test you plan to complete.
- The level of detail in your plan may be driven by several factors, such as the following: How much test time do you have?

- Will you use the test plan as a training tool for newer team members?
- A configuration control system provides a way of tracking the changes to the code. At a minimum, every time the code changes, a record should be kept that tracks which module has been changed, who changed it, and when it was altered, with a comment about why the change was made. Without configuration control, you may have difficulty keeping the testing in line with the changes, since frequent changes may occur without being communicated to the testers.

A well-thought-out design tends to produce better code and result in more, complete testing, so it is a good idea to try to keep the plan up to date. Generally, the older a plan gets, the less useful it becomes. If a test plan is so old that it has become part of the fossil record, it is not terribly useful. As you approach the end of a project, you will have less time to create plans. If you do not take the time to document the work that needs to be done, you risk forgetting something in the mad dash to the finish line. Try to develop a habit of routinely bringing the test plan in sync with the product or product specification. At the end of each month or as you reach each milestone, take time to complete routine updates. This will help you avoid being overwhelmed by being so out of- date that you need to rewrite the whole plan. Keep configuration information on your plan, too. Notes about who made which updates and when can be very helpful down the road

5.5. MYERS'S DEBUGGING PRINCIPLES

The Myers's bug location and debugging principles:

1. **Bug Locating Principles.** Think. If you reach an impasse, sleep on it. If the impasse remains, describe the problem to someone else. Use debugging tools (this is slightly different from Myers's suggestion). Experimentation should be done as a last resort (this is slightly different from Myers's suggestion).
2. **Debugging Principles.** Where there is one bug, there is likely to be another. Fix the error, not just the symptom of it. The probability of the solution being correct drops as the size of the program increases. Beware of the possibility that an error correction will create a new error (this is less of a problem in an object-oriented environment).

CASE STUDY: DEVELOPING TEST CASES FOR THE VIANET BANK ATM SYSTEM

We identified the scenarios or use cases for the ViaNet bank ATM system. The ViaNet bank ATM system has scenarios involving Checking Account, Savings Account, and general Bank Transaction (see Figures. Here again is a list of the use cases that drive many object-oriented activities, including the usability testing:. Bank Transaction (see Figure). Checking Transaction History (see Figure). .Deposit Checking (see Figure).
.Deposit Savings (see Figure). .Savings Transaction History (see Figure). .Withdraw Checking (see Figure). .Withdraw Savings (see Figure). .Valid/Invalid PIN (see Figure).

The activity diagrams and sequence/collaboration diagrams created for these use cases are used to develop the usability test cases. For example, you can draw activity and sequence diagrams to model each scenario that exists when a bank client withdraws, deposits, or needs information on an account. Walking through the steps can assist you in developing a usage test case.

Let us develop a test case for the activities involved in the ATM transaction based on the use cases identified so far. (See the activity diagram in Figure and the sequence diagram of Figure to refresh your memory.)

5.6. SYSTEM USABLILITY AND USER SATISFACTION

Quality refers to the ability of products to meet the users' needs and expectations. The task of satisfying user requirements is the basic motivation for quality. Quality also means striving to do the things right the first time, while always looking to improve how things are being done. Sometimes, this even means spending more time in the initial phases of a project-such as analysis and design-making sure that you are doing the right things. Having to correct fewer problems means significantly less wasted time and capital. When all the losses caused by poor quality are considered, high quality usually costs less than poor quality.

Two main issues in software quality are validation or user satisfaction and verification or quality assurance (see Previous chapter). There are different reasons for testing. You can use testing to look for potential problems in a proposed design. You can focus on comparing two or more designs to determine which is better, given a specific task or set of tasks. Usability testing is different from quality assurance testing in that, rather than finding programming defects, you assess how well the interface or the software fits the use cases, which are the reflections of users' needs and expectations. To ensure user satisfaction, we must measure it throughout the system development with user satisfaction tests. Furthermore, these tests can be used as a communication vehicle between designers and end users. In the next section, we look at user satisfaction tests that can be invaluable in developing high- Once the design is complete, you can walk users through the steps of the scenarios to determine if the design enables the scenarios to occur as planned.

USABILITY TESTING

The International Organization for Standardization (ISO) defines usability as the effectiveness, efficiency; and satisfaction with which a specified set others can achieve a specified set of tasks in particular environments. The ISO definition requires. Defining tasks. What are the tasks? Defining users. Who are the users? A means for measuring effectiveness, efficiency, and satisfaction. How do we measure usability?

The phrase two sides of the same coin is helpful for describing the relationship between the usability and functionality of a system. Both are essential for the development of high-quality software. Usability testing measures the ease of use as well as the degree of comfort and satisfaction users have with the software. Products with poor usability can be difficult to learn, complicated to operate, and misused or not used at all. Therefore, low product usability leads to high costs for users and a bad reputation for the developers. Usability is one of the most crucial factors in the design and development of a product, especially the user interface. Therefore, usability testing must be a key part of the UI design process.

Usability testing should begin in the early stages of product development; for example, it can be used to gather information about how users do their work and find out their tasks, which can complement use cases. You can incorporate your findings into the usability test plan and test cases. As the design progresses, usability testing continues to provide valuable input for analyzing initial design concepts and, in the later stages of product development, can be used to test specific product tasks, especially the ill.

Usability test cases begin with the identification of use cases that can specify the target audience, tasks, and test goals. When designing a test, focus on use cases or tasks, not features. Even if your goal is testing specific features, remember that your users will use them within the context of particular tasks. It also is a good idea to run a pilot test to work the bugs out of the tasks to be tested and make certain the task scenarios, prototype, and test equipment work smoothly. Test cases must include all use cases identified so far. Recall from Previous chapter that the use case can be used through most activities of software development.

Furthermore, by following Jacobson's life cycle model, you can produce designs that are traceable across requirements, analysis, design, implementation, and testing. The main advantage is that all design traces directly back to the user requirements. Use cases and usage scenarios can become test scenarios; and therefore, the use case will drive the usability, user satisfaction, and quality assurance test cases (see Figure).

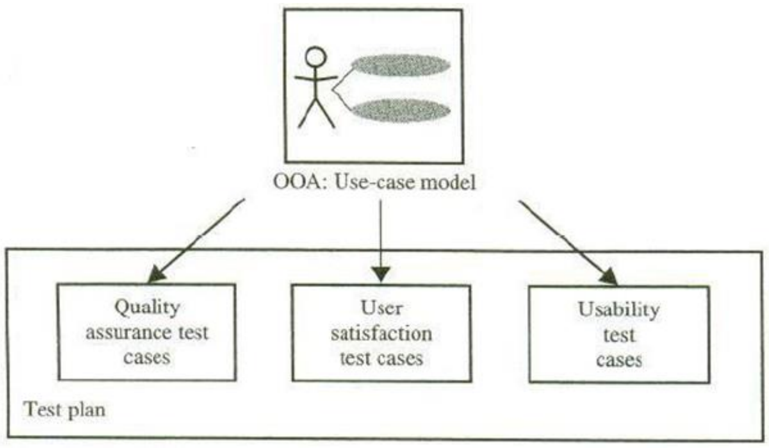


Fig. 5.4. Usability Testing

The use cases identified during analysis can be used in testing the design. Once the design is complete, walk users through the steps of the scenarios to determine if the design enables the scenarios to occur as planned.

GUIDELINES FOR DEVELOPING USABILITY TESTING

Many techniques can be used to gather usability information. In addition to use cases, focus groups can be helpful for generating initial ideas or trying out new ideas. A focus group requires a moderator who directs the discussion about aspects of a task or design but allows participants to freely express their opinions.

Usability tests can be conducted in a one-on-one fashion, as a demonstration, or as a "walk through," in which you take the users through a set of sample scenarios and ask about their impressions along the way. In a technique called the Wizard of OZ, a testing specialist simulates the interaction of an interface. Although these latter techniques can be valuable, they often require a trained, experienced test coordinator 9. Let us take a look at some guidelines for developing usability testing: The usability testing should include a U of a software's components. Usability testing need not be very expensive or elaborate, such as including trained specialists working in a soundproof lab with one-way mirrors and sophisticated recording equipment. Even the small investment of tape recorder, stopwatch, and notepad in an office or conference room can produce excellent results. Similarly, all tests need not involve many subjects. More typically, quick, iterative tests with a small, well-targeted sample of 6 to 10 participants can identify 80-90 percent of most design problems. Consider the user's experience as part of your software usability. You can study 80-90 percent of most design problems with as few as three or four users if you target only a single skill level of users, such as novices or intermediate level users.

RECORDING THE USABILITY TEST

When conducting the usability test, provide an environment comparable to the target setting; usually a quiet location, free from distractions is best. Make participants feel comfortable. It often helps to emphasize that you are testing the software, not the participants. If the participants become confused or frustrated, it is no reflection on them. Unless you have participated yourself, you may be surprised by the pressure many test participants feel. You can alleviate some of the pressure by explaining the testing process and equipment. Tandy Trower, director of the Advanced User Interface group at Microsoft, explains that the users must have reasonable time to try to work through any difficult situation they encounter. Although it generally is best not to interrupt participants during a test, they may get stuck or end up in situations that require intervention. This need not disqualify the test data, as long as the test coordinator carefully guides or hints around a problem. Begin with general hints before moving to specific advice. For more difficult situations, you may need to stop the test and make adjustments. Keep in mind that less intervention usually yields better results. Always record the techniques and search patterns users employ when attempting to work through a difficulty and the number and type of hints you have to provide them.

Ask subjects to think aloud as they work, so you can hear what assumptions and inferences they are making. As the participants work, record the time they take to perform a task as well as any problems they encounter. You may want to follow up the session with the user satisfaction test (more on this in the next section) and a questionnaire that asks the participants to evaluate the product or tasks they performed.

Record the test results using a portable tape recorder or, better, a video camera. Since even the best observer can miss details, reviewing the data later will prove invaluable. Recorded data also allows more direct comparison among multiple participants. It usually is risky to base conclusions on observing a single subject. Recorded data allows the design team to review and evaluate the results.

Whenever possible, involve all members of the design team in observing the test and reviewing the results. This ensures a common reference point and better design solutions as team members apply their own insights to what they observe. If direct observation is not possible, make the recorded results available to the entire team. To ensure user satisfaction and therefore high-quality software, measure user satisfaction along the way as the design takes form. In the next section, we look at the user satisfaction test, which can be an invaluable tool in developing highquality software.

USER SATISFACTION TEST INTRODUCTION

A positive side effect of testing with a prototype is that you can observe how people actually use the software. In addition to prototyping and usability testing, another tool that can assist us in developing high-quality software is measuring and monitoring user satisfaction during software development, especially during the design and development of the user interface.

USER SATISFACTION TEST

User satisfaction testing is the process of quantifying the usability test with some measurable attributes of the test, such as functionality, cost, or ease of use. Usability can be assessed by defining measurable goals, such as .95 percent of users should be able to find how to withdraw money from the ATM machine without error and with no formal training. .70 percent of all users should experience the new function as "a clear improvement over the previous one.”. 90 percent of consumers should be able to operate the VCR within 30 minutes. Furthermore, if the product is being built incrementally, the best measure of user satisfaction is the product itself, since you can observe how users are using it-or avoiding it. Gause and Weinberg have developed a user satisfaction test that can be used along with usability testing. Here are the principal objectives of the user satisfaction test:

As a communication vehicle between designers, as well as between users and designers. To detect and evaluate changes during the design process. To provide a periodic indication of divergence of opinion about the current design. To enable pinpointing specific areas of dissatisfaction for remedy. To provide a clear understanding of just how the completed design is to be evaluated.

Even if the results are never summarized and no one fills out a questionnaire, the process of creating the test itself will provide useful information. Additionally, the test is inexpensive, easy to use, and it is educational to both those who administer it and those who take it.

GUIDELINES FOR DEVELOPING A USER SATISFACTION TEST

The format of every user satisfaction test is basically the same, but its content is different for each project. Once again, the use cases can provide you with an excellent source of information throughout this process. Furthermore, you must work with the users or clients to find out what attributes should be included in the test. Ask the users to select a limited number (5 to 10) of attributes by which the final product can be evaluated. For example, the user might select the following attributes for a customer tracking system: ease of use, functionality, cost, intuitiveness of user interface, and reliability.

A test based on these attributes is shown in Figure. Once these attributes have been identified, they can play a crucial role in the evaluation of the final product. Keep these attributes in the foreground, rather than make assumptions about how the design will be evaluated. The user must use his or her judgment to answer each question by selecting a number between 1 and 10, with 10 as the most favourable and 1 as the least. Comments often are the most significant part of the test. Gause and Weinberg raise the following important point in conducting a user satisfaction test: "When the design of the test has been drafted, show it to the clients and ask, 'If you fill this out monthly (or at whatever interval), will it enable you to express what you like and don't like?' If they answer negatively then find out what attributes would enable them to express themselves and revise the test."

A TOOL FOR ANALYZING USER SATISFACTION: THE USER SATISFACTION TEST TEMPLATE

Commercial off-the-shelf (COTS) software tools are already written and a few are available for analysing and conducting user satisfaction tests. However, here, I have selected an electronic spreadsheet to demonstrate how it can be used to record and analyse the user satisfaction test. The user satisfaction test spreadsheet (USTS) automates many bookkeeping tasks and can assist in analysing the user satisfaction test results. Furthermore, it offers a quick start for creating a user satisfaction test for a particular project.

Recall from the previous section that the tests need not involve many subjects. More typically, quick, iterative tests with a small, well-targeted sample of 6 to 10 participants can identify 80-90 percent of most design problems. The spreadsheet should be designed to record responses from up to 10 users. However, if there are inputs from more than 10 users, it must allow for that (see Figures).

One use of a tool like this is that it shows patterns in user satisfaction level. For example, a shift in the user satisfaction rating indicates that something is happening (see Figure. Gause and Weinberg explain that this shift is sufficient cause to follow up with an interview. The user satisfaction test can be a tool for

Measuring User Satisfaction
Project Name: Customer Tracking System

User	1	2	3	4	5	6	7	8	9
Ease of use	4	7							
Functionality	5	4							
Cost	1	6							
Realiablity	3	4							

Fig. 5.5. User Satisfaction test for a Customer Tracing System

Periodical plotting can reveal shifts in user satisfaction, which can pinpoint a problem-Plotting the high and low responses indicates where to go for maximum information (Gause and Weinberg) finding out what attributes are important or unimportant. An interesting side effect of developing a user satisfaction test is that you benefit from it even if the test is never administered to anyone; it still provides useful information. However, performing the test regularly helps to keep the user involved in the system. It also helps you focus on user wishes. Here is the user satisfaction cycle that has been suggested by Gause and Weinberg:

1. Create a user satisfaction test for your own project. Create a custom form that fits the project's needs and the culture of your organization. Use cases are a great source of information; however, make sure to involve the user in creation of the test.
2. Conduct the test regularly and frequently.
3. Read the comments very carefully, especially if they express a strong feeling.
4. Never forget that feelings are facts, the most important facts you have about the users of the system.
5. Use the information from user satisfaction test, usability test, reactions to prototypes, interviews recorded, and other comments to improve the product.

Another benefit of the user satisfaction test is that you can continue using it even after the product is delivered. The results then become a measure of how well users are learning to use the product and how well it is being maintained. They also provide a starting point for initiating follow-up projects.

CASE STUDY: DEVELOPING USABILITY TEST PLANS AND TEST CASES FOR THE VIANET BANK ATM SYSTEM

In previous chapter, we learned that test plans need not be very large; in fact, devoting too much time to the plans can be counterproductive. Having this in mind let us develop a usability test plan for the ViaNet ATM kiosk by going through the following steps.

DEVELOP TEST OBJECTIVES

The first step is to develop objectives for the test plan. Generally, test objectives are based on the requirements, use cases, or current or desired system usage. In this case, ease of use is the most important requirement, since the ViaNet bank customers should be able to perform their tasks with basically no training and are not expected to read a user manual before withdrawing money from their checking accounts.

Here are the objectives to test the usability of the ViaNet bank ATM kiosk and its user interface: 95 percent of users should be able to find out how to withdraw money from the
ATM machine without error or any formal training. .90 percent of consumers should be able to operate the ATM within 90 seconds.

DEVELOP TEST CASES

Test cases for usability testing are slightly different from test cases for quality assurance. Basically, here, we are not testing the input and expected output but how users interact with the system. Once again, the use cases created during analysis can be used to develop scenarios for the usability test. The usability test scenarios are based on the following use cases:
Deposit Checking (see Figures). Withdraw Checking (see Figures). Deposit Savings
(see Figures). Withdraw Savings (see Figures).
Savings Transaction History (see Figures). Checking Transaction History (see Figures).

Next, we need to select a small number of test participants (6 to 10) who have never before used the kiosk and ask them to perform the following scenarios based on the use case:

1. Deposit $1056.65 to your checking account.
2. Withdraw $40 from your checking account.
3. Deposit $200 to your savings account.
4. Withdraw $55 from savings account.
5. Get your savings account transaction history.
6. Get your checking account transaction history.

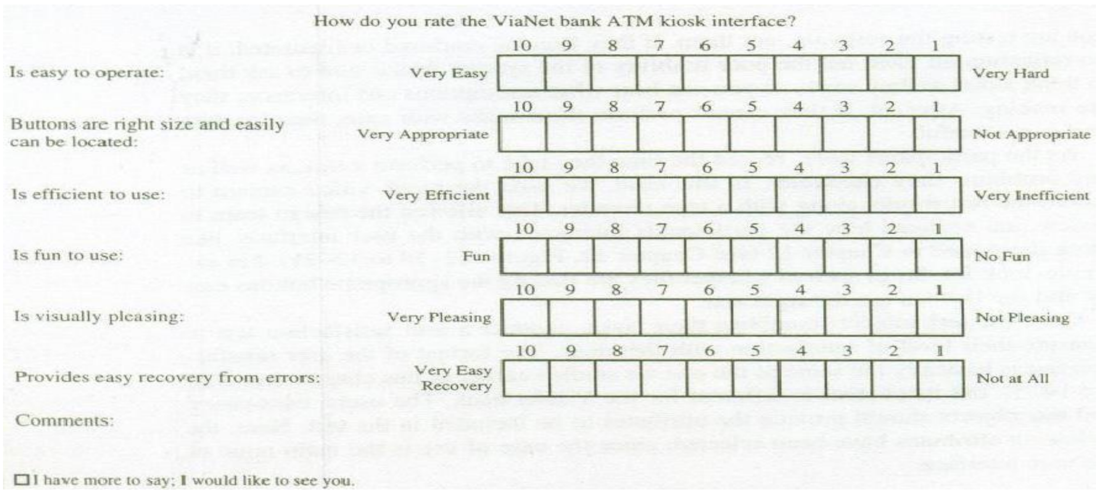
Start by explaining the testing process and equipment to the participants to ease the pressure. Remember to make participants feel comfortable by emphasizing that you are testing the software, not them. If they become confused or frustrated, it is no reflection on them but the poor usability of the system. Make sure to ask them to think aloud as they work, so you can hear what assumptions and inferences they are making. After all, if they cannot perform these tasks with ease, then the system is not useful. As the participants work, record the time they take to perform a task as well as any problems they encounter. In this case, we used the kiosk video...camera to record the test results along with a tape recorder. This allowed the design team to review and evaluate how the participants interacted with the user interface, like those developed in Previous chapter. For example, look for things such as whether they are finding the appropriate buttons easily and the buttons are the right size. Once the test subjects complete their tasks, conduct a user satisfaction test to measure their level of satisfaction with the kiosk.

ANALYZE THE TESTS

The final step is to analyze the tests and document the test results. Here, we need to answer questions such as these: What percentage were able to operate the ATM within 90 seconds or without error? Were the participants able to find out how to withdraw money from the ATM machine with no help? The results of the analysis must be examined.

We also need to analyze the results of user satisfaction tests. The USTS described earlier or a tool similar to it can be used to record and graph the results of user satisfaction tests. As we learned earlier, a shift in user satisfaction pattern indicates that something is happening and a follow-up interview is needed to find out the reasons for the changes. The user satisfaction test can be used as a tool for finding out what attributes are important or unimportant. For example, based 011 the user satisfaction test, we might find that the users do not agree that the system "is efficient to use," and it got a low score.

After the follow-up interviews, it became apparent that participants wanted, in addition to entering the amount for withdrawal, to be able to select from a list with predefined values (say, $20, $40).



How do you rate the ViaNet bank ATM kiosk interface?

		10	9	8	7	6	5	4	3	2	1	
Is easy to operate:	Very Easy											Very Hard
Buttons are right size and easily can be located:	Very Appropriate											Not Appropriate
Is efficient to use:	Very Efficient											Very Inefficient
Is fun to use:	Fun											No Fun
Is visually pleasing:	Very Pleasing											Not Pleasing
Provides easy recovery from errors:	Very Easy Recovery											Not at All

Comments:

☐ I have more to say; I would like to see you.

Fig. 5.6. ViaNet bank ATM Kiosk Interface

TEXT/ REFERENCE BOOKS:

1. Ali Bahrami, “Object oriented systems development using the unified modelling language”, 1st Edition, McGraw-Hill, 1998.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson, “The Unified Modelling Language User Guide”, 3rd Edition Addison Wesley,2007.
3. John Deacon, "Object Oriented Analysis and Design", 1st Edition, Addison Wesley, 2005.
4. Grady Booch, James Rumbaugh, and Ivar Jacobson, “The Unified Modelling Language User Guide”, 3rd Edition Addison Wesley.
5. John Deacon, “Object Oriented Analysis and Design”, 1st Edition, Addison Wesley.
6. Bernd Oestereich, “Developing Software with UML, Object - Oriented Analysis and Design in Practice”, Addison-Wesley

Questions

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Compare Debugging and Testing.	Analysis	BTL 4
2.	Why quality assurance is needed?	Evaluate	BTL 5
3.	Define Blackbox Testing.	Remember	BTL 1
4.	List out the Testing strategies.	Remember	BTL 1
5.	Justify the importance of usability testing.	Evaluate	BTL 5
6.	Define Test case.	Remember	BTL 1
7.	Discuss scenario-based testing?	Evaluate	BTL 5
8.	Define Test Plan.	Remember	BTL 1
9.	What is meant by SQA?	Remember	BTL 1
10.	Compare Alpha and Beta testing.	Analysis	BTL 4
Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Explain Myer’s debugging principles	Analyze	BTL4
2.	Describe the different types of testing strategies	Understand	BTL 2
3.	Explain user satisfaction test with example.	Analyze	BTL4
4.	Explain in detail about (i)Blackbox Testing (ii)Whitebox Testing	Remember	BTL 1

5.	(i) Analyze the guidelines for developing quality assurance Test cases described by Freedman and Thomas. (ii)Explain about Software Quality Assurance.	Analyze	BTL4
6.	Describe the following (i)Debugging (ii)Guidelines for developing Test Plans	Understand	BTL 2
7.	Develop usability test plans and test cases for vianet bank ATM system.	Create	BTL 6