Chapter 1
Introduction to Object Oriented Analysis and Design

1. Analysis and Design

Analysis emphasizes an investigation of the problem and requirements, rather than a solution. For example, if a new online trading system is desired, how will it be used? What are its functions? "Analysis" is a broad term, best qualified, as in requirements analysis (an investigation of the requirements) or object-oriented analysis (an investigation of the domain objects).

Design emphasizes a conceptual solution (in software and hardware) that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects. Design ideas often exclude low-level or "obvious" details obvious to the intended consumers. Ultimately, designs can be implemented, and the implementation (such as code) expresses the true and complete realized design.

As with analysis, the term is best qualified, as in object-oriented design or database design. Useful analysis and design have been summarized in the phrase do the right thing (analysis), and do the thing right (design).

2. Object-Oriented Analysis and Design

In dealing with object-oriented technology, Object-Oriented Analysis and Design is the method of choice for the software development life-cycle. It can be applied in the analysis and design phase and provides general instructions as for what has to be accomplished. In discussing Object-Oriented Analysis and Design the distinction between these two phases has to be clarified first.

In the phase of OOA the typical question starts with What...? like "What will my program need to do?", "What will the classes in my program be?" and "What will each class be responsible for?" . Hence, OOA cares about the real world and how to model this real world without getting into much detail. Larman describes in [Lar02] the OOA phase as an investigation of the problem and requirements, rather than finding a solution to the problem.

In contrast, in the OOD phase, the question typically starts with How...? like "How will this class handle it's responsibilities?", "How to ensure that this class knows all the information it needs?" and "How will classes in the design communicate?" . The OOD phase deals with finding a conceptual solution to the problem – it is about fulfilling the requirements, but not about implementing the solution

During **object-oriented analysis(OOA)** there is an emphasis on finding and describing the objects or concepts in the problem domain. For example, in the case of the flight information system, some of the concepts include *Book, Library,* and *Patron*

During software requirement phase, requirement analysis and object analysis, it is a method of analysis that examines requirements from the perspective of classes and objects as related to problem domain. Object oriented analysis emphasizes the building of real-world model using the object oriented view of the world.

During **object-oriented design (OOD)** (or simply, object design) there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. During user requirement phase, OOD involves understanding of the application domain and build an object model. Identify objects; it is methods of design showing process of object oriented decomposition. Object oriented design is a method of design encompassing the process of objects oriented decomposition and a notation for depicting logical and physical as well as static and dynamic models of the system under design. For example, in the library system, a *Book* software object may have a title attribute and a *getChapter* method.

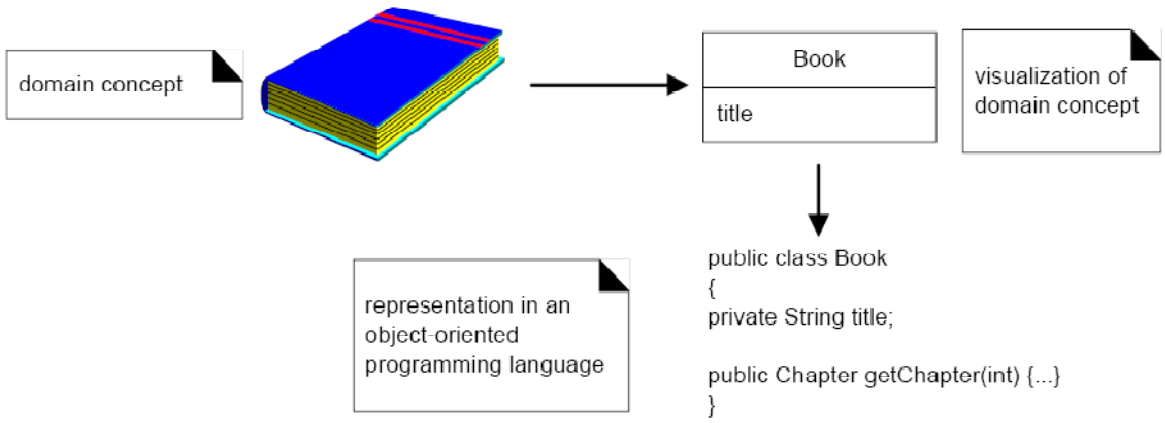


Fig: Object-orientation emphasizes representation of objects.

Finally, during implementation or object-oriented programming, design objects are implemented, such as a *Book* class in Java.

3. OOP (Object oriented programming)

During system implementation phase, t is a method of implementation in which programs are organized as cooperative collection of objects, each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united in inheritance relationships.

Object oriented programming satisfies the following requirements:

- *It supports objects that are data abstractions with an interface of named operations and a hidden local state.*
- *Objects have associated type (class).*
- *Classes may inherit attributes from supertype to subtype.*

4. Analysis Vs Design

Analysis Vs Design	
Object Oriented Analysis ***Identify*** *Classes* • *Who am I?* • *What is the same/different?* • *What do I contain?* • *Who am I associated with?* *Attributes* • *What do I need to know?* *Behaviors* • *What can I do?* *Collaborations* • *What help do I need?* • *Who needs my help?*	***Object Oriented Design*** *Decide how to implement* • *Classes* • *State* • *Behavior* • *Collaborations* *Add Implementation-Specific Components* • *Human interaction* • *Data management* • *Other implementation areas*

How are OOA,OOD and OOP related?

The product of OOA serves as the models from which we may start an OOD , the product of OOD can then be used as blueprint for completely implementing a system using OOP methods. Object-oriented analysis, design and programming are related but distinct OOA is concerned with developing an object model of the application domain OOD is concerned with developing an object-oriented system model to implement requirements OOP is concerned with realising an OOD using an OO programming language such as Java or C++

5. Steps/Activities for Object oriented Analysis

A. Analyze the domain problem
B. Describe the process of systems
C. Identify the objects
D. Specify attributes
E. Defining operations
F. Define and establish Inter-object Communication mechanism

Generic Steps

1. Elicit customer requirements for the system.
2. Identify scenarios for use-cases.
3. Select classes and objects using basic requirements as a guide.
4. Identify attributes and operations for each system object.
5. Define structures and hierarchies that organize classes.
6. Build an object-behavior model.
7. Review the OO analysis model against usec ases or scenarios.

A. Domain Analysis (Domain Class Diagram) –will study later

B. Describe the Process of System

C. Identifying objects

What are physical objects in system ?

- Individuals, organizations, machines, units of information, pictures, whatever makes up application/ make sense in context of real world
- Objects can be:
    - External Entity (e.g., other systems, devices, people) that produce or consume information to be used by system
    - Things (e.g., reports, displays, letters, signals) that are part of information domain for the problem
    - Places (e.g., book's room) that establish the context of the problem and the overall function of the system.
    - Organizational units (e.g., division, group, team, department) that are relevant to an application,
    - Transaction (e.g., loan, take course, buy, order).

Objects help establish workable system so work iteratively between use-case & object models

Example of candidate objects

Just a Line management wishes to increase security, both in their building and on site, without antagonizing their employees. They would also like to prevent people who are not part of the company from using the Just a Line car park.

It has been decide to issue identity cards to all employees, which they are expected to wear while on the Just a Line site. The cards records the name, department and number of the member of staff, and permit access to the Just a Line car park.

A barrier and a card reader are placed at the entrance to the car park. The driver of an approaching car insert his or her numbered card in the card reader, which then checks that the card number is known to the Just a Line system. If the card is recognized, the reader sends a signal to raise the barrier and the car is able to enter the car park.

At the exit, there is also a barrier, which is raised when a car wishes to leave the car park.

When there are no spaces in the car park a sign at the entrance display "Full" and is only switched off when a car leaves.

Special visitor's cards, which record a number and the current date, also permit access to the car park. Visitor's cards may be sent out in advance, or collected from reception. All visitor's cards must be returned to reception when the visitor leaves Just a Line.

Candidate objects in given case are:

Just a Line	management	security	building
site	employee	people	company
car park	card	name	department
number	member of staff	access	barrier
card reader	entrance	driver	car
system	signal	exit	space
sign	visitor	reception	

Candidate objects' rejection rule

- ***Duplicates****: if two or more objects are simply different names for the same thing.*
- ***Irrelevant****: objects which exists in the problem domain, but which are not intended.*
- ***Vague****: when considering words carefully it sometimes becomes clear that they do not have a price meaning and cannot be the basis of a useful in the system.*
- ***General****: the meaning is too broad.*
- ***Attributes****: as the attribute of objects.p*
- ***Associations****: actually represents the relationships between objects.*
- ***Roles****: sometimes objects referred to by the role they play in a particular part of the system.*

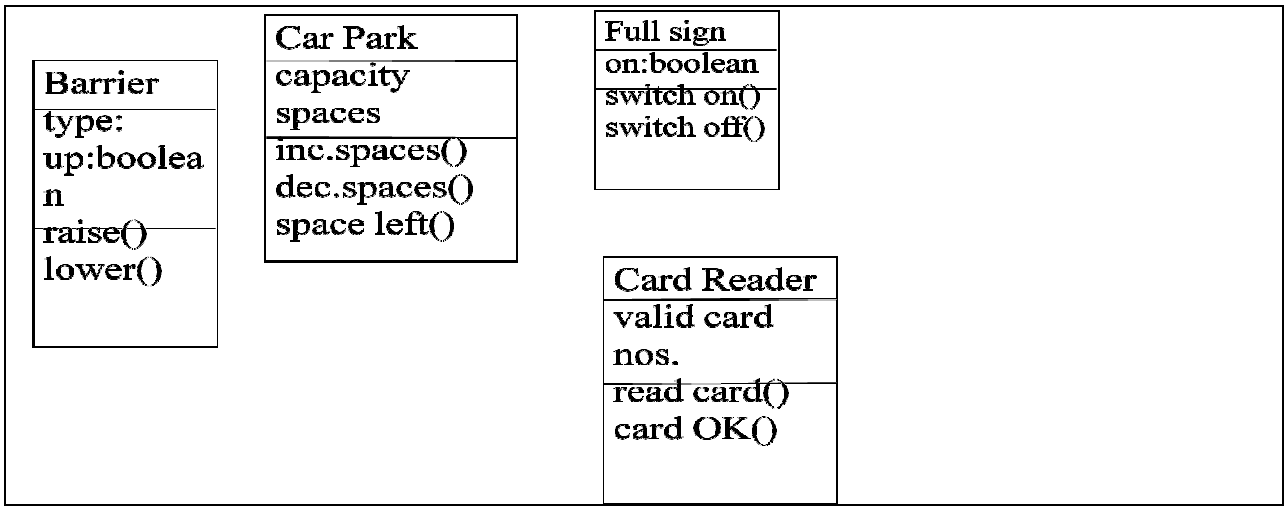
Rejected Candidate objects

Candidate objects	Rejection criteria
Just a Line, member of staff	duplicates with company, employee respectively
management,company, building, site, visitor and reception	irrelevant to the system
security, people	vague
system	too general
name, department,	attribute
access	association
driver	role

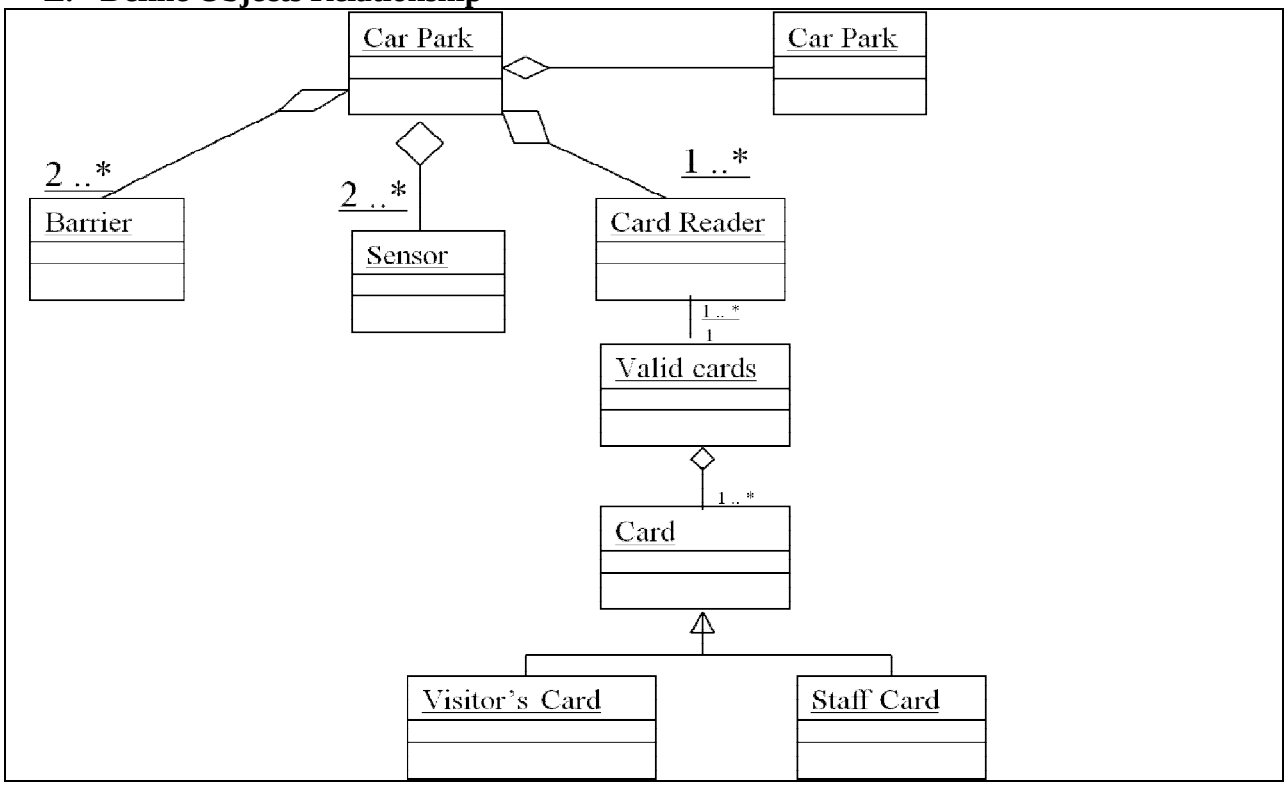
Final Objects

Car park	Staff Card	Visitor's card
Employee	Entrance	exit
card reader	barrier	Full sign
space	sensor	car

D. Define class Attributes and Operations

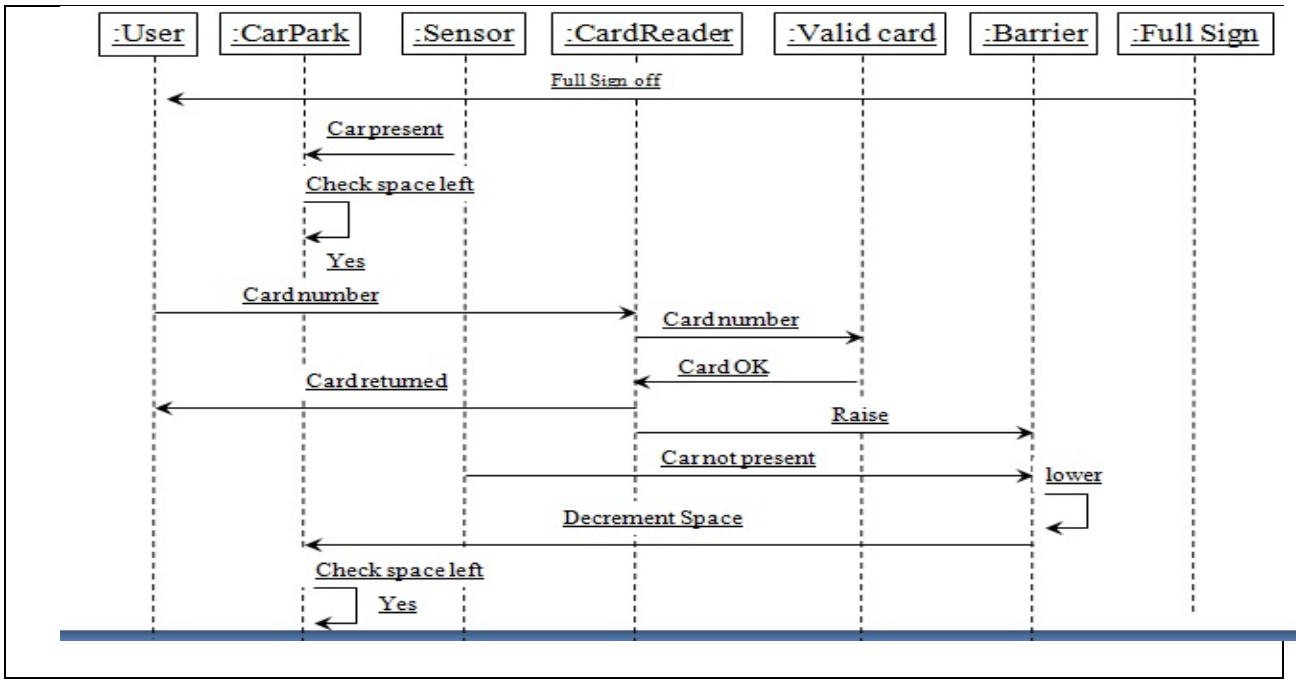


E. Define Objects Relationship



F. Object behaviour modelling(defining inter object communication)

- *A behavioural model shows the interactions between objects to produce some particular system behaviour that is specified as a use-case*
- *Sequence diagrams (or collaboration diagrams) in the UML are used to model interaction between objects*



6. Object Oriented Analysis Approaches (Complementary Analysis Approaches)

A number of additional activities can be used which are complimentary to object-oriented analysis and design methodologies.

A. Domain Analysis.

- Identify classes which are common to all applications in this domain.
- Domain analysis seeks to identify the classes and objects that are common to all applications of a given domain.
    - For example: all accounting systems have certain classes in common.
- Domain analysis works well, because there are very few truly unique kinds of software systems.
- Try to compare the system you are developing to a generalized class of systems.

When starting to design a system.

- Survey the architecture of existing systems in that domain.
- Define key abstractions and mechanisms.
- Evaluate which can be used in the new system.
- A domain expert may be required to assist in this effort.

B. Scenario Planning or Use-Case Analysis.

- Identify operations patterns by users initiating some sequence of interrelated events.
- A use case is a scenario that begins with some user of the system initiating some transaction or sequence of interrelated events.
- Scenario planning of this type is employed by many other OO methodologies, including Coad/Yourdon, Jacobson, and Wirfs-Brock.
- A basic scenario planning approach is introduced under human interaction in the section on design.
- Basic goal is to group the primary function points of the system by related behavior or hierarchy
- This technique is similar to storyboarding.

Use-Case Analysis Activities

For each interesting set of function points, storyboard a scenario to identify the following.

- *Actors that participate.*
- *Actors are classes or categories of users, with a specific role.*
- *Their responsibilities.*
- *How they collaborate with each other.*

Modify the scenarios as necessary.

- *Expand the initial scenarios to identify secondary scenarios.*
- *Create a generalized scenario from several related scenarios.*

Update the list of classes, responsibilities, and behaviors that result for this effort.

C. CRC Cards.

- Class-Responsibility-Collaboration cards.
- CRC stands for Class - Responsibility - Collaboration.
- They are simple 3x5 index cards with the class name along the top and two columns below for responsibilities and collaborations.
- They provide a simple tool for team development tasks, such as storyboarding.
- They can be used during storyboarding sessions to keep track of the classes for each scenario.
- They can be easily grouped and organized.

D. Informal English.

- Identify key concepts from textual description of the problem.
- An alternative is to write an English language description of the problem, or part of the problem.
- Take the natural language description and underline the nouns and verbs to identify candidate classes and operations, respectively.
- This is by no means a rigorous approach.
- Natural language is to imprecise and ambiguous.
- Any noun can be verbed, and any verb can be nouned.

However, since natural language is often used for system descriptions and requirements documents, it may provide some useful insight.

E. **Structured Analysis**

F. **classical categorization**

G. **Behavior Analysis**

7. Object-oriented Design

- OOD transforms the analysis model created using OOA into a design model that serves as a blueprint for software construction.
- OOD results in a design that achieves a number of different levels of modularity.
- Subsystems: Major system components.
- Objects: Data and the operations.
- Four important software design concepts:
  - Abstraction
  - Information Hiding
  - Functional Independence
  - Modularity

- As stated earlier, analysis is the practice of studying a problem domain, leading to a specification of externally observable behavior.

- Building on this, design is the practice of taking a specification of externally observable behavior and adding details needed for actual computer system implementation.
- Analysis is what or the problem phase. Design is the how or the solution phase.
- OOD consists mainly of expanding the requirements model to account for the complexities introduced in selecting a particular implementation.
    - Human interaction.
    - Data management.
    - Other implementation areas.

OOD Goal :

- To design classes identified during analysis phase & user interface
- Identify additional objects & classes that support implementation of requirements
    - E.g. add objects for user interface to system (data entry windows, browse windows)
- Can be intertwined(entangled) with analysis phase
    - Highly incremental, e.g. can start with object-oriented analysis, model it, create object-oriented design, then do some more of each again & again, gradually refining & completing models of system
- Activities & focus of OO analysis & OO design are intertwined

OOD Steps:

- First, build object model based on objects & relationship
- Then iterate & refine model
    - Design & refine classes
    - Design & refine attributes
    - Design & refine methods
    - Design & refine structures
    - Design & refine associations

Guidelines in OOD

- Reuse rather than build new classes : Know existing classes
- Design large number of simple classes rather than small number of complex classes
- Design methods
- Critique what has been proposed : Go back & refine classes

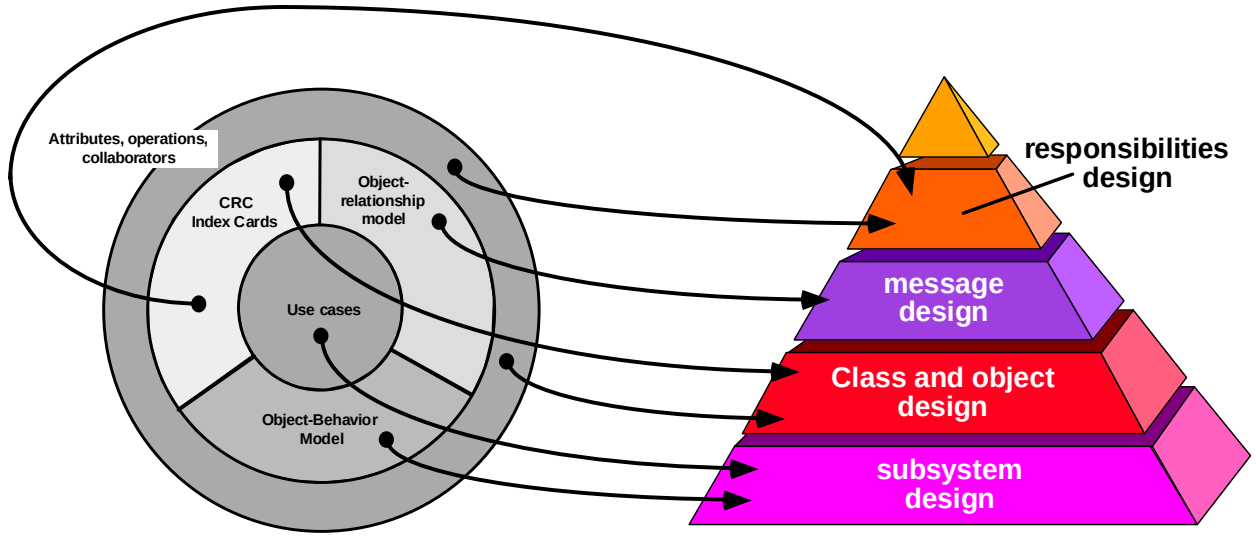
OOD Design Strategy

1. **Problem Domain Component**.
    - The OOA results are placed here directly. Certain modifications are made base on implementation-specific criteria.
2. **Human Interaction Component.**
    - The actual displays and inputs needed for human interaction. The classes will vary somewhat depending upon the type of user interface. Example classes would include specializations of Menu, Screen, and Display.
3. **Data Management Component.**
    - Deals with the access and management of persistent data, using a flat file, relational, or object-oriented database.

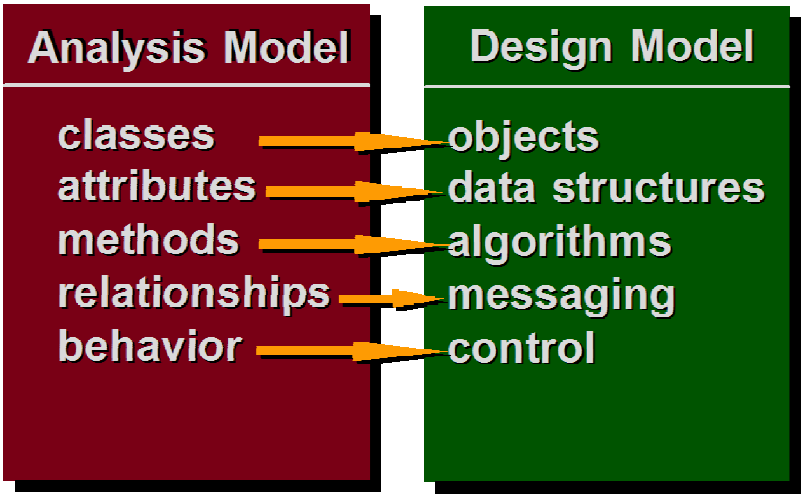
Generic Components for OOD

- Problem domain component—the subsystems that are responsible for implementing customer requirements directly;
- Human interaction component —the subsystems that implement the user interface (this included reusable GUI subsystems);
- Task Management Component—the subsystems that are responsible for controlling and coordinating concurrent tasks that may be packaged within a subsystem or among different subsystems;
- Data management component—the subsystem that is responsible for the storage and retrieval of objects.

OOA to OOD



THE ANALYSIS MODEL THE DESIGN MODEL



8. 0 The models of Object Oriented Development

The models of object oriented analysis and design reflect the importance of explicitly capturing both the class and object hierarchies of the system under design. These models also over the spectrum of the important design decisions that we must consider in developing a complex system and so encourage us to craft implementations that embody the five attributes of well formed complex systems.

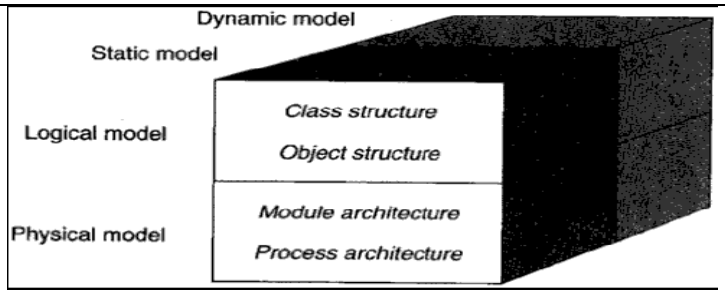


Fig: Different types of model of OO Development

Booch presents a model of object-oriented development that identifies several relevant perspectives

The classes and objects that form the system are identified in a **logical model**. For this logical model, again two different perspectives have to be considered

- A **static perspective** identifies the structure of classes and objects, their properties and the relationships classes and objects participate in.
- A **dynamic model** identifies the dynamic behavior of classes and objects, the different valid states they can be in and the transitions between these states.

Besides the logical model, also a **physical model** needs to be identified. This is usually done later in the system's lifecycle.

The module architecture identifies how classes are kept in separately compliable modules and the process architecture identifies how objects are distributed at run-time over different operating system processes and identifies the relationships between those.

Again for this physical model a static perspective is defined that considers the structure of module and process architecture and a dynamic perspective identifies process and object activation strategies and inter-process communication.

Object-orientation has not, however, emerged fully formed. In fact it has developed over a long period, and continues to change.

- ***Descriptive models*** *written in English are often ambiguous*
- ***Mathematical models*** *often frightens developers though it is good for safety critical system.*
- ***Graphical models*** *can be seen by the user and other developers, e.g. UML*

The Importance of Model Building

- The buildings of models have a broad acceptance among all engineering disciplines largely because model building appeals to the principles of decomposition, abstraction and hierarchy.
- Each model within a design describes a specific aspect/ *perspective* of the system under consideration, when put together will provide an overall view of the system.
- Models give us the opportunity to fail under controlled conditions.

- We evaluate each model under both expected and unusual situations and then after them when they fail to behave as we expect or desire. More than one kind of model is used on order to express all the subtleties of a complex system.

- *Creating a model for a given level of abstraction decides which elements are to be included and which are to be excluded.*
- *The notation often takes the form of graphical symbols and connections.*
- *Models in software help us to visualize, specify, construct and document the artifacts of a software intensive system.*

Business Modeling

- *UML is used for modeling business process.*
- *Business models provide ways of expressing the business processes in terms of business activities and collaborative behavior.*
- *Business modeling is a technique which will help in finding out whether we have identified all the system use cases as well as determining the business value of the system*

Why Business Modeling

- *The system can provide value only if we know how it will be used, who will use it and in what circumstances it will be used.*
- *To ensure that customer-oriented solutions are built, we must not overlook*
  - *the environment in which these systems will work*
  - *the roles and responsibilities of the employees using the system*
  - *the "things" that are handled by the business, as a basis for building the system*
- *great benefits of business modeling*
  - *elicit better system requirements,*
  - *requirements that will drive the creation of information systems that actually fit in the organization and that will indeed be used by end-users*

Key Steps of OOAD

Define Use Cases

Requirements analysis may include a description of related domain processes; these can be written as use cases. Use cases are not an object-oriented artifact—they are simply written stories. However, they are a popular tool in requirements analysis and are an important part of the Unified Process. For example, here is a brief version of the Play a Dice Game use case:

- Play a Dice Game: A player picks up and rolls the dice. If the dice face value total seven, they win; otherwise, they lose.

Define a Domain Model

Object-oriented analysis is concerned with creating a description of the domain from the perspective of objects. There is an identification of the concepts, attributes, and associations that

are considered noteworthy. The result can be expressed in a **domain model** that shows the *noteworthy* domain concepts or objects.

Note that a domain model is not a description of software objects; it is a visualization of the concepts or mental models of a real-world domain. Thus, it has also been called a **conceptual object model**. For example, a partial domain model is shown in

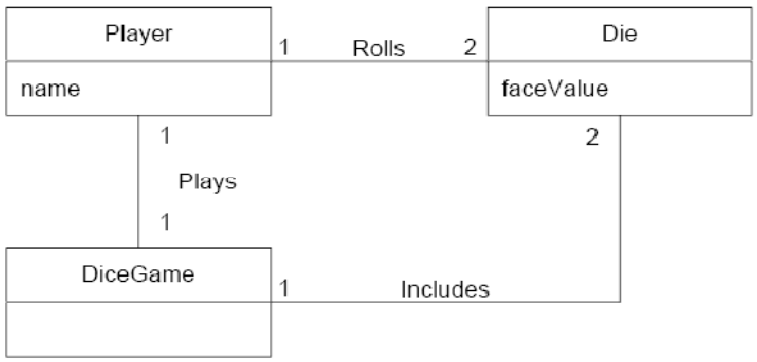


Fig: Partial domain model of the dice game

This model illustrates the noteworthy concepts *Player*, *Die*, and *DiceGame*, with their associations and attributes**.**

Define Interaction Diagrams

Object-oriented design is concerned with defining software objects their responsibilities and collaborations. A common notation to illustrate these collaborations is the sequence diagram (a kind of UML interaction diagram). It shows the flow of messages between software objects, and thus the invocation of methods.
For example, the sequence diagram in Figure illustrates an OO software design, by sending messages to instances of the *DiceGame* and *Die* classes. Note this illustrates a common real-world way the UML is applied: by sketching on a whiteboard.

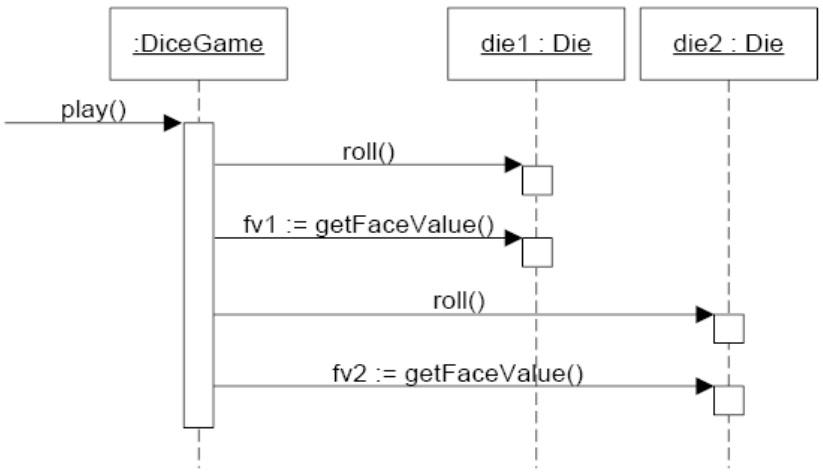


Fig: Interaction diagram illustrating messages between software objects.

Notice that although in the real world a *player* rolls the dice, in the software design the *DiceGame* object "rolls" the dice (that is, sends messages to *Die* objects). Software object designs and programs do take some inspiration from real-world domains, but they are *not* direct models or simulations of the real world.

Define Design Class Diagrams

In addition to a *dynamic* view of collaborating objects shown in interaction diagrams, a *static* view of the class definitions is usefully shown with a **design class diagram**. This illustrates the attributes and methods of the classes.

For example, in the dice game, an inspection of the sequence diagram leads to the partial design class diagram shown in fig below. Since a play message is sent to a **DiceGame** object, the **DiceGame** class requires a play method, while class Die requires a roll and **getFaceValue** method.

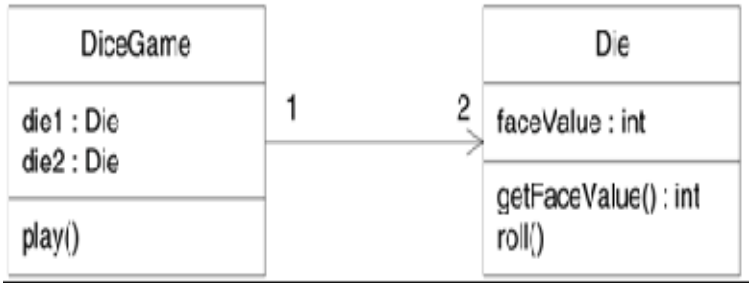


Fig: design class diagram.

In contrast to the domain model showing real-world classes, this diagram shows software classes. Notice that although this design class diagram is not the same as the domain model, some class names and content are similar.

In this way, OO designs and languages can support a **lower representational gap** between the software components and our mental models of a domain. That improves comprehension.

9. Case Studies:

In this section we will present two different types of case study, one is business transaction type problem and other is computer simulation game.

A. Case One: The NextGen POS System

A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).

A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.

Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.

Using an iterative development strategy, we are going to proceed through requirements, object-oriented

analysis, design, and implementation.

B. **Case Two: The Monopoly Game System**

To show that the same practices of OOA/D can apply to very different problems, I've chosen a software version of the game of Monopoly® as another case study. Although the domain and requirements are not at all like a business system such as the NextGen POS, we will see that domain modeling, object design with patterns, and applying the UML are still relevant and useful. As with a POS, software versions of Monopoly are truly developed and sold, with both rich client and Web UIs.

The software version of the game will run as a simulation. One person will start the game and indicate the number of simulated players, and then watch while the game runs to completion, presenting a trace of the activity during the simulated player turns. [Read Rule of monopoly game from different websites]

10. Use Cases

OOD (and all software design) is strongly related to the prerequisite activity of requirements analysis, which often includes writing use cases. Use Case, is a name for a scenario to describe the user–computer system interaction. Use Cases are used to determine system requirements; identify classes & their relationship to other classes in domain.

Use cases are text stories, widely used to discover and record requirements. They influence many aspects of a project including OOA/D and will be input to many subsequent artifacts in the case studies.

Requirements analysis may include stories or scenarios of how people use the application; these can be written as **use cases**.

Use cases are not an object-oriented artifact they are simply written stories. However, they are a popular tool in requirements analysis. For example, here is a brief version of the *Play a Dice Game* use case:

Play a Dice Game: Player requests to roll the dice. System presents results: If the dice face value totals seven, player wins; otherwise, player loses.

To understand system requirements:

- Need to identify the users or actors
    - Who are the actors?
    - How do they use system?

Use case is typical interaction between user & system that captures users' goal & needs. In simple, usage, capture use case by talking to typical users, discussing various things they might want to do with system. Use cases can be used to examine who does what in interactions among objects, what role they play, intersection among objects' role to achieve given goal is called

collaboration. Several scenarios (usual & unusual behavior, exceptions) needed to understand all aspects of collaboration & all potential actions.

Use case modeling: expressing high level processes & interactions with customers in a scenario & analyzing it .It gives system uses, system responsibilities

Developing use case is iterative process, when use case model better understood & developed, start identifying classes & create their relationship

Use case modeling:

- Helps in capturing requirements
- Helps in planning iterations of development
- Helps in validating systems
- expressing high level processes & interactions with customers in a scenario & analyzing it
- It gives system uses, system responsibilities

Scenarios vs Usecase:

- Great way to establish communication with client
- Different types of scenarios: As-Is, visionary, evaluation and training

Use cases

- Abstractions of scenarios
- Use cases bridge the transition between functional requirements and objects.

How to write a use case

- Name of Use Case
- Actors
  - Description of Actors involved in use case
- Entry condition
  - "This use case starts when…"
- Flow of Events
  - Free form, informal natural language
- Exit condition
  - "This use cases terminates when…"
- Exceptions
  - Describe what happens if things go wrong
- Special Requirements
  - Nonfunctional Requirements, Constraints

Format of Use case

Black-box versus white-box visibility type, use cases are written in varying degrees of formality:

- **Brief**—terse one-paragraph summary, usually of the main success scenario. The prior Process Sale example was brief.

- **Casual**—informal paragraph format. Multiple paragraphs that cover various scenarios. The prior Handle Returns example was casual.
- **Fully dressed**—the most elaborate. All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.

Fully Dressed use case format

Various format templates re available for fully dressed use cases. However, perhaps the most widely used and shared format is the template available at www.usecases.org. The following example illustrates this style.

Use Case ID:	
Use Case Name:	
Primary Actor:	
Stakeholders and Interests::	
Preconditions:	
Success Guarantee(Postconditions):	
Main Success Scenario (or Basic Flow)::	
Extensions (or Alternative Flows)::	
Special Requirements:	
Technology and Data Variations List::	
Frequency of Occurrence::	
Open Issues	

Based on a real POS system's requirements (example of use case text story)

Use case ID	UC1
Use case Name	Process Sale
Primary Actor	Cashier
Stakeholders and Interests	- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer short ages are deducted from his/her salary. - Salesperson: Wants sales commissions updated. - Customer: Wants purchase and fast service with minimal effort. Wants proof of purchase to support returns. - Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory. - Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county. - Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.
Preconditions	Cashier is identified and authenticated
Success Guarantee (Postconditions)	Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.
Main Success Scenario (or Basic Flow)	1. Customer arrives at POS checkout with goods and/or services to purchase. 2. Cashier starts a new sale. 3. Cashier enters item identifier. 4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules. *Cashier repeats steps 3-4 until indicates done.* 5. System presents total with taxes calculated. 6. Cashier tells Customer the total, and asks for payment. 7. Customer pays and System handles payment. 8. System logs completed sale and sends sale and payment information to the external

	Accounting system (for accounting and commissions) and Inventory system (to update inventory). 9. System presents receipt. 10. Customer leaves with receipt and goods (if any).
Extensions (or Alternative Flows):	*a. At any time, System fails: To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario. 1. Cashier restarts System, logs in, and requests recovery of prior state. 2. System reconstructs prior state. 2a. System detects anomalies preventing recovery: 1. System signals error to the Cashier, records the error, and enters a clean state. 2. Cashier starts a new sale. 3a. Invalid identifier: 1. System signals error and rejects entry. 3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers): 1. Cashier can enter item category identifier and the quantity. 3-6a: Customer asks Cashier to remove an item from the purchase: 1. Cashier enters item identifier for removal from sale. 2. System displays updated running total. 3-6b. Customer tells Cashier to cancel sale: 1. Cashier cancels sale on System. 3-6c. Cashier suspends the sale: 1. System records sale so that it is available for retrieval on any POS terminal. 4a. The system generated item price is not wanted (e.g., Customer complained about something and is offered a lower price): 1. Cashier enters override price. 2. System presents new price. 5a. System detects failure to communicate with external tax calculation system service: 1. System restarts the service on the POS node, and continues. 1a. System detects that the service does not restart. 1. System signals error. 2. Cashier may manually calculate and enter the tax, or cancel the sale. 5b. Customer says they are eligible for a discount (e.g., employee, preferred customer): 1. Cashier signals discount request. 2. Cashier enters Customer identification. 3. System presents discount total, based on discount rules. 5c. Customer says they have credit in their account, to apply to the sale: 1. Cashier signals credit request. 2. Cashier enters Customer identification. 3. Systems applies credit up to price=0, and reduces remaining credit. 6a. Customer says they intended to pay by cash but don't have enough cash: 1a. Customer uses an alternate payment method. 1b. Customer tells Cashier to cancel sale. Cashier cancels sale on System. 7a. Paying by cash: 1. Cashier enters the cash amount tendered. 2. System presents the balance due, and releases the cash drawer. 3. Cashier deposits cash tendered and returns balance in cash to Customer. 4. System records the cash payment. 7b. Paying by credit: 1. Customer enters their credit account information. 2. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval. 2a. System detects failure to collaborate with external system: 1. System signals error to Cashier. 2. Cashier asks Customer for alternate payment. 3. System receives payment approval and signals approval to Cashier. 3a. System receives payment denial: 1. System signals denial to Cashier. 2. Cashier asks Customer for alternate payment. 4. System records the credit payment, which includes the payment approval. 5. System presents credit payment signature input mechanism. 6. Cashier asks Customer for a credit payment signature. Customer enters signature. 7c. Paying by check...

	7d. Paying by debit... 7e. Customer presents coupons: 1. Before handling payment, Cashier records each coupon and System reduces price as appropriate. System records the used coupons for accounting reasons. 1a. Coupon entered is not for any purchased item: 1. System signals error to Cashier. 9a. There are product rebates: 1. System presents the rebate forms and rebate receipts for each item with a rebate. 9b. Customer requests gift receipt (no prices visible): 1. Cashier requests gift receipt and System presents it.
Special Requirements:	- Touch screen Ul on a large flat panel monitor. Text must be visible from 1 meter. - Credit authorization response within 30 seconds 90% of the time. - Somehow, we want robust recovery when access to remote services such the inventory system is failing. - Language internationalization on the text displayed. - Pluggable business rules to be insertable at steps 3 and 7.
Technology and Data Variations List:	3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard. 3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme. 7a. Credit account information entered by card reader or keyboard. 7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.
Frequency of Occurrence:	Could be nearly continuous
Open Issues	- What are the tax law variations? - Explore the remote service recovery issue. - What customization is needed for different businesses? - Must a cashier take their cash drawer when they log out? - Can the customer directly use the card reader, or does the cashier have to do it?

Example 2

Use case #6

Use case name: PauseGame

Participating actors: Player

Entry condition: Player is already playing the game.

Exit condition:

- Player continues to play the game, OR
- Player can choose to return to the main menu, OR
- Player can exit from the whole game, OR
- Player can load another game, if there is already a game saved.

Main Flow of Events:

1. Player presses the pause button during the game.
2. The system displays the in-game menu.
3. Player continues the game.

Alternative Flow of Event:

- Player choses to save the game and returns to the in-game menu.
- Player choses to return to the main menu.
- Player directly exits the game instead of continuing the game.
- Player loads another game. (go to step 3) If there is no saved game load button will be disabled and warning massage will be shown.

A use case describes three things:

- An actor (user) that initiates an event
- An event that triggers a use case
- The use case that performs the actions triggered by the event

There are two kinds of use cases:

- Primary, the standard flow of events within a system that describe a standard system behavior
- Use case scenarios that describe variations of the primary use case

Steps for Creating a Use Case Model

The steps required to create a use case model are

- *Review the business specifications and identify the actors within the problem domain*

- *Identify the high-level events and develop the primary use cases that describe the events and how actors initiate them*
- *Review each primary use case to determine possible variations of flow through the use case*
- *Develop the use case documents for all primary use cases and all important use case scenarios*
- *Move to UML diagramming techniques to complete the systems analysis and design*

Use case diagrams:

Use case diagrams give us that capability. Use case diagrams are used to depict the context of the system to be built and the functionality provided by that system. They depict who (or what) interacts with the system. They show what the outside world wants the system to do.

Essentials Elements of use case diagrams:

a. Actors:

An actor is anything with behavior, including the system under discussion (SuD) itself when it calls upon the services of other systems.[6] Primary and supporting actors will appear in the action steps of the use case text. Actors are not only roles played by people, but organizations, software, and machines.

There are three kinds of **external actors** in relation to the SuD:

Primary actor *has user goals fulfilled through using services of the SuD.*

For example, the cashier

Why identify? To find user goals, which drive the use cases.

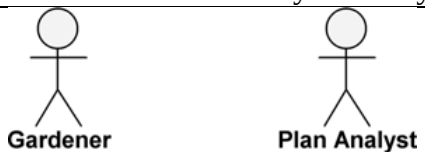
Supporting actor *provides a service (for example, information) to the SuD. The automated payment authorization service is an example. Often a computer system, but could be an organization or person.*

Why identify? To clarify external interfaces and protocols.

Offstage actor *has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.*

Why identify? To ensure that all necessary interests are identified and satisfied. Offstage actor interests are sometimes subtle or easy to miss unless these actors are explicitly named.

Actors are entities that interface with the system. They can be people or other systems. Actors, which are external to the system they are using, are depicted as stylized stick figures.



Actor

Figure shows two actors for the Hydroponics Gardening System we discussed earlier. One way to think of actors is to consider the roles the actors play.
In the real world, people (and systems) may serve in many different roles; for example, a person can be a salesperson, a manager, a father, an artist, and so forth.

b. Use Cases

Use cases represent *what* the actors want your system to do for them.

A use case is a specific way of using the system by using some part of the functionality. A use case is thus a special sequence of related transactions performed by an actor and the system in a dialogue. Each use case is a complete course of events in the system from a user's perspective.

Fig: Use case

Figure depicts some use cases, shown as ovals, for the Hydroponics Gardening System. Use cases are not just any capability that your system may provide. A use case must be a complete flow of activity, from the actor's point of view, that provides value to the actor.

c. The Use Case Diagram

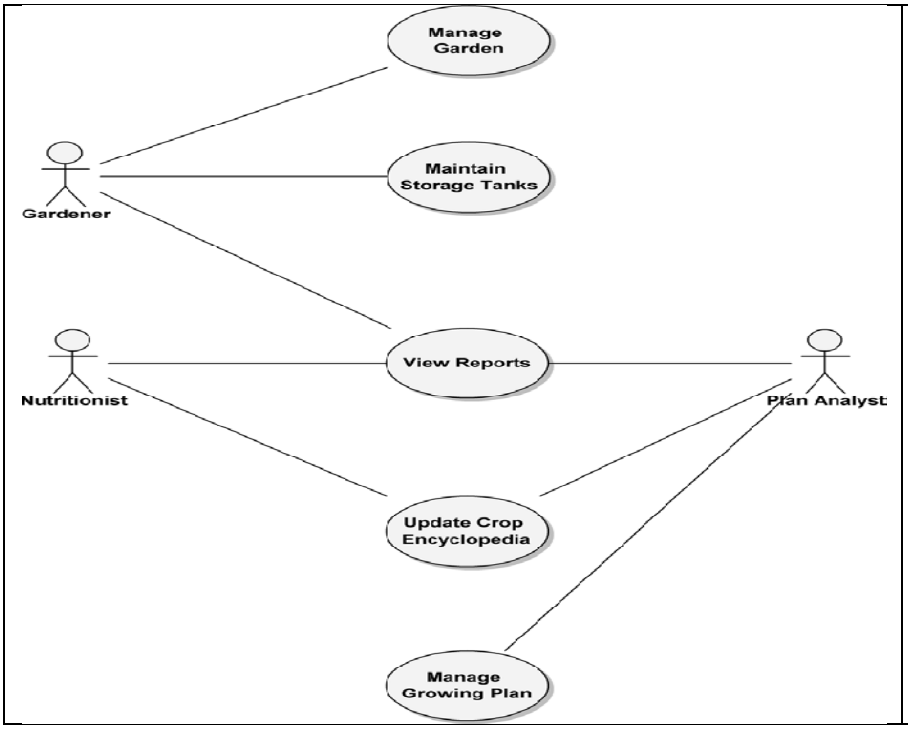


Fig: Use case diagrams for gardening system

To show which actors use which use cases, you can create a use case diagram by connecting them via basic associations, shown by lines, as in Figure below. The associations in the use case diagram indicate which actors initiate which use cases. Here we see that only the Gardener actor can maintain the storage tanks, but all the actors may view reports

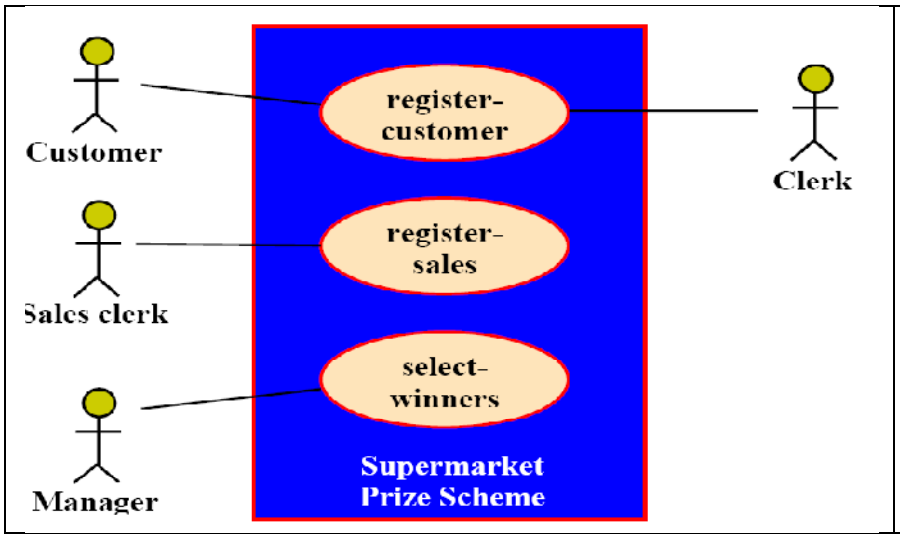


Fig: Example of Use case model (Super Market Prize scheme)

Relationship in Use Case Diagram/ Advanced Concepts in Use case diagrams:

Relationship in Use case diagrams:

- Association
- Include
- Extends
- Generalization

a. «include» Relationships

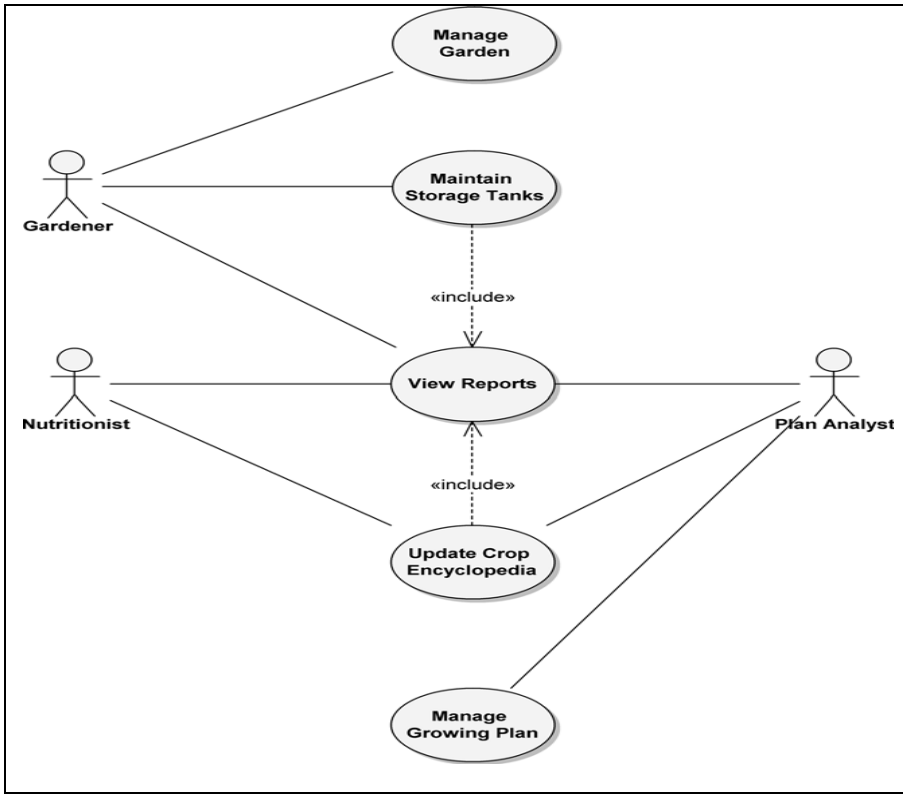


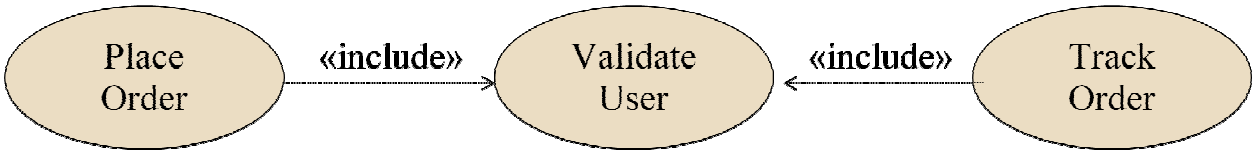
Fig: Example of Include Relationship

This diagram states, for example, that the Update Crop Encyclopedia use case includes the View Reports use case. This means that View Reports must be executed when Update Crop Encyclopedia is executed. Update Crop Encyclopedia would not be considered complete without View Reports.

Where an included use case is executed, it is indicated in the use case specification as an inclusion point. The inclusion point specifies where, in the flow of the including use case, the included use case is to be executed.

Include: A use cases may contain the functionality of another use case. In general it is assumed that any included use case will be called every time the basic path is run.

Dotted line labelled <<include>> beginning at base use case and ending with an arrows pointing to the include use case.



b. <<Extends>> relationship:

While developing your use cases, you may find that certain activities might be performed as part of the use case but are not mandatory for that use case to run successfully.

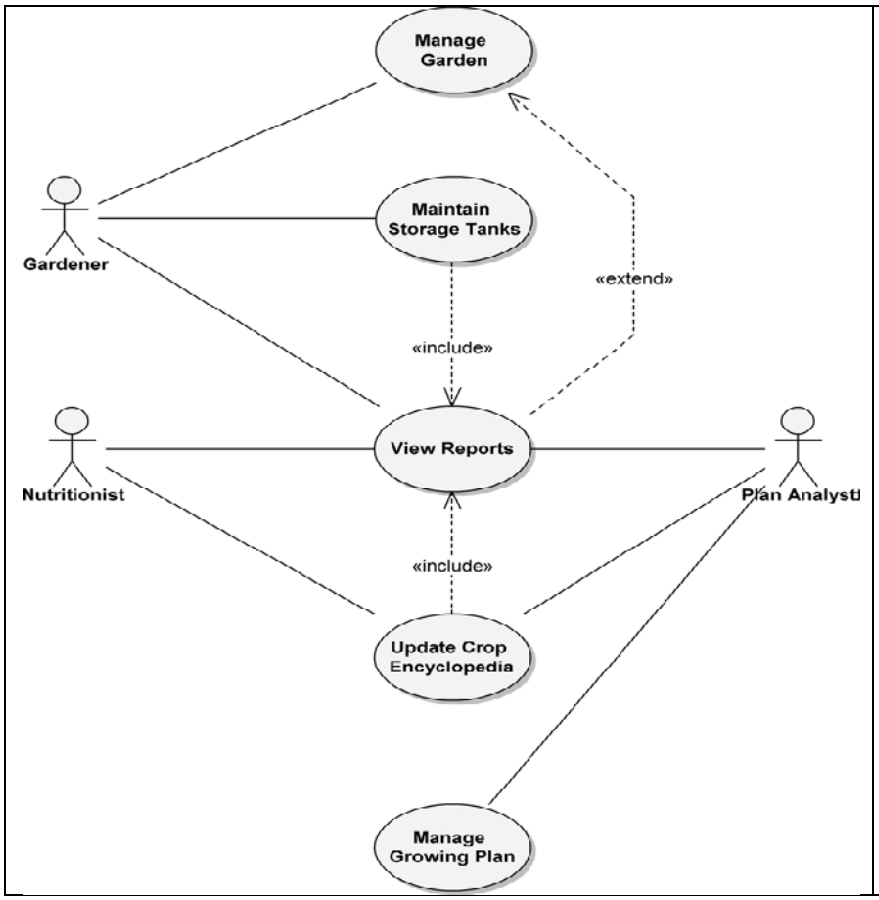
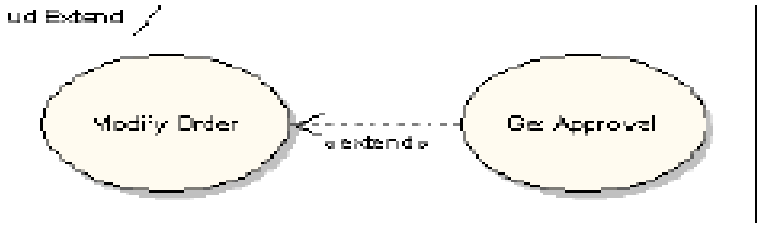


Fig: Example of extends relationship

Extend: *One use case may be used to extend the behaviour of another usecase. This is typically used in exceptional circumstances.*

Dotted line labelled <<extend>> with an arrow toward the base case. The extending use case may add behaviour to the base use case.

For example, if before modifying a particular type of customer order, a user must get approval from some higher authority, then the <Get Approval> use case may optionally extend the regular <Modify Order> use case.



Key Difference between extends and include

	Included Use Case	Extending Use Case
Is this use case optional?	No	Yes
Is the base use case complete without this use case?	No	Yes
Is the execution of this use case conditional?	No	Yes
Does this use case change the behavior of the base use case?	No	Yes

Generalization relationship

Generalization relationships, as described in Chapter 3, can also be used to relate use cases. As with classes, use cases can have common behaviors that other use cases (i.e., child use cases) can modify by adding steps or refining others For example, Fig 1.6 shows the use cases for purchasing various tickets. Purchase Ticket contains the basic steps necessary for purchasing any tickets, while the child use cases specialize Purchase Ticket for the specific kinds of tickets being purchased.

Generalization: relationship between one general use case and a special use case. Represented by a line with a triangular arrow head toward the parent use case.

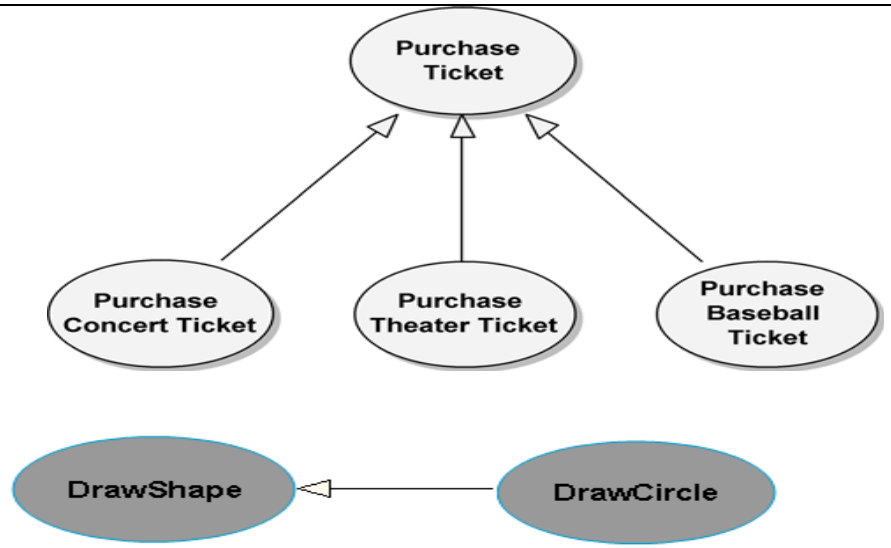


Fig: Examples of Generalization Relationship

Association: communication between an actor and a use case is represented by a solid line.

Examples of Use case Diagram

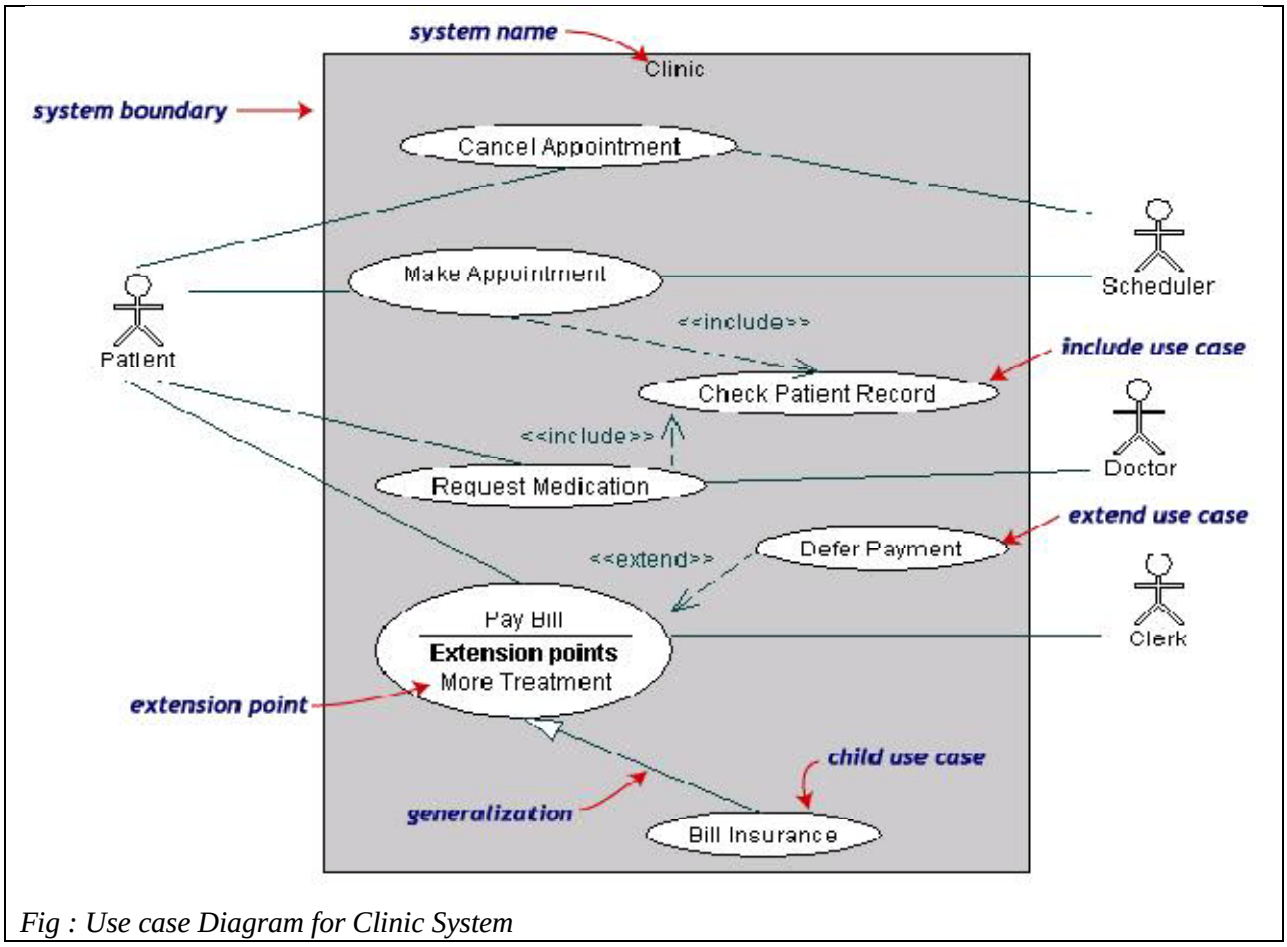


Fig : Use case Diagram for Clinic System

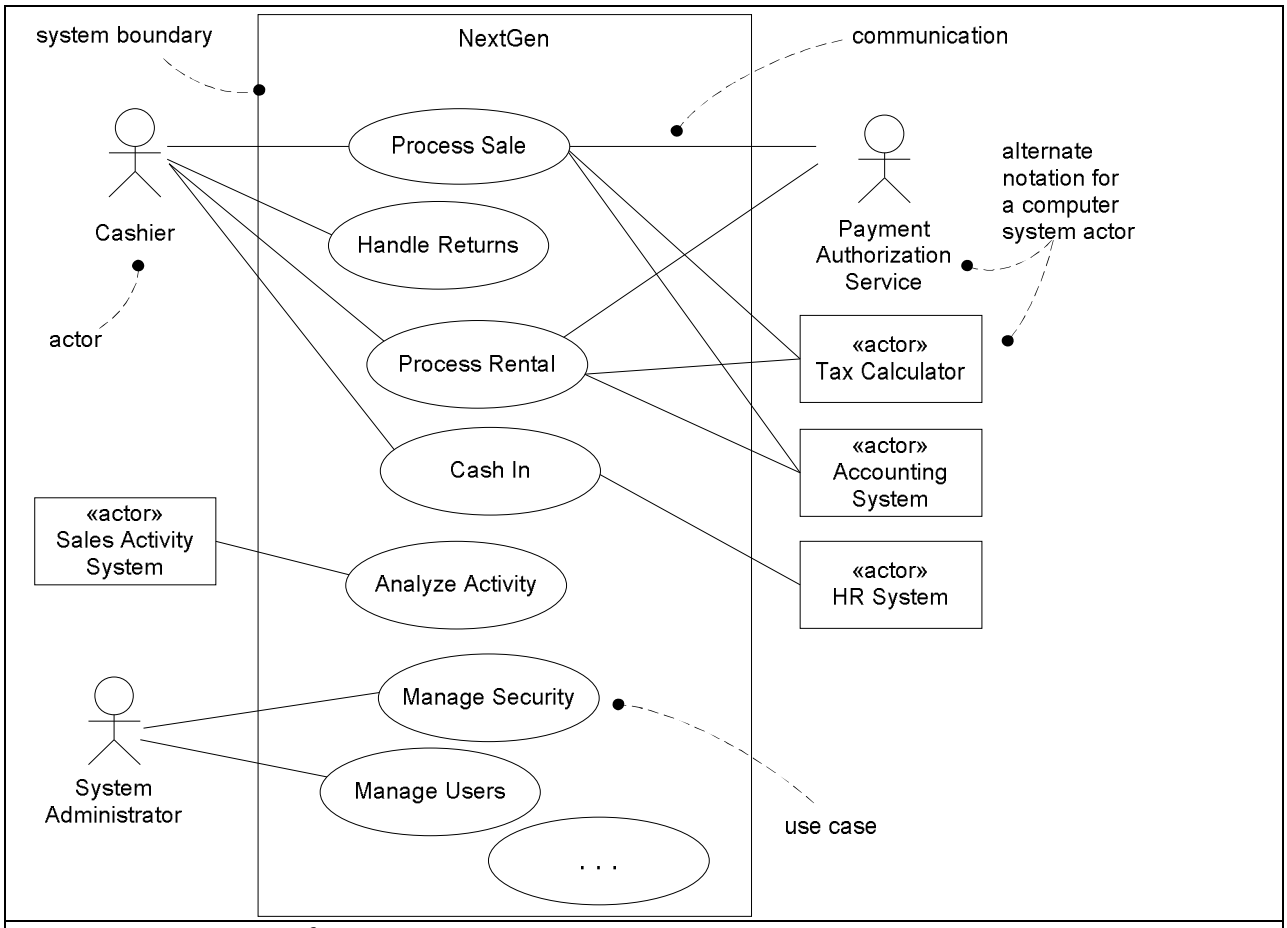
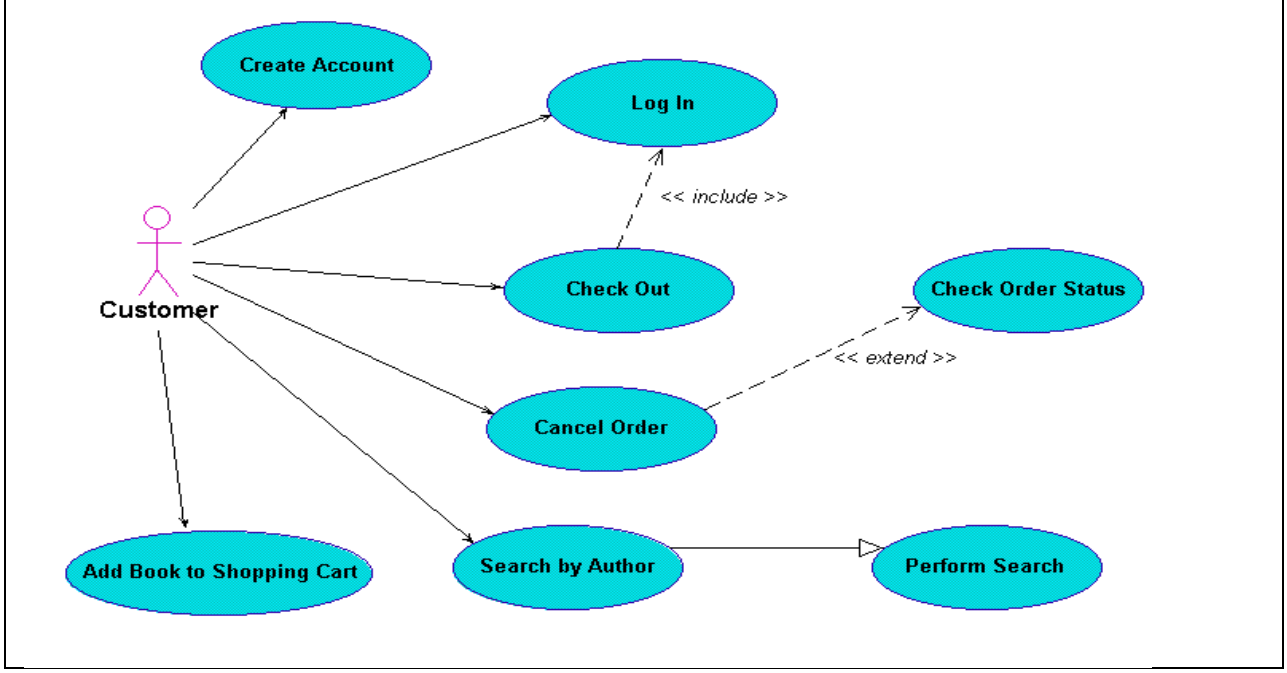
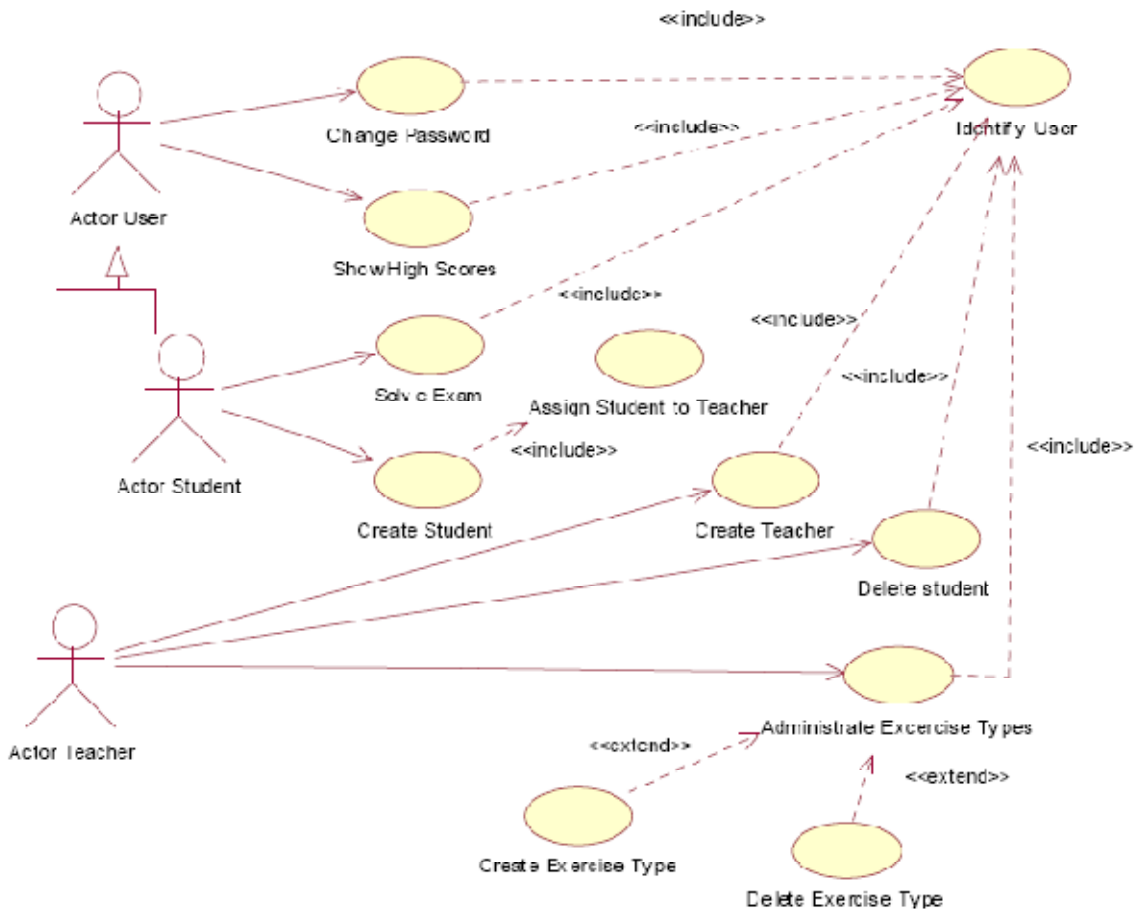


Fig: Use case Diagram for NexGen POS System





The Requirements of MathTrainer
Highlighted verbs in the requirements of the MathTrainer – example

*MathTrainer **aids in perfecting** the mental arithmetic of elementary school students.*
*MathTrainer **poses** each student ten random arithmetical exercises, which **should be solved** as fast and correct as possible. From the responses **scores are collected** which **can be viewed** by the users of MathTrainer.*
*Teachers **can define** types of exercises by **determining** numerical ranges and allowed mathematical operations. They can also **delete types** which were **defined by** themselves.*
*Students are **assigned to** their teacher and can request exercises for an exercise type of their teacher. New teachers and students are able to **apply as** new MathTrainer users themselves by specifying username and password – this **is done** in the context of the usual user identification. The password **can be changed** anytime. Teachers **can delete** the students which **are assigned** to them.*
*During the realization of a test (ten exercises) the time needed for each exercise **is stopped** . However, the scores **are based** on the cumulative time.*

11. What is the UML?

UML, as the name implies, is a modeling language. The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems. The word *visual* in the definition is a key point the UML is the de facto standard *diagramming notation* for drawing or presenting pictures (with some text) related to software primarily OO software.

The UML defines various UML profiles that specialize subsets of the notation for common subject areas. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences).

Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology.

11.1 Origin of UML

In the late 1980s and early 1990s, there was a proliferation of object-oriented design techniques and notations. Different software development houses were using different notations to document their object-oriented designs. These diverse notations used to give rise to a lot of confusion.

UML was developed to standardize the large number of object-oriented modeling notations that existed and were used extensively in the early 1990s. The principles ones in use were:

- *Object Management Technology [Rumbaugh 1991]*
- *Booch's methodology [Booch 1991]*
- *Object-Oriented Software Engineering [Jacobson 1992]*
- *Odell's methodology [Odell 1992]*
- *Shaler and Mellor methodology [Shaler 1992]*

Especially, concepts from the first three methodologies have been heavily drawn upon. UML was adopted by Object Management Group (OMG) as a de facto standard in 1997. OMG is an association of industries which tries to facilitate early formation of standards.

11.2 Three Ways to Apply UML

In three ways people apply UML are introduced:

- **UML as sketch** Informal and incomplete diagrams (often hand sketched on whiteboards) created to explore difficult parts of the problem or solution space, exploiting the power of visual languages.

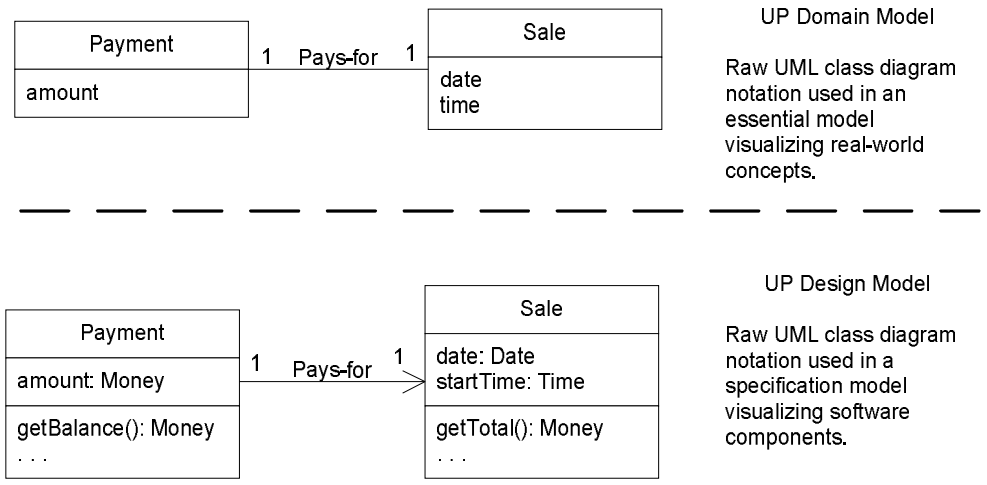
- **UML as blueprint** Relatively detailed design diagrams used either for 1) reverse engineering to visualize and better understanding existing code in UML diagrams, or for 2) code generation (forward engineering).
    - If reverse engineering, a UML tool reads the source or binaries and generates (typically) UML package, class, and sequence diagrams. These "blueprints" can help the reader understand the bigpicture elements, structure, and collaborations.
    - Before programming, some detailed diagrams can provide guidance for code generation (e.g., in Java), either manually or automatically with a tool. It's common that the diagrams are used for some code, and other code is filled in by a developer while coding (perhaps also applying UML sketching).
- **UML as programming language** Complete executable specification of a software system in UML. Executable code will be automatically generated, but is not normally seen or modified by developers; one works only in the UML "programming language." This use of UML requires a practical way to diagram all behavior or logic (probably using interaction or state diagrams), and is still under development in terms of theory, tool robustness and usability.

Agile modeling emphasizes *UML as sketch*; this is a common way to apply the UML, often with a high return on the investment of time (which is typically short). UML tools can be useful, but I encourage people to also consider an agile modeling approach to applying UML.

11.3 Different Perspectives to Apply UML

The UML describes raw diagram types, such as class diagrams and sequence diagrams. It does not superimpose a modeling perspective on these. For example, the same UML class diagram notation can be used to draw pictures of concepts in the real world or software classes in Java.
This insight was emphasized in the Syntropy object-oriented method . That is, the same notation may be used for three perspectives and types of models

- **Conceptual perspective** the diagrams are interpreted as describing things in a situation of the real world or domain of interest.
- **Specification (software) perspective** the diagrams (using the same notation as in the conceptual perspective) describe software abstractions or components with specifications and interfaces, but no commitment to a particular implementation (for example, not specifically a class in C# or Java).
- **Implementation (software) perspective** the diagrams describe software implementations in a particular technology (such as Java).



We've already seen an example of this in Figure where the same UML class diagram notation is used to visualize a domain model and a design model

Fig: Different perspectives with UML.

.

The Meaning of "Class" in Different Perspectives

In the raw UML, the rectangular boxes are called classes, but this term encompasses a variety of phenomena physical things, abstract concepts, software things, events, and so forth. A UML class is a special case of the general UML model element classifier something with structural features and/or behavior, including classes, actors, interfaces, and use cases.

In the UP, when the UML boxes are drawn in the Domain Model, they are called domain concepts or conceptual classes; the **Domain Model** shows a conceptual perspective. In the UP, when UML boxes are drawn in the **Design Model**, they are called design classes; the Design Model shows a specification or implementation perspective, as desired by the modeler.

- **Conceptual class** real-world concept or thing. A conceptual or essential perspective. The UP Domain Model contains conceptual classes.
- **Software class** a class representing a specification or implementation perspective of a software component, regardless of the process or method.
- **Implementation class** a class implemented in a specific OO language such as Java.

UML diagrams :

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following five views of a system:

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

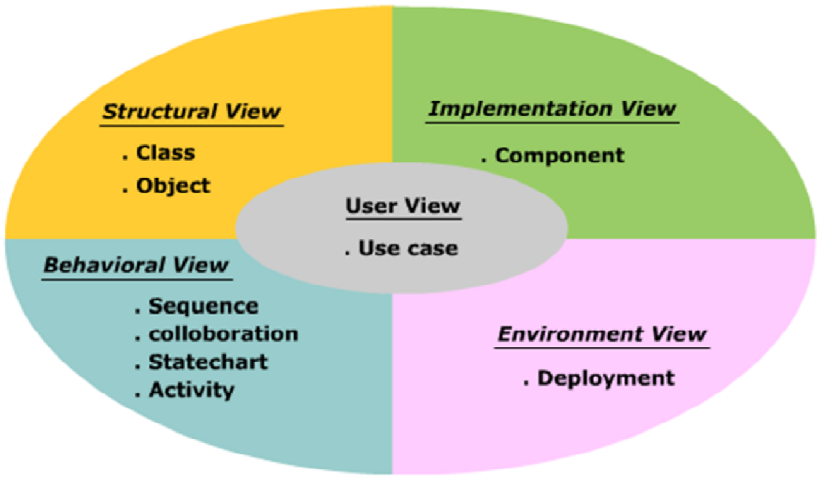


Fig: Different types of diagrams and views supported in UML

User's view: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system**.**
Structural view: The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

Behavioral view: The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

Implementation view: This view captures the important components of the system and their dependencies.

Environmental view: This view models how the different components are implemented on different pieces of hardware.

BASIC BUILDING BLOCKS OF UML

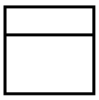
The basic Building Blocks of UML are:

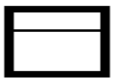
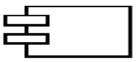
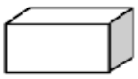
- Things
- Relationships

- Diagrams

UML Things

Things are used to describe different parts of a system; existing types of things in UML are presented in table

Types of Things			
Name	**Symbol**	**Description**	**Variations/other related elements**
Class		Description of a set of objects that share the same: attributes, operations, relationships and semantics.	- actors - signals - utilities

Interface		A collection of operations that specify a service of a class or component.	
Collaboration		An interaction and a society or roles and other elements that work together to provide some cooperative behavior that is bigger than the sum of all the elements. Represent implementation of patterns that make up the system.	
Actor		The outside entity that communicates with a system, typically a person playing a role or an external device	
Use Case		A description of set of sequence of actions that a system perform that produces an observable result of value to a particular actor. Used to structure behavioral things in the model.	
Active class		A class whose objects own a process or execution thread and therefore can initiate a control activity on their own.	- processes - threads
Component		A component is a physical and replacable part that conforms to and provides the realisation of a set of interfaces.	
Node		A physical resource that exists in run time and represents a computational resource.	
Interaction		Set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose.	- messages - action sequences - links
State machine		A behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events.	- states - transitions - events - activities

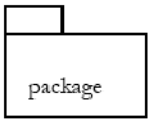
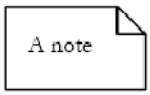
Packages		General purpose mechanism of organizing elements into groups.	- frameworks - models - subsystems
Note		A symbol for rendering notes and constraints attached to an element or a collection of elements.	

Table 1, UML things

Relationships:

The types of UML relationships are shown in the table 2, relationships are used to connect things into well defined models (UML diagrams).

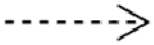
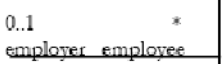
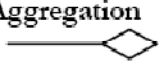
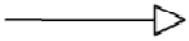
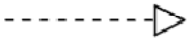
Types of *Relationships*			
Name	**Symbol**	**Description**	**Specialization**
Dependency		A semantic relationship between two things in which a change to one thing may affect the semantics of the dependent thing.	
Association		Structural relationship that describes a set of links, where a link is a connection between objects. Aggregation and composition are "has-a" relationship. Aggregation (white diamond) is an association indicating that one object is temporarily subordinate or the other, while the composition (black diamond) indicates that an object is a subordinate of another through its lifetime.	**Aggregation**  **Composition** 
Generalization		Specialization/generalization relationship in which objects of the specialized element are substitutable for objects of the generalized element.	
Realization		Semantic relationship between two classifiers, where one or them specifies a contract and the other guaranties to carry out the contract. They are used between:	
		- interfaces and classes or components - use cases and collaborations that realize them	

Table 2, UML relations

Different Types of UML Diagrams are:

- **Use case diagrams;** shows a set of use cases, and how actors can use them
- **Class diagrams;** describes the structure of the system, divided in classes with different connections and relationships
- **Sequence diagrams;** shows the interaction between a set of objects, through the messages that may be dispatched between them
- **State chart diagrams; state machines**, consisting of states, transitions, events and activities
- **Activity diagrams;** shows the flow through a program from an defined start point to an end point
- **Object diagrams;** a set of objects and their relationships, this is a snapshot of instances of the things found in the class diagrams
- **Collaboration diagrams;** collaboration diagram emphasize structural ordering of objects that send and receive messages.
- **Component diagrams;** shows organizations and dependencies among a set of components. These diagrams address static implementation view of the system.
- **Deployment diagrams;** show the configuration of run-time processing nodes and components that live on them.

12.0 OO System Development Life Cycles (SDLC):

12.1 OO Software development

Software Development is the process of analysis, design, implementation, testing & refinement to transform users' need into software solution that satisfies those needs.
Object-oriented approach is :

- more rigorous process to do things right
- more time spent on gathering requirements, developing requirements model & analysis model, then turn into design model

Software development process

- *It is the process to change, refine, transform & add to existing product*
- ***transformation 1(analysis)*** *- translates user's need into system's requirements & responsibilities*
    - *how they use system can give insight into requirements, eg: analyzing incentive payroll - capacity must be included in requirements*
- ***transformation 2 (design)*** *- begins with problem statement, ends with detailed design that can be transformed into operational system*
    - *bulk of development activity, include definition on how to build software, its development, its testing, design description + program+ testing material*
- ***transformation 3 (implementation)*** *- refines detailed design into system deployment that will satisfy users needs*

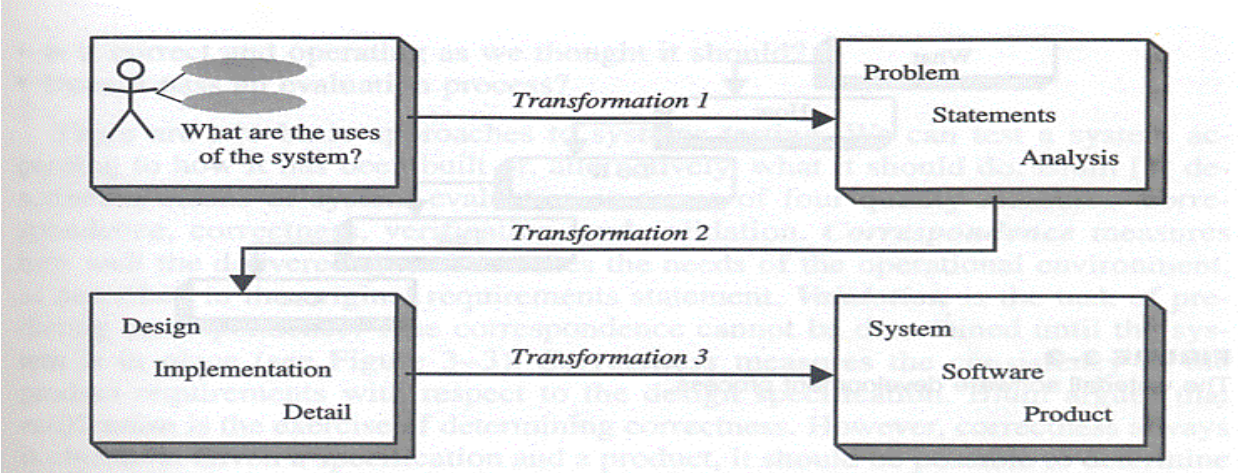


FIGURE 3–1
Software process reflecting transformation from needs to a software product that satisfies those needs.

Fig: Transferring user Needs Software Products

Object-oriented Systems Development Approach

Object-oriented development is highly incremental; in other words, you start with object-oriented analysis, model it, create an object-oriented design, then do some more of each, again and again, gradually refining and completing models of the system:

FIGURE 3–4
The object-oriented systems development approach. Object-oriented analysis corresponds to transformation 1; design to transformation 2, and implementation to transformation 3 of Figure 3–1.

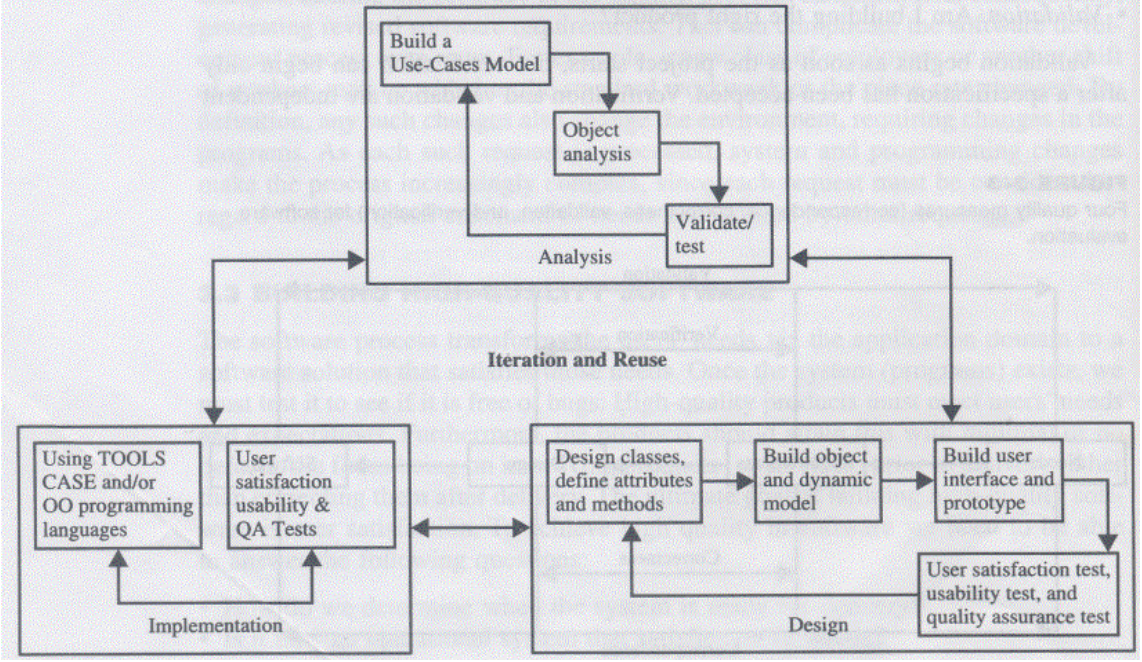


Fig: OO SDLC

Using Jacobson et al. life cycle model – traceable design across development

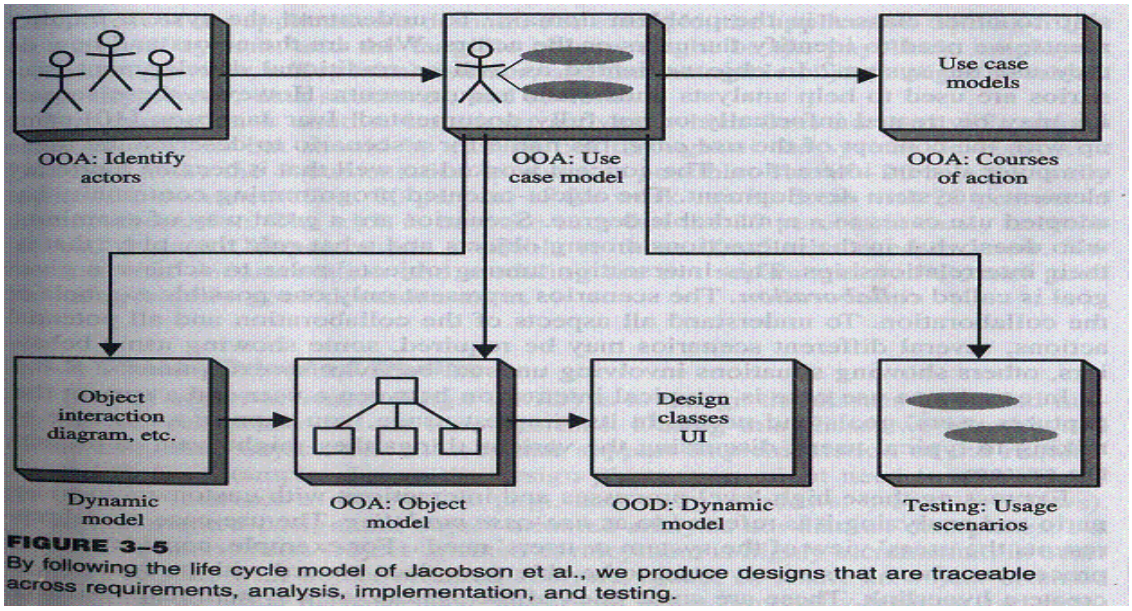


FIGURE 3–5
By following the life cycle model of Jacobson et al., we produce designs that are traceable across requirements, analysis, implementation, and testing.

6.2 Waterfall/Sequential Lifecycle

In a **waterfall** (or sequential) lifecycle process there is an attempt to define (in detail) all or most of the requirements before programming. And often, to create a thorough design (or set of models) before programming. Likewise, an attempt to define a "reliable" plan or schedule near the start not that it will be.

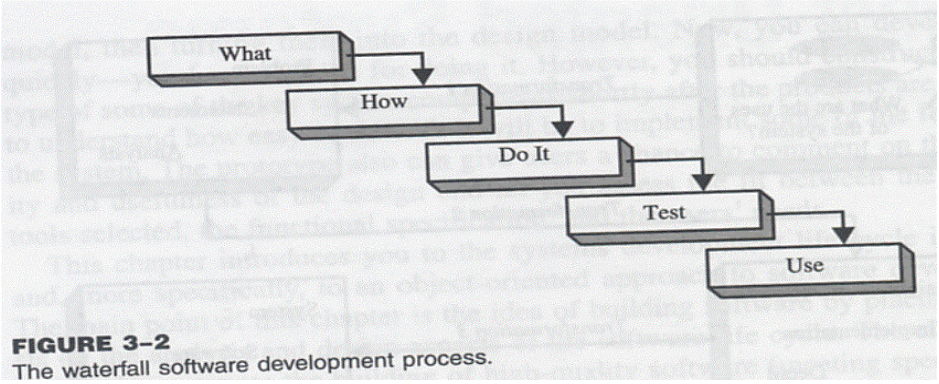
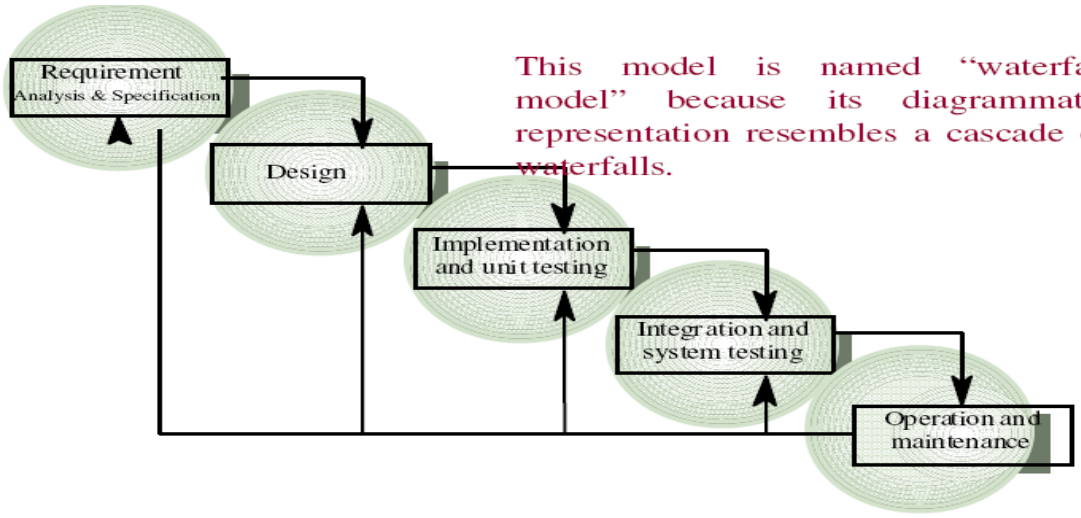


FIGURE 3–2
The waterfall software development process.

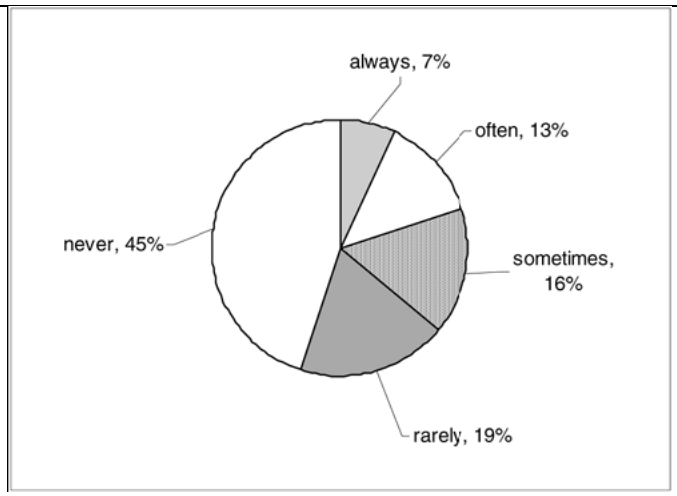


Research now shows conclusively that the 1960s and 1970s-era advice to apply the waterfall was ironically a poor practice for most software projects, rather than a skillful approach. It is strongly associated with high rates of failure, lower productivity, and higher defect rates (than iterative projects). On average, 45% of the features in waterfall requirements are never used, and early waterfall schedules and estimates vary up to 400% from the final actual.

Reason Behind Failure of Waterfall Lifecycle:

- A key false assumption underlying many failed software projects that the specifications are predictable and stable and can be correctly defined at the start, with low change rates. This turns out to be far from accurate and a costly misunderstanding.
- It Assumes requirements remain static over development cycle
- It Assumes requirements are known before design begins
    - sometimes needs experience with product before requirements can be fully understood
- When there is uncertainty regarding what's required or how it can be built

Actual use of waterfall-specified features



The Need for Feedback and Adaptation

In complex, changing systems (such as most software projects) feedback and adaptation are key ingredients for success.

1. Feedback from early development, programmers trying to read specifications, and client demos to refine the requirements.
2. Feedback from tests and developers to refine the design or models.
3. Feedback from the progress of the team tackling early features to refine the schedule and estimates.
4. Feedback from the client and marketplace to re-prioritize the features to tackle in the next iteration.

6.4 Iterative and Evolutionary Development

A key practice in both the UP and most other modern methods is **iterative development**. In this lifecycle approach, development is organized into a series of short, fixed-length (for example, three-week) mini-projects called **iterations**; the outcome of each is a tested, integrated, and executable *partial* system. Each iteration includes its own requirements analysis, design, implementation, and testing activities.

The iterative lifecycle is based on the successive enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation as core drivers to converge upon a suitable system. The system grows incrementally over time, iteration by iteration, and thus this approach is also known as **iterative and incremental development** (see Figure 1). Because feedback and adaptation evolve the specifications and design, it is also known as **iterative and evolutionary development**.

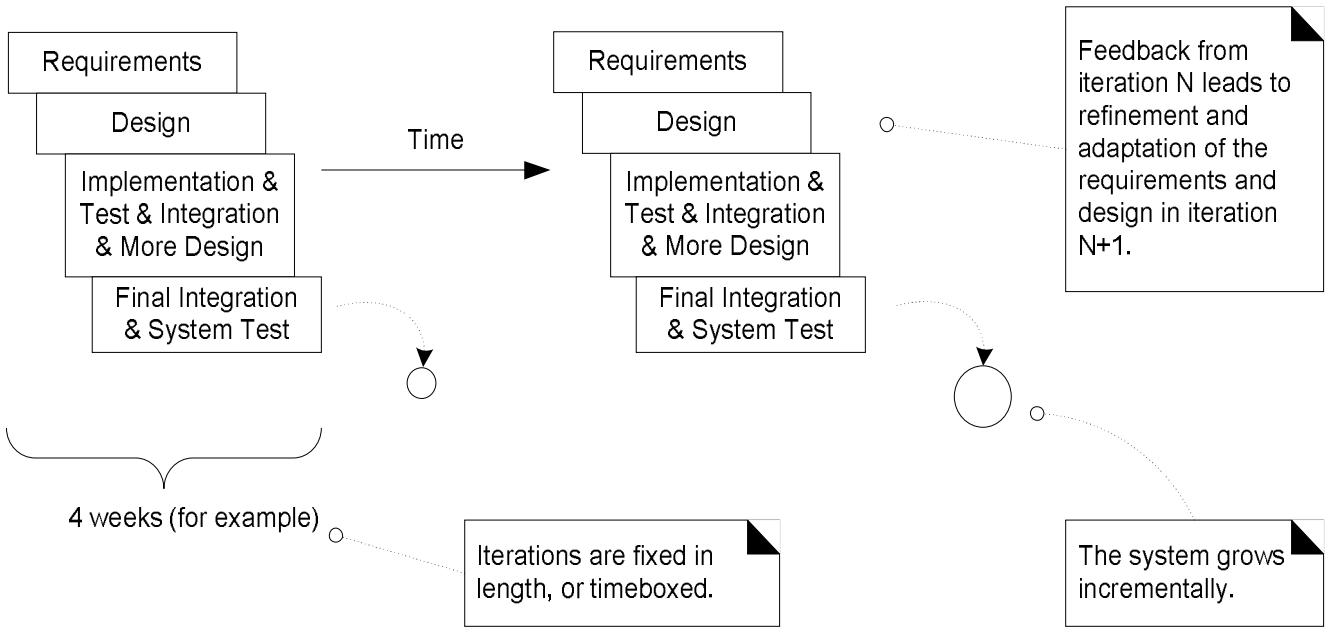


Fig 1: Iterative and evolutionary development.
Early iterative process ideas were known as spiral development and evolutionary development

Benefits of Iterative Development

1. Less project failure, better productivity, and lower defect rates; shown by research into iterative and evolutionary methods
2. Early rather than late mitigation of high risks (technical, requirements, objectives, usability, and so forth)
3. Early visible progress
4. Early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders
5. Managed complexity; the team is not overwhelmed by "analysis paralysis" or very long and complex steps
6. The learning within an iteration can be methodically used to improve the development process itself, iteration by iteration

6.5.1 Unified Process (UP)

A **software development process** describes an approach to building, deploying, and possibly maintaining software. The **Unified Process** has emerged as a popular iterative, incremental, architecture-centric, use-case driven software development process for building object-oriented systems.
In particular, the **Rational Unified Process** or **RUP,** a detailed refinement of the Unified Process, has been widely adopted. Unified Process (UP) is a relatively popular iterative process for projects using OOA/D, UP is common and promotes widely recognized best practices; it's useful for industry professionals to know it.

- In Unified Process, Process is a set of activities intended to reach a goal.
- In Unified Process , The inputs to the software process are the needs of the business and the output will be the software product.
- The Unified Process is one such lifecycle approach well-suited to the UML.
- Unified Process provides a disciplined approach on how to assign tasks and responsibilities within a software development organization.

The UP is very flexible and open, and encourages including skillful practices from other iterative methods, such as from **Extreme Programming** (**XP**), **Scrum**, and so forth. For example, XP's **test-driven development, refactoring** and **continuous integration** practices can fit within a UP project. So can Scrum's common project room ("war room") and daily Scrum meeting practice. Introducing the UP is not meant to downplay the value of these other methods quite the opposite.

The UP combines commonly accepted best practices, such as an iterative lifecycle and risk-driven development, into a cohesive and well-documented process description.

Three reason behind using UP are:

1. The UP is an *iterative* process.
2. UP practices provide an example *structure* for how to do and thus how to explain OOA/D.

3. The UP is flexible, and can be applied in a lightweight and *agile* approach that includes practices from other agile methods (such as XP or Scrum).

A UP project organizes the work and iterations across four major phases:

1. ***Inception*** *approximate vision, business case, scope, vague estimates.*

- *During the inception phase, we develop business model for the project.*
- *The feasibility study is performed as well as the overall scope and size of the project is determined during the inception phase.*
- *The actors of the system and their interaction with the system are analyzed at a high level.*
- *At the end of the Inception stage, the following objectives are to be achieved:*
    - *Concurrence on the scope of the project and the estimates*
    - *Understanding of the requirements*

2. ***Elaboration*** *refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.*

- *In the elaboration phase, a baseline architecture is established, the project plan is developed and risk assessment is also performed.*
- *The major types of risks are*
    - *Requirements Risks*
    - *Technological Risks*
    - *Skills Risks*
    - *Political Risks*
- *At the end of the elaboration*
    - *The use case model should be complete*
    - *Nonfunctional Requirements should be elaborated*
    - *Software Architecture should be described*
    - *Revised risk list should be present*
    - *A preliminary user manual (optional)*

3. ***Construction*** *iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.*

- *All the components are developed and the components are integrated during the construction phase.*
- *All the features are completely tested during this stage. Resources are managed and operations are controlled to optimize cost, schedule and quality.*
- *The construction phase is incremental and iterative.*
- *Refactoring is done after every iteration.*
- *At the end of the construction,*
    - *The product should be stable and mature for release*
    - *Actual versus planned expenditure should be acceptable*

4. ***Transition*** *beta tests, deployment.*

- *The objective of this phase is to transition the software product to the user community.*
- *new releases, correcting defects and optimization are part of this phase.*
- *The activities in this phase include*
  - *User Training*
  - *Conversion of Operational databases*
  - *Roll out the product to marketing and sales*
- *The objectives of the transition phase are*
  - *Customer Satisfaction*
  - *Achieving the concurrence of the stakeholders that the deployment baselines are complete and consistent with the evaluation criteria*
  - *Achieving final product baseline rapidly in a cost effective manner*

Schedule oriented terms in UP

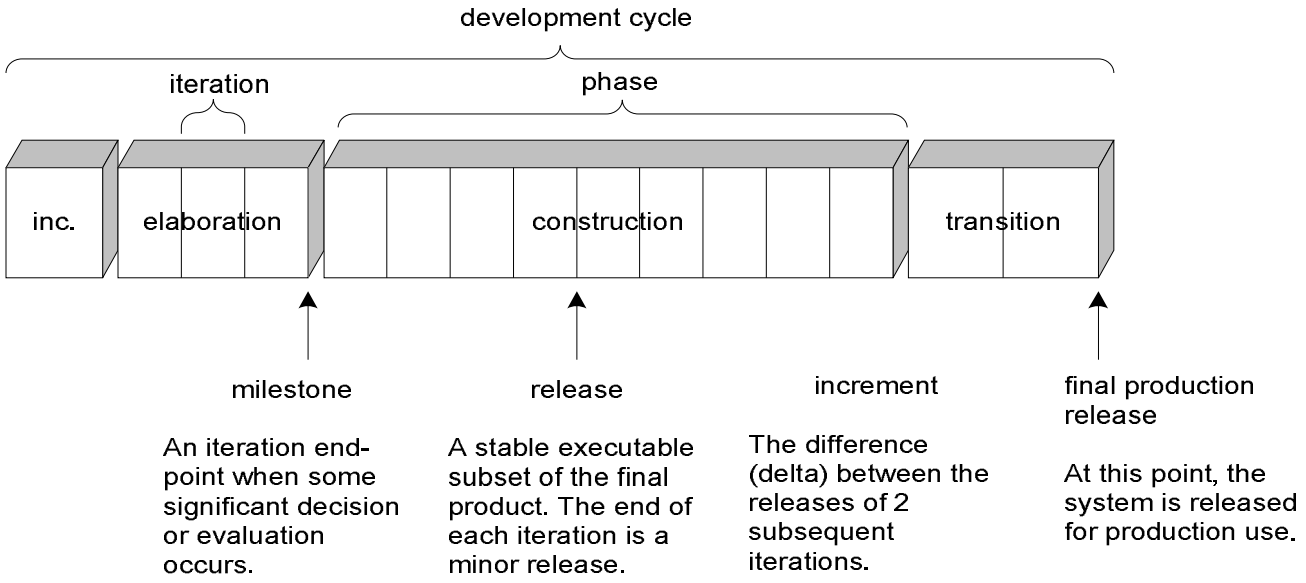
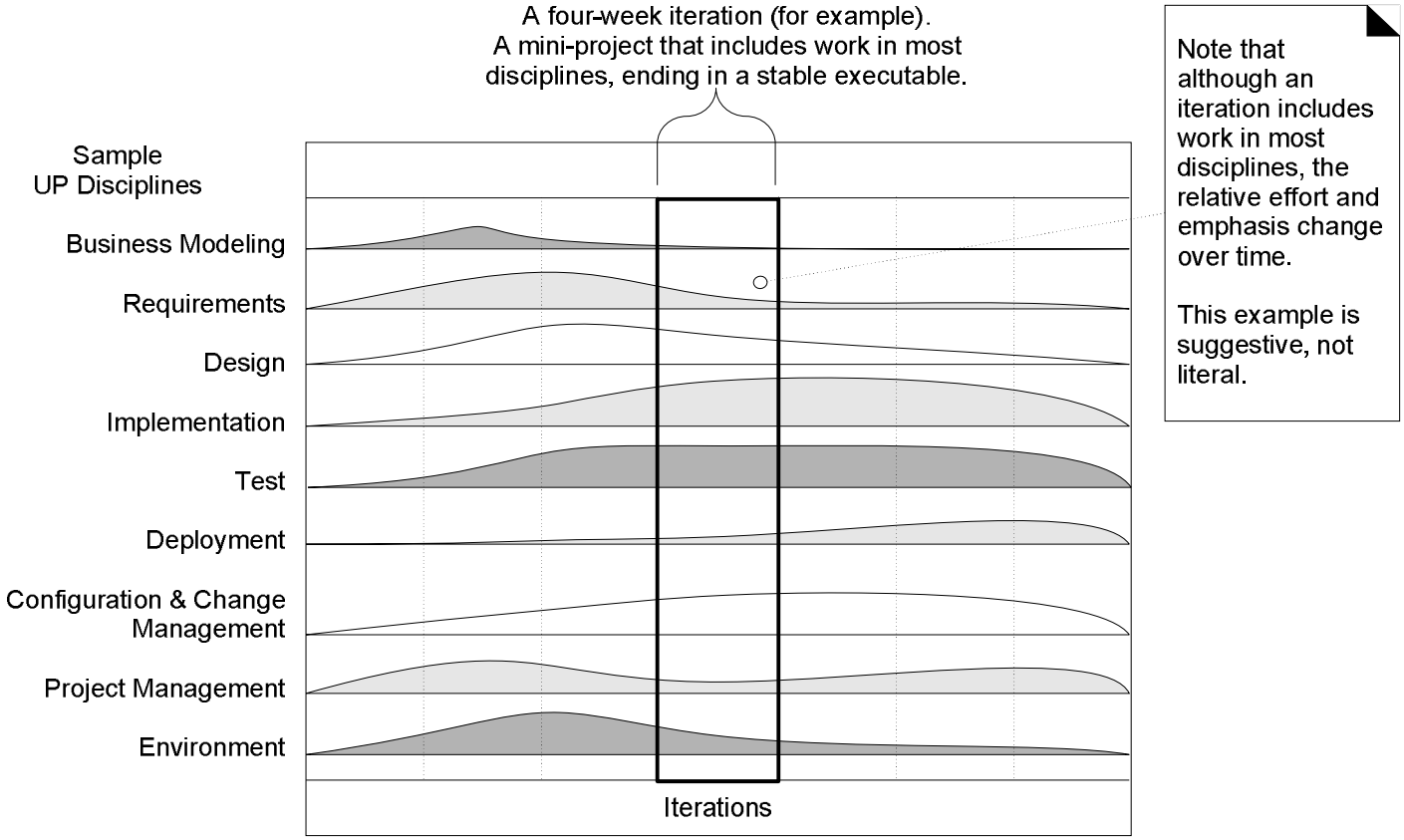


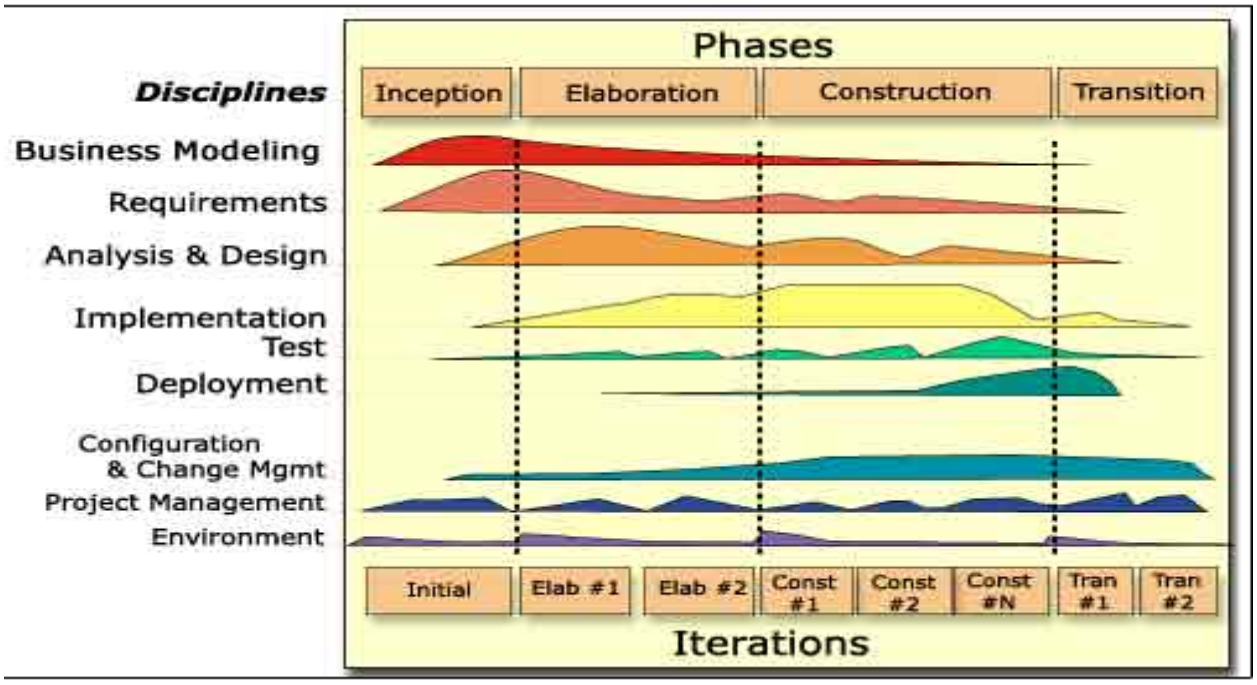
Fig: Schedule-oriented terms in the UP.

UP disciplines

The UP describes work activities, such as writing a use case, within **disciplines** a set of activities (and related artifacts) in one subject area, such as the activities within requirements analysis. In the UP, an **artifact** is the general term for any work product: code, Web graphics, database schema, text documents, diagrams, models, and so on.



Relationship between UP disciplines/workflows and Phases



Risk-Driven and Client-Driven Iterative Planning

The UP (and most new methods) encourage a combination of risk-driven and client-driven iterative planning. This means that the goals of the early iterations are chosen to

1) identify and drive down the highest risks, and
2) build visible features that the client cares most about.
3)

Risk-driven iterative development includes more specifically the practice of architecture-centric iterative development, meaning that early iterations focus on building, testing, and stabilizing the core architecture. Why? Because not having a solid architecture is a common high risk.

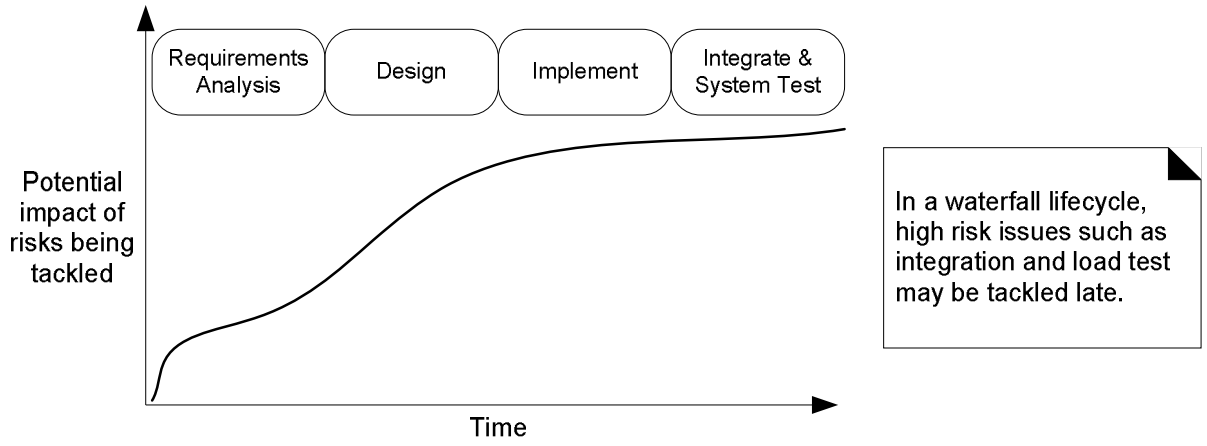


Fig: Potential impact of risk in waterfall lifecycle

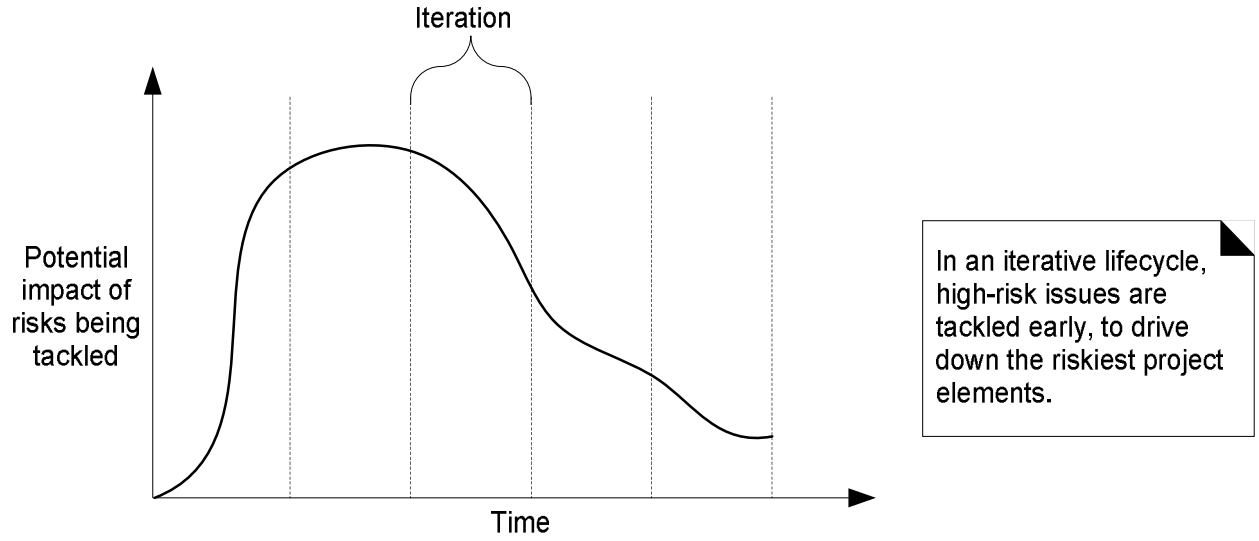


Fig: Potential impact of Iterative lifecycle

6.5.2 Agile development Method

Agile development methods usually apply timeboxed iterative and evolutionary development, employ adaptive planning, promote incremental delivery, and include other values and practices that encourage *agility* rapid and flexible response to change.

It is not possible to exactly define **agile methods**, as specific practices vary widely. However, short timeboxed iterations with evolutionary refinement of plans, requirements, and design is a basic practice the methods share. In addition, they promote practices and principles that reflect an agile sensibility of simplicity, lightness, communication, self-organizing teams, and more.

Example practices from the Scrum agile method include a *common project workroom* and *self-organizing teams* that coordinate through a daily stand-up meeting with four special questions each member answers.

Example practices from the Extreme Programming (XP) method include **programming in pairs** and **test-driven development**.

Existing Agile Development Methods

- Rational Unified Process
- Extreme Programming
- Scrum
- Crystal Family of Methodologies
- Feature Driven Development
- Dynamic Systems Development Methods
- Adaptive software development
- Open source software development
- Agile Modeling
- Pragmatic Programming

The Agile Principles

1. *To satisfy the customer through early and continuous delivery of valuable software*
2. *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage*
3. *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.*
4. *Business people and developers must work together daily throughout the project.*
5. *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*
6. *Working software is the primary measure of progress.*
7. *Agile processes promote sustainable development.*
8. *The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
9. *Continuous attention to technical excellence and good design enhances agility.*
10. *Simplicity the art of maximizing the amount of work not done is essential.*
11. *The best architectures, requirements, and designs emerge from self-organizing teams.*

12. *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

7.0 Requirements Process

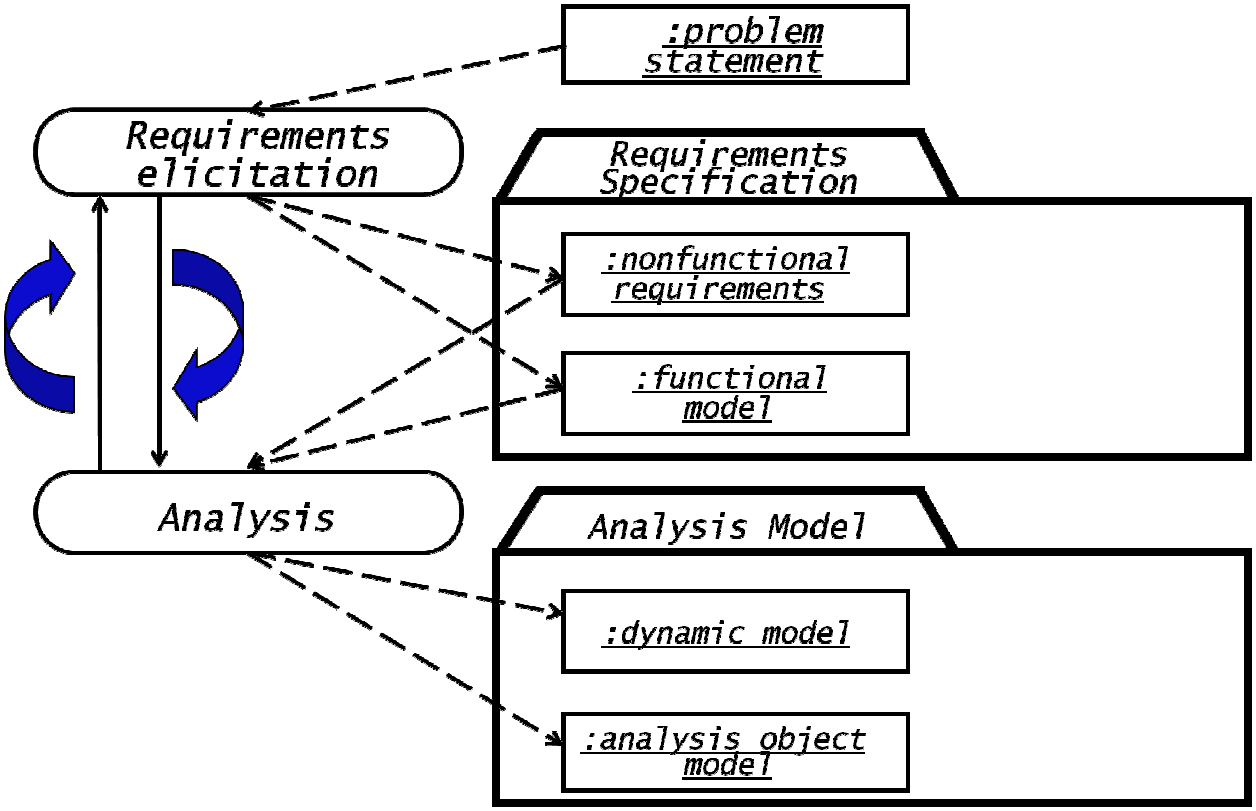


Fig: Requirement Process

Requirements are capabilities and conditions to which the system and more broadly, the project must conform. The UP promotes a set of best practices, one of which is manage requirements. This does not mean the waterfall attitude of attempting to fully define and stabilize the requirements in the first phase of a project before programming, but rather in the context of inevitably changing and unclear stakeholder's wishes, this means " a systematic approach to finding, documenting, organizing, and tracking the changing requirements of a system".

In short, doing it iteratively and skillfully, and not being sloppy.

A prime challenge of requirements analysis is to find, communicate, and remember (that usually means write down) what is really needed, in a form that clearly speaks to the client and development team members.

7.1 Types and Categories of Requirements

In the UP, requirements are categorized according to the FURPS+ model , a useful mnemonic with the following meaning:

- **Functional** features, capabilities, security.
- **Usability** human factors, help, documentation.
- **Reliability** frequency of failure, recoverability, predictability.
- **Performance** response times, throughput, accuracy, availability, resource usage.
- **Supportability** adaptability, maintainability, internationalization, configurability.

The "+" in FURPS+ indicates ancillary and sub-factors, such as:

- **Implementation** resource limitations, languages and tools, hardware, ...
- **Interface** constraints imposed by interfacing with external systems.
- **Operations** system management in its operational setting.
- **Packaging** for example, a physical box.
- **Legal** licensing and so forth.

It is helpful to use FURPS+ categories (or some categorization scheme) as a checklist for requirements coverage, to reduce the risk of not considering some important facet of the system.

Some of these requirements are collectively called the **quality attributes, quality requirements, or the "qualities" of a system**. These include usability, reliability, performance, and supportability. In common usage, requirements are categorized as **functional (behavioral) or non-functional** (everything else); some dislike this broad generalization, but it is very widely used.

As we shall see when exploring architectural analysis, the quality attributes have a strong influence on the architecture of a system. For example, a high-performance, high-reliability requirement will influence the choice of software and hardware components, and their configuration.

7.2 How are Requirements Organized/Represented?

The UP offers several requirements artifacts. As with all UP artifacts, they are optional. Key ones include:

- **Use-Case Model** A set of typical scenarios of using a system. There are primarily for functional (behavioral) requirements.
- **Supplementary Specification** Basically, everything not in the use cases. This artifact is primarily for all non-functional requirements, such as performance or licensing. It is also the place to record functional **features** not expressed (or expressible) as use cases; for example, a report generation.

- **Glossary** In its simplest form, the Glossary defines noteworthy terms. It also encompasses the concept of the **data dictionary**, which records requirements related to data, such as validation rules, acceptable values, and so forth. The Glossary can detail any element: an attribute of an object, a parameter of an operation call, a report layout, and so forth.
- **Vision** Summarizes high-level requirements that are elaborated in the Use-Case Model and Supplementary Specification, and summarizes the business case for the project. A short executive overview document for quickly learning the project's big ideas.
- **Business Rules** Business rules (also called Domain Rules) typically describe requirements or policies that transcend one software project they are required in the domain or business, and many applications may need to conform to them. An excellent example is government tax laws. Domain rule details *may* be recorded in the Supplementary Specification, but because they are usually more enduring and applicable than for one software project, placing them in a central Business Rules artifact (shared by all analysts of the company) makes for better reuse of the analysis effort.

Requirements Elicitation Methods :

Bridging the gap between end user and developer:

- **Questionnaires:** Asking the end user a list of pre-selected questions
- **Task Analysis/observation:** Observing end users in their operational environment
- **Scenarios:** Describe the use of the system as a series of interactions between a concrete end user and the system
- **Case studies**
- **Use cases:** Abstractions that describe a class of scenarios

Requirements Specification vs Analysis Model

Both focus on the requirements from the user's view of the system

- The requirements specification uses natural language (derived from the problem statement)
- The analysis model uses a formal or semi-formal notation (we use UML)

Functional vs. Nonfunctional Requirements

Functional Requirements	Nonfunctional Requirements
• Describe user tasks that the system needs to support • Phrased as actions "Advertise a new league" "Schedule tournament" "Notify an interest group"	• Describe properties of the system or the domain • Phrased as constraints or negative assertions "All user inputs should be acknowledged within 1 second" "A system crash should not result in data loss".
