Object Oriented Software Engineering

Satvik Gupta

May 19, 2023

Contents

Object Oriented Software Engineering

It consists of two terms – object oriented, and software engineering.

Object Oriented

It is a collection of information that itself act as a singular entity. It allows the user to focus completely on the task rather than on the tools.

For example – C++, etc.

With the help of this, reusability as well as abstraction is possible.

The necessity of developing a maintaining a large-size, complex, and varied functionalities software system has caused us to look for new approaches of software design and development.

The conventional approaches like Waterfall Model may not be very useful due to non-availability of iterations, no provision of reuse, and difficulty in incorporating changing requirements.We may also build every software system from scratch that results into a costly software system, including very high maintenance cost.

An object oriented approach may address such issues, that's why it has become very popular in designing,developing, and maintaining large size software systems. Object oriented approach's modelling ability helps us to represent the real world situations and visualize them.

Software Engineering

> It is a profession dedicated to designing, implementing and modifying so that the software is more affordable, maintainable, faster to build, and high quality.

OR

> The establishment and use of some engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines.

Software

It is a combination of programs, documentation and operating manual.

Program

A certain set of instructions that are written for a specific purpose. It may contain statements to enhance the readability of the program.

Documentation

Documentation is created and used during development. It is used to explain the code, what it does, and why it has been coded in a certain way.

Operating Manual

Explains to the customer how the software is to be used. It is delivered along with the software to the customer, at the time of release.

The use of use cases was introduced in Object Oriented Methodology.

Characteristics of Software

Bathtub and software curve bs

Object Oriented Basic Concepts

1. Classes
2. Objects
3. Data Abstraction
4. Encapsulation
5. Inheritance
6. Polymorphism

Classes

> A class represents a template for different objects and describes how these objects are structured internally. Objects of the same class have the same definition, both for the operations, and for the information structures.

OR

> It is a collection of objects and it doesn't take any space in memory. It is also called a blueprint, or a logical entity.

There are two types:

- *Pre-defined*

  Their logic is already written somewhere, and we can use it by importing. For example - Scanner, Console, etc. in Java

- *User-defined*

  The logic for these classes is defined by the programmer.

Objects

Fundamental entities used to model any system. Anything and everything can be an object. It contains data(attributes) and operations(behaviors).

Encapsulation

> The wrapping up of data and functions into a single unit. It is also known as information hiding concept.

Data is hidden from the outside world. The only way to get and modify the data is through operations that are meant to operate on that data. This helps in minimizing impact of changes in the program.

Inheritance

> Deriving a new class from existing class in such a way that the new class can access all the features and properties of the existing class.

The existing class is called parent class, super class, base class. The new class is called child class, subclass, derived class.

Polymorphism

The ability of an instruction,message,etc. to take many forms in an object oriented system is called polymorphism.

Sender of a stimulus (message) doesn't need to know the receiver's class. The receiver can belong to an arbitrary class.

Achieved through function overriding.

For eg - Superclass OutputDevice, with Subclasses Printer and Monitor. Both have a function called ShowData(). Both implement it differently, and a program calling obj.ShowData() doesn't need to know whether obj is a Printer or Monitor. As long as it is an OutputDevice, the program can call the function. The behaviour of the function depends on which subclass is being used.

Data Abstraction

Hiding of complexity of data and operations. Irrelevant details are hidden and important details are amplified to the outside world.

Object Oriented Software Development (OOSD)

The major phases of software development using the object oriented methodology are:

1. **Object Oriented Analysis**

   In this stage, problem is formulated. User Requirements are identified and then a model is built, based upon real world objects.

   The analysis produces models on how the desired system should function and how it must be developed.

   The models do not include any implementation details, so that it can be understood by any non-technical application expert.

2. **Object Oriented Design**

   Object Oriented Design includes two main stages.

   1. *System Design*

      In this stage, the complete architecture of the desired system is designed. The system is conceived as a set of interacting subsystems, that in turn are composed of a hierarchy of interacting objects, grouped into classes.

      System Design is done according to both the system analysis model, and proposed system architecture.

      Here, the emphasis is on the objects comprising the system, rather than the processes in the system.

   2. *Object Design*

      In this phase, a design model is developed based on both the models in the system analysis phase and the architecture designed in the system design phase.

      All the classes required are identified. The designer decides where

      1. The new classes are to be created from scratch.
      2. Any existing classes can be used in their original form,or
      3. New classes should be inherited from the existing classes.
      4. The associations between the identified classes are established and the hierarchy of the classes are identified.

Besides this, the developer designs the internal details of the classes, and their associations, i.e, the data structure for each attribute, and the algorithm for the operations

3. **Object Oriented Implementation + Testing**

   In this stage, the design model developed in the object design is translated into code in an appropriate programming language or software tool. The databases are created and the specific hardware requirements are ascertained. Once the code is in shape, it is tested using different techniques in order to identify and remove errors from the code.

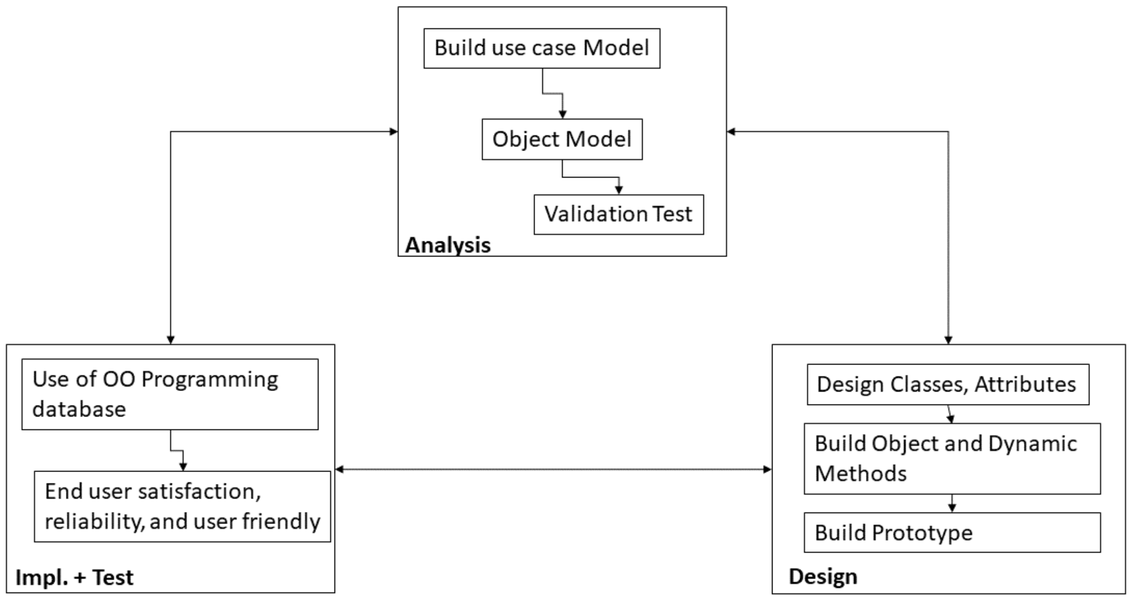

Build use case Model
Object Model
Validation Test
Analysis
Use of OO Programming database
End user satisfaction, reliability, and user friendly
Impl. + Test
Design Classes, Attributes
Build Object and Dynamic Methods
Build Prototype
Design

Figure 1: Object Oriented Software Development

Coad/Yourdon Methodology

Known as Object Oriented Analysis

1. Identify classes and objects (study environment and document behaviours)
2. Identification of Structures (identify is-a and whole-part relationships)
3. Definition of Subjects (each structure is classified into a subject)
4. Definition of Attributes
5. Definition of Services (methods)

Rumbaugh Methodology

Known as Object Modelling Technique (OMT)

1. Analysis Phase

   1. **Object Model -** Static aspects of system

   Classes and inheritance relationships are extracted from problem statement.

   2. **Dynamic Model -** Behavioural aspects of object model and describes state of the system

   Identifies states and events in classes identified by object model.

   3. **Functional Model -** Represents functional aspects of the system

Depicts functionality of the system by creating data flow diagrams.

2. Sys Design - HLD is developed taking implementation env., including DBMS,etc. into account.
3. Object Design - Objects are defined in detail. Algorithms and operations defined.
4. Implementation

Booch Methodology

Object Oriented Design - Combines analysis, design and implementation. Iterative and incremental.

Macro Process

High Level Process

1. **Establish requirements** - Context diagram, prototypes
2. **Analysis Model** - Use case model, identification and prioritization of risks.
3. **Design of Architecture**
4. **Evolution in the form of refinements** - Implementation
5. **Maintenance of delivered functionality** - Post deployment activities

Micro Process

Lower level process.

1. Identification of classes and objects
2. Identification of semantics of classes and objects
3. Identification of relationships btw classes and objects
4. Specification of interfaces and implementation of classes and objects

Jacobson Methodology

OOSE methodology, 5 models:

1. **Requirement model** - Gather s/w requirements. Use cases, actors, etc.
2. **Analysis model** - Create robust and ideal structure of objects. Identify interface objects, DB related objects, control objects, etc.
3. **Design model** - Refine the object w.r.t implementation environment. Objects become *blocks*.
4. **Implementation model** - Implements the objects (blocks) into modules.
5. **Test model** - Validate and verify the functionality of the system

OO Modelling and UML

Object oriented modelling - constructing visual models based on real world objects - Helps in understanding problems and developing documents and producing code. - Well understood requirements, robust designs, etc, etc.

Most popular methodologies - OOD (Booch), OMT (Rumbaugh), OOSE (Jacobson). All were combined into Unified Modelling Language (UML).

- Language for visual modelling
- Allows specifying, visualizing, constructing and understanding various artifacts of the system.
- Models static and dynamic aspects of the system.
  - Static aspects - Objects and their relationships
  - Dynamic - Events, states and object interactions

Class, Responsibility, Collaboration (CRC)

1. Class - Template consisting of attributes and operations
2. Responsibility - Attributes and operations included in a class
3. Collab - Other classes that a class calls to achieve its functionality.

Traditional	OO
The system is viewed as a collection of processes.	The system is viewed as a collection of objects.
Data flow diagrams, ER diagrams, data dictionary and structured charts are used to describe the system.	UML models including use case diagram, class diagram, sequence diagrams, component diagrams, etc. are used to describe the system.
Reusable source code may not be produced.	The aim is to produce reusable source code.
Data flow diagrams depicts the processes and attributes.	Classes are used to describe attributes and functions that operate on these attributes.
It follows a top-down approach for modelling the system.	It follows a bottom-up approach for modelling the system.
It is non-iterative.	It is highly iterative.

Process Framework

Software Process Framework is a foundation of complete software engineering process. It includes all the umbrella activities.

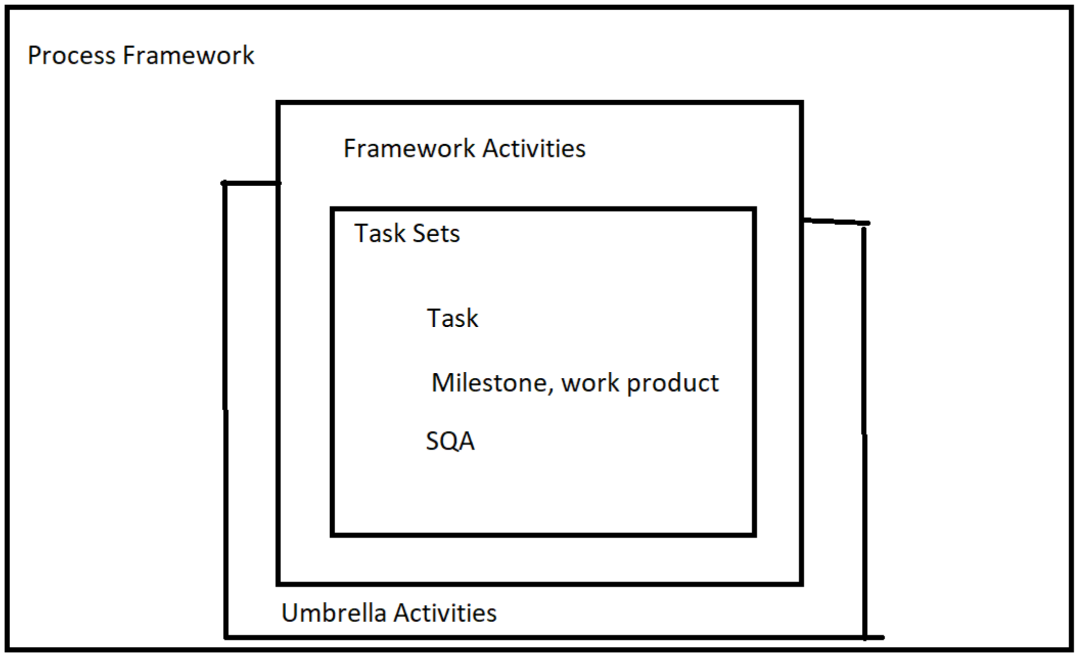


Figure 2: Process Framework

A generic process framework consists of 5 activities:

1. Communication

   Requirement Gathering, extensive communication with customer

2. Planning

   We discuss the technical related tasks, work schedule, risks, and required resources

3. Modelling

   It is about building representations of things in the real world.

   In modelling, a product's model is created in order to better understand requirements

4. Construction

   In SE, construction is the application of set of procedures that are needed to assemble the product. In this activity, we generate the code and test the product in order to maintain better product.

5. Deployment

   In this activity, a complete or a non-complete product or software, are presented to the customers to evaluate, and give feedback.

   On the basis of their feedback, we modify the products to supply a better product.

Umbrella Activities

Umbrella Activities are a set of steps or procedures that the SE team follows to maintain the progress, quality, change and risk of the overall development task.

SE is a collection of 4 related steps. These steps are presented or accessed in different approaches, in different software process models.

These steps of umbrella activities will evolve through the phases of the generic view of SE.

1. **Software Project Tracking and Control**

   Before the actual development begins, a schedule for development of the software is created. Based on that schedule, the development will be done.

   However, after certain period of time, it is required to review the progress of the development and to find out the actions which are in need to be taken to complete the development,testing etc.

   The outcome of the review may require the software development to be rescheduled.

2. **FTR (Formal Technical Review)**

   SE is done in clusters or modules. After completing each module, it is good practice to review the completed module and find out and remove errors so that the next module can be prevented.

3. **SQA**

   The quality of software, such as UX, performance, load handling capacity, etc. should be tested, and make sure it matches predetermined milestones.

   This reduces the task at the end of the development process. It should be conducted by dedicated teams so that the development can keep going on.

4. **SCM (Software Config Mgmt)**

   It's a set of activities designed to control change by identifying the work products that are likely to change and establish relationships among them.

   Defining mechanisms for managing different versions of these work products.

5. **Document Preparation and Production**

   All the project planning, and other activities, should be documented properly.

6. **Reusability Management**

   This includes the packing up of each part of the software project. They can be connected, or any kind of support can be given to them, later to update or upgrade the software at user demand or time demand.

7. **Measurement and Metrics**

   This will include all the measurement of every aspect of the software project.

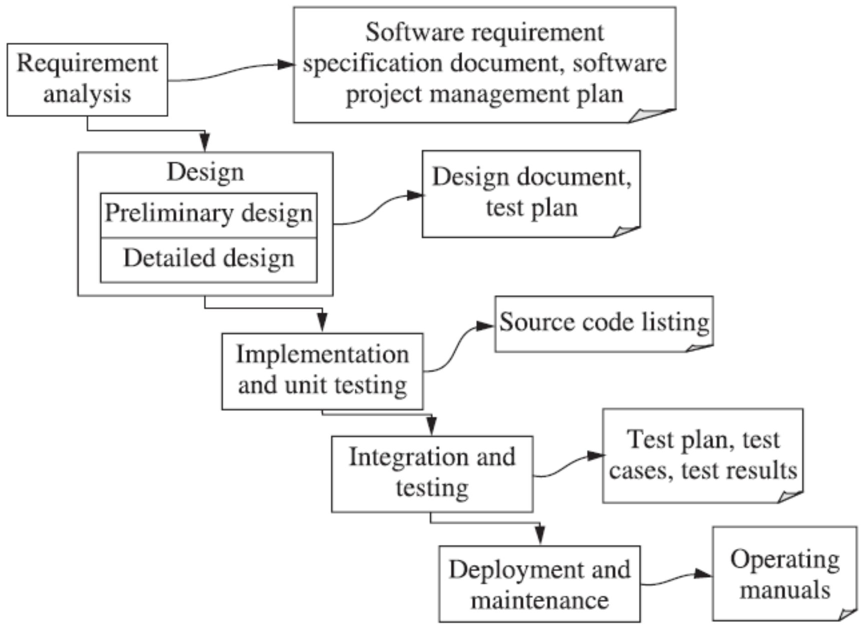
8. **Risk Management**

   It is a series of steps that helps a software team to manage and understand uncertainty. It's a really good idea to identify, assess, estimate its impact, estimate probability of threats, and establish a plan for what to do in case the problem actually occurs.

Often combined in Object Oriented Analysis.

SDLCs

Waterfall


Requirement analysis
Software requirement specification document, software project management plan
Design
Preliminary design
Detailed design
Design document, test plan
Implementation and unit testing
Source code listing
Integration and testing
Test plan, test cases, test results
Deployment and maintenance
Operating manuals

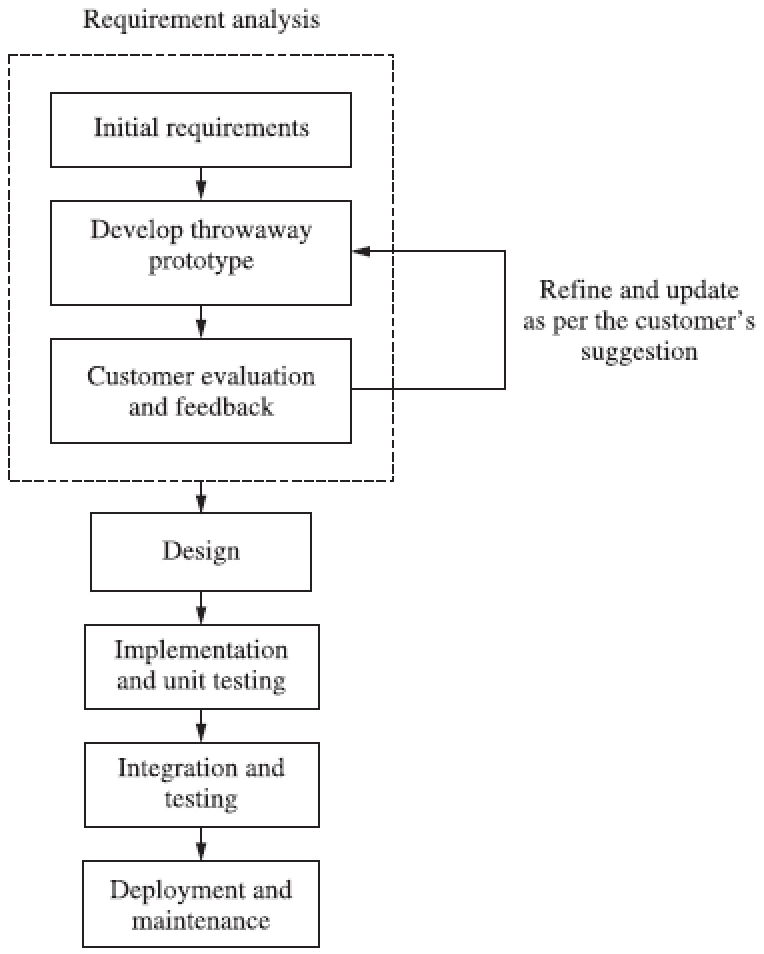
Advantages

- Easy to understand
- Simple to implement
- Distinct phases

Disadvantages

- Large no of documents
- Requirements freezed at start
- Working product delivered late
- Slow, may take years
- Testing is difficult
- Real projects rarely sequential

Prototyping


Requirement analysis
Initial requirements
Develop throwaway prototype
Customer evaluation and feedback
Refine and update as per the customer's suggestion
Design
Implementation and unit testing
Integration and testing
Deployment and maintenance

Advantages

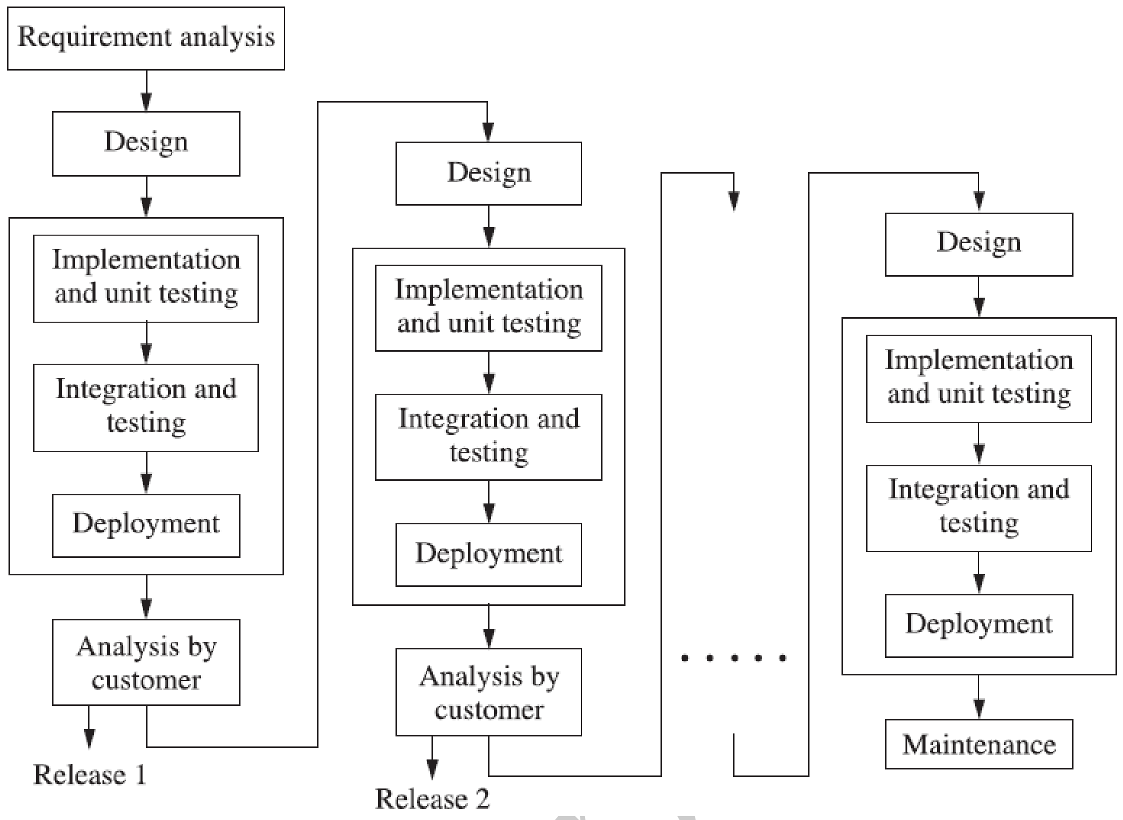
- Stable requirements
- High quality system
- Low cost

Disadvantages

- Slower delivery

Iterative Enhancement

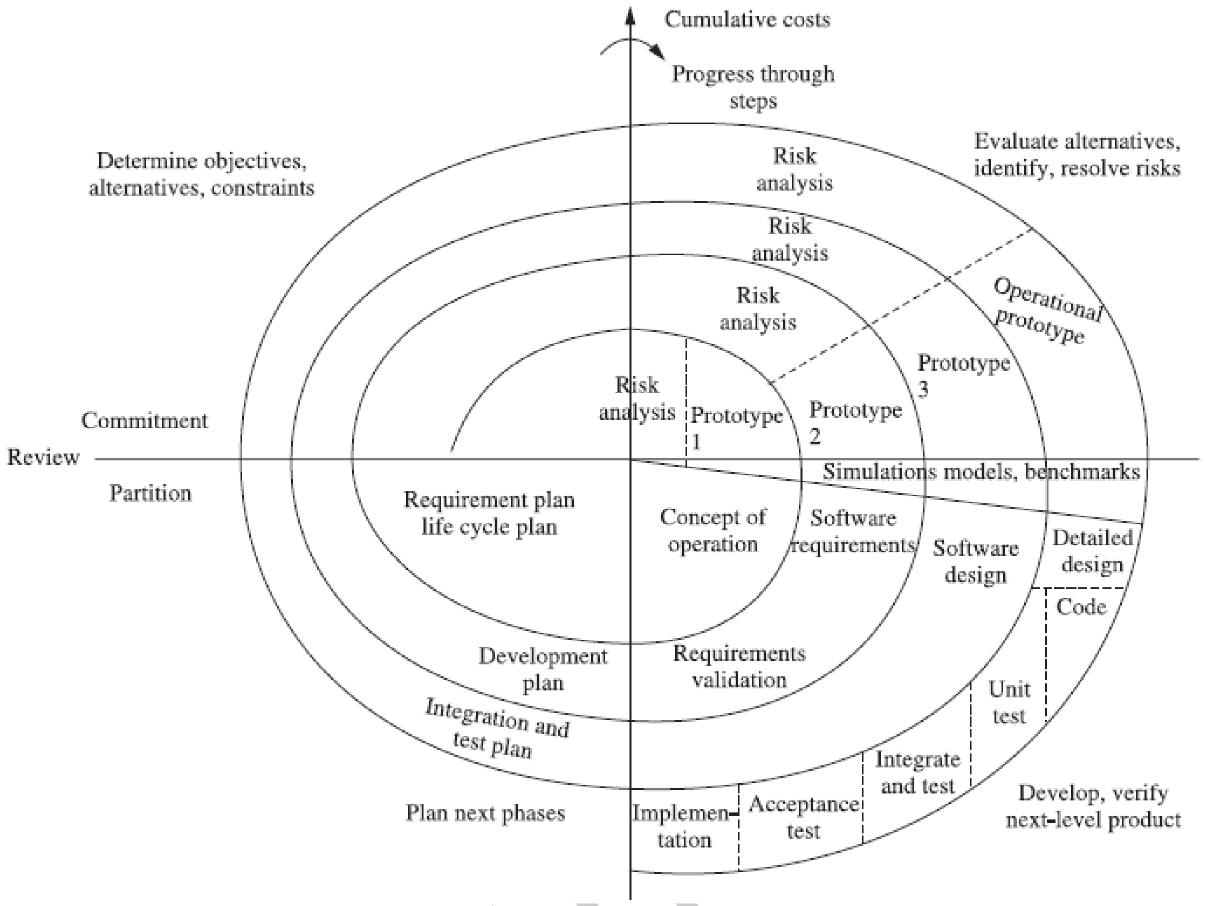
Waterfall stages in many cycles



- Partial product delivered every cycle
- Complete product delivered after several cycles

Spiral Model

Risk-based.



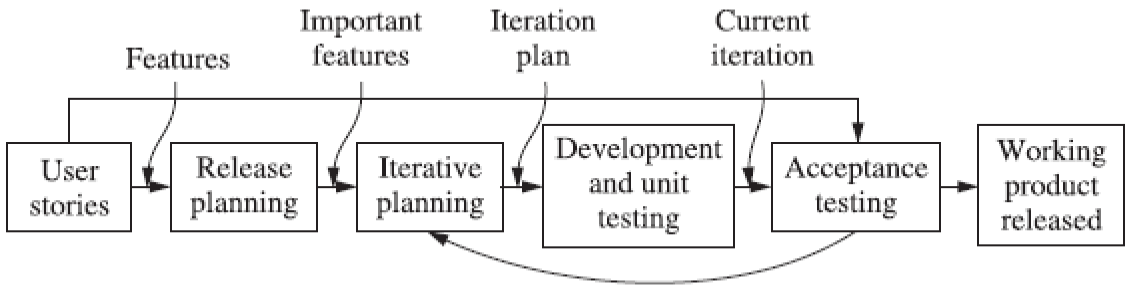
Rounds

1. Round 0 - Feasibility study
2. Round 1 - Concept of operation
3. Round 2 - Top level requirement analysis
4. Round 3 - Software design
5. Round 4 - Design, implementation and testing

XP - Extreme Programming

Agile methodology:

1. Team cohesiveness
2. Customer is part of the team
3. Requirement changes are accepted
4. Working software produced quickly
5. Progress is measured by working software and not documents
6. Iterative planning instead of iterative development. Plans are changed based on learnings.
7. Distributed leadership



1. **User Stories -** only contain estimate of time taken for the feature. Requirement details taken from customer at development time.
2. **Release planning -**
   - Developers estimate story time, and customer selects the order of story development.
   - Large stories may be divided into substories.
   - Developer may do exploration (spike) of story
3. **Iteration Planning -** Stories divided into tasks that are handed to developers. Working product released after each iteration.
4. **Dev and Unit tests -**
   - Important tasks chosen by customers and implemented.
   - Pair Programming
   - Refactoring
   - Automated unit tests
5. **Acceptance Testing -** Automated black box acceptance tests are created from user stories. Customer runs and verifies them.
6. **Working product Released.**

Object Oriented SDLCs

Difference btw Conventional and OOP SDLCs

	Conventional	OO
Methodology	Functional, process driven	Object Driven
Requirement	DFD, ER, Data dictionary	Use-case approach
Analysis	DFD, ER, Data dictionary	Object identification and description, attribute and function determination
Design	Structure chart, flowchart, pseudocode	Class Diagram, Sequence Diagram, Object Diagram, UML
Implementation & Test	Implement process, functions	Implement objects and interactions among objects.
Documentation	Many documents at the end of each stage	Document may or may not be produced at the end of each stage

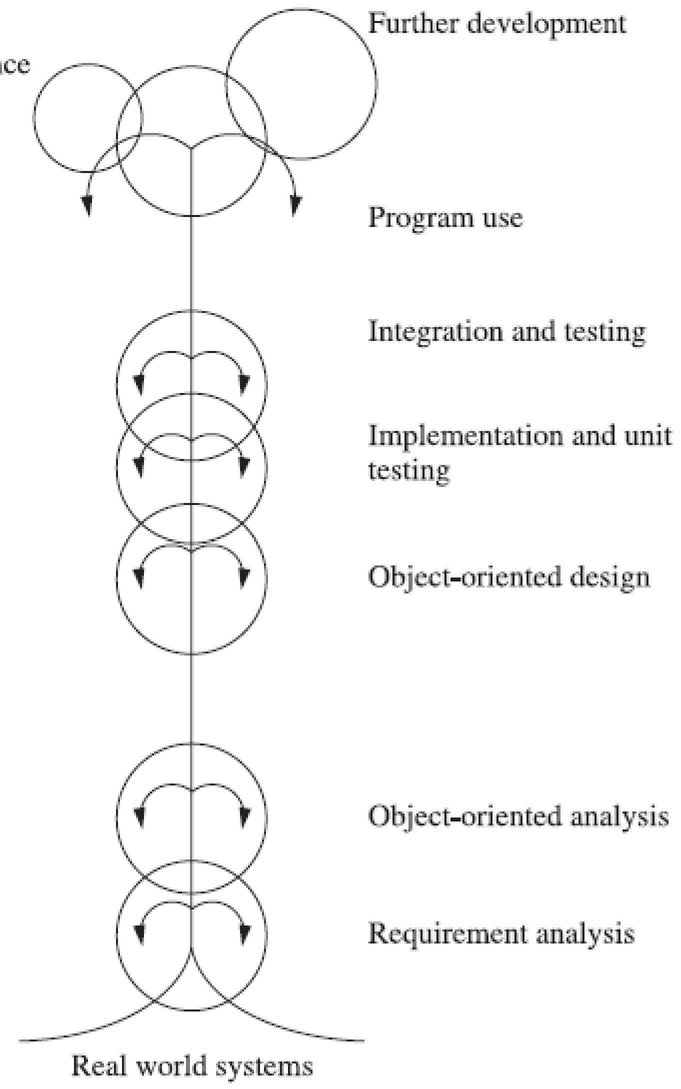
Phases of OOSDLC

1. Object Oriented Requirement Analysis
2. Object Oriented Analysis
3. Object Oriented Design
4. Object Oriented Programming and testing

Fountain Model

- Reusability of source code
- Like a fountain with ideas and new features flowing from top to bottom
- Arrows represent iterations

- Circles represent overlapping phases.



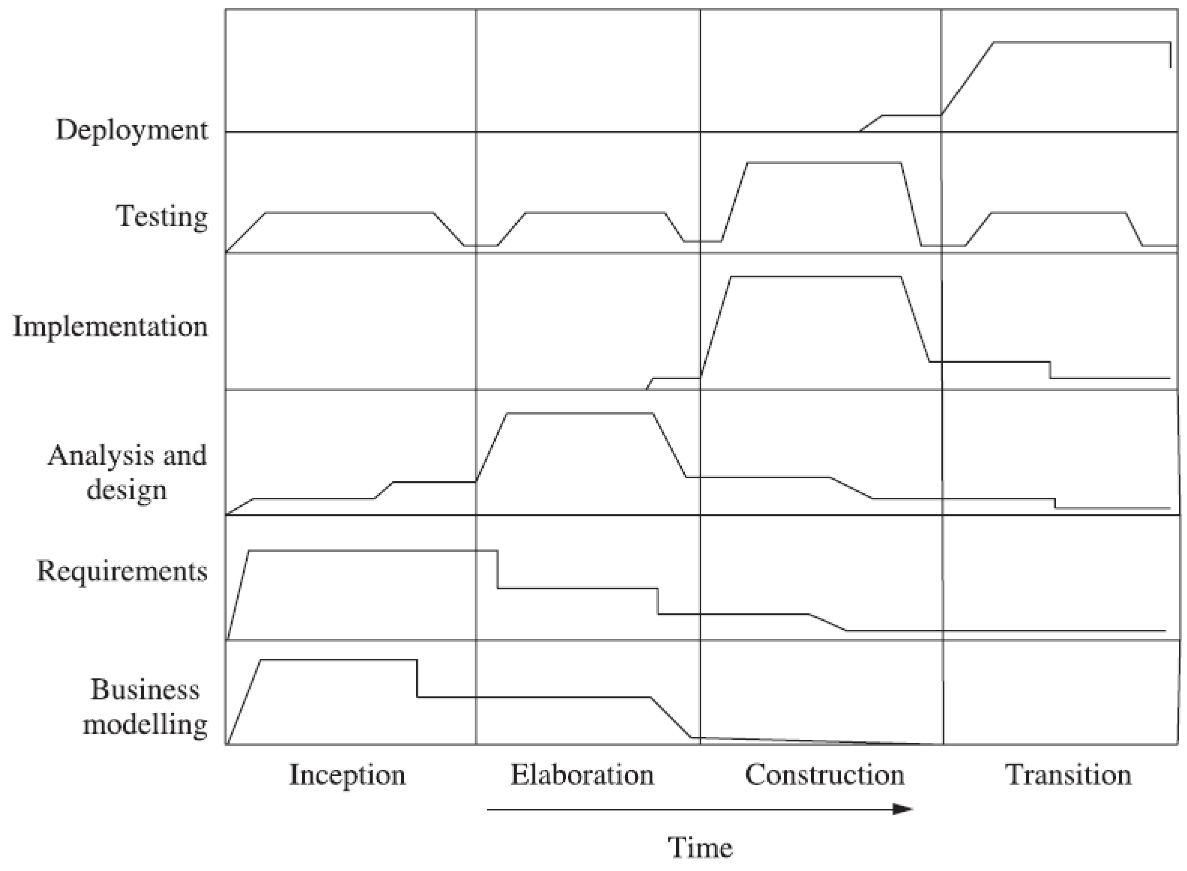
Rational Unified Process

- Adaptable Process Framework
- Iterative
- UML

Features

1. **Iterative Dev -** Series of iterations, feedback after each. Helps monitoring schedule and budget.
2. **EFfective req. elicitation -** Use case approach.
3. **Visual Modelling -** Build (abstracted) models that portray different views of the system. Use UML.
4. **Reusable Components -** Develop and use reusable components (independent subsystem that fulfills a clear goal).
5. **Ensure quality -** Continuously assess quality. It becomes harder to maintain quality in later stages of development.
6. **Change control and management -** Manage and track changes
7. **Automated Testing -** Functional as well as non-functional automated testing.

Structure of RUP



Static Structure

Describes the process in terms of roles, activities, artifacts,disciplines and workflows.

Who(roles) does **what** (artifacts), **when** (workflows), and **how**(activities).

Roles perform activities to produce artifacts.

- **Roles** describe the position or function of a particular person. One person may have multiple roles.
- **Activities** describe the tasks/work performed by a person in a specific role.
- **Artifacts** are outputs produced during the development, design, etc. phases.They may be final products or inputs to the next phases.
- Roles are associated with activities.
- Activities are associated with artifacts.
- **Workflows** consist of a series of activities to produce a particular output.
- **Disciplines** are used to organize a set of activities. RUP consists of 6 major disciplines.
  1. Business Modelling
  2. Requirement
  3. Analysis & Design
  4. Implementation
  5. Testing
  6. Deployment

Roles - Manager, Analyst, Tester, Developer, Designer.

Activities - Review Requirement, Generate use case, Define class, Prepare test plan.

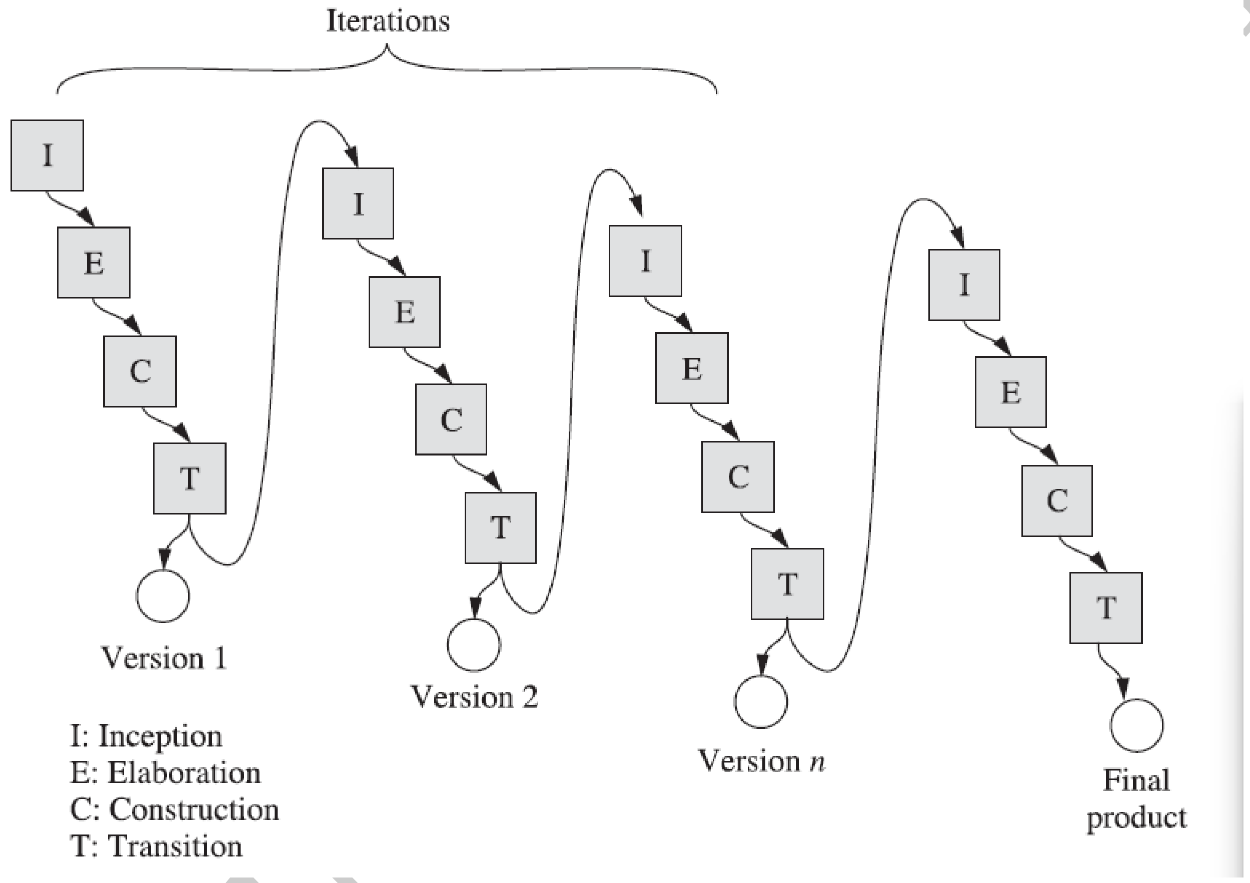
Artifacts - SRS, Use case model, Class model, Design document, Source code, Test plan, user manual.

Dynamic Structure

Organized along time. It has 4 phases.

1. **Inception**
2. **Elaboration**
3. **Construction**
4. **Transition**

These 4 phases run iteratively. Each iteration produces a new version of the software.



Inception

- Initial Stage, non-iterative.
- How feasible is the project, what are the risks, what are the high level requirements, how long will it take?

Essential Activities:

- Scope and boundary of project
- Cost And schedule
- Iteration Plan
- High level risks
- Significant use cases and actors.

Artifacts produced:

- Vision Document
- Business Model
- Iteration Plan
- Initial Use case
- Prototype
- Project Glossary

- Risk Assessment
- Software Development Plan
- Software Tools

Elaboration

- Most critical phase.
- Planning and architectural design
- Elaboration is done for each use case in the current iteration.

Essential Activities:

- Establishment and validation of architectural baselines
- Design use case model
- Select components and create policies for their purchase and usage
- Address significant risks
- Detailed iteration plan
- Prototypes

Artifacts produced:

- Updated risk list
- Use case model
- Detailed iteration plan
- Software architecture description document
- Design and data model
- Implementation model
- Development case
- Test plan
- Test automation architecture

Construction

- Product constructed on the basis of architecture and design of elaboration phase.
- Testing also done
- Remaining requirements determined
- Deployable product constructed.

Essential Activities:

- Optimize work by avoiding rework and unnecessary coding
- Assess and verify quality
- Test all functionality of the system (unit, system and integration test)

Artifacts produced:-

- Software Product
- Test suite
- Test plan
- Documentation manual
- Deployment plan
- Design model
- Implementation model
- Training material
- Iteration plan (for transition phase)

Transition

- Usable product of sufficient quality has been produced.
- Product handed over to customer.
- Delivering, training users and maintaining software.
- Beta releases, bug fixes, enhancement releases.

Artifacts produced:

- Product release
- Beta release report
- Release notes
- User Manual
- Training material

Software Requirements

A requirement is defined as a condition or capability to which a system must conform. It describes the "what" of the system, not "how".

Characteristics of a Good Requirement

- Correct
- Unambiguous
- Complete
- Consistent
- Verifiable
- Traceable
- Modifiable
- Clear
- Feasible
- Necessary
- Understandable

Identification of Stakeholders

Every person who is affected by the system (directly or indirectly) is a stakeholder.

Internal People of Customer's Organization

This can be both customers and users. Customers are those who ask for the software to be developed, request changes, approve the software, and also pay for the system. Users are those who use the system after it has been deployed. Customer's preferences are given more importance compared to user.

External People of Customer's Organization

Includes consultants, domain experts, maintenance people, etc.

Internal People of Developer's Organization

Everyone involved in the development of the system - developers, programmers, testers, project managers, graphic designers, etc.

External People of Developer's Organization

Any external people involved in the development of the software. Domain experts, consultants, third party testers, etc.

Functional vs Non-Functional Requirements

Functional

- They describe what the software will do.
- They're also called product features.
- They tell what the customer expects from the system.
- Sometimes they specify what the software should not do.

Non-Functional

- They tell about the quality of the software.
- Describe how well the software does what it's supposed to do.
- Include stuff like reliability, usability, maintainability, availability, etc.
- Also called quality attributes.

Requirement Elicitation Techniques

Techniques to understand what exactly the customer wants, i.e, to translate vague wants of customers into concrete requirements that can be formalized and written down.

Interviews

- Easy and simple technique
- Can be formal or informal (Structured/non-structured)
- Informal has free flow of discussion.
- A questionnaire may be given to the stakeholders before the interview. We can ask any questions in the questionnaire, and get clarifications and remove ambiguities during the interview.

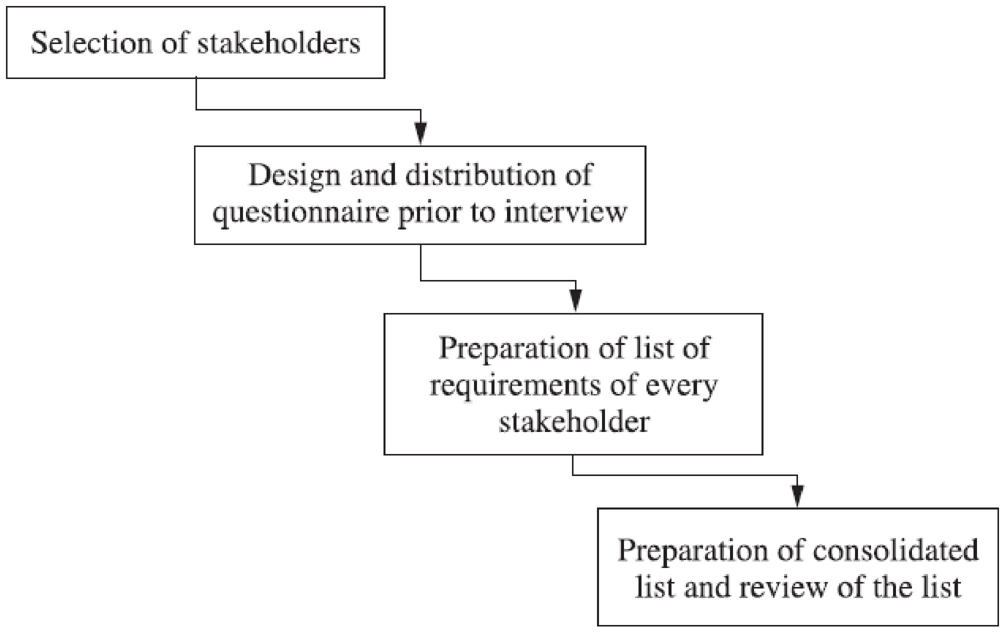

Selection of stakeholders
Design and distribution of questionnaire prior to interview
Preparation of list of requirements of every stakeholder
Preparation of consolidated list and review of the list

Figure 3: Structured Interview

- Based on discussion with each stakeholder, we prepare a list of requirements of each stakeholder
- We then combine all the captured requirements , remove redundancies, inconsistencies and ambiguities.

Brainstorming

- Group discussion
- Various levels of stakeholders may be present together
- Projectors, whiteboards should be there. Free flow is encouraged and criticism is not allowed.
- Creativity is encouraged.
- Facilitator is there who handles conflicts, bias, etc.

- Ideas are written down in simple way so everyone can understand them. Incomplete ideas are also written so they can be discussed later.
- IRD (Initial Requirement Document) is generated from this.

Facilitated Application Specification Technique (FAST)

- Team-oriented approach similar to brainstorming.
- Customers and developers work together to finalize requirements. Facilitator manages everything.
- Facilitator may be customer, developer or outsider.

Guidelines

- Conducted at neutral site.
- Rules should be created and given to everyone in advance.
- Free flow of ideas
- Facilitator gives overview of the project.
- Something like projector, stickers, charts, whiteboard, etc. should be present so everyone can see all the ideas.
- Long debates and criticism should be avoided.

Preparation

Each member needs to create the following things

- Objects
  - that are part of the environment of the system
  - that are produced by the system
  - that are used by the system
- Services that interact with the objects
- Constraints (size, cost)
- Performance criteria (speed, accuracy)

Activities

- Each member presents their lists
- A group is created to create a consolidated list
- The consolidated list is presented and further discussions happen under the facilitator's directions. Changes may be made to the list
- Few small groups may be created to draft mini-specifications
- Each subteam presents the mini-specs to all the FAST attendees. Discussions and changes happen.
- List of issues is prepared
- Validation criteria is decided for each requirement,i.e, how will we check that this requirement has been fulfilled
- Subteam creates the final draft for the specification. Final draft is prepared using inputs of all the meetings and stakeholders.

Prototyping

- Prototyping is rapid development of a system for the purpose of understanding the requirements
- Expensive
- Simplified version (prototype) of the system is created and shown to the customers
- Feedback and views is taken.
- Helps understand requirements
- Prototype should be discarded, but experience and feedback gained while developing it should be used when creating the actual system.
- Should be developed quickly. Internal structure of the prototype is not very important.

Use Case Approach

- Generally for OO systems

- Describes only functional requirements
- Use case describes who does what with the system, for what goal. It may include alternate flows.
- Doesn't consider or care about internal details of the system.

Terms

- **Use case** - structured outline for description of requirement, written in natural language.
- **Use case scenario** - Instance of a use case. Represents a path through a use case. A use case may contain many such paths.
- **Use case diagram** - Graphical Representation of a use case.
- **Actors** - Actors is someone/something that interacts with the system, but lies outside the system.

Relationships between Use Cases

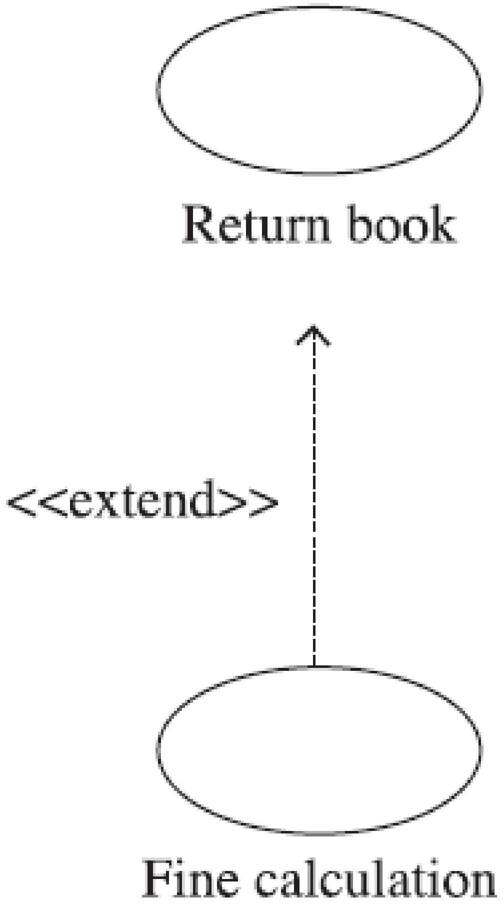
Extend Relationship

- Used to model an alternative/special path of the use case that may not always occur.
- Extends the functionality of the use case.

Used in the following way

- Normal use case occurs, till the special condition occurs for new use case
- New use case is inserted and executed
- After new use case finishes, the normal use case resumes.

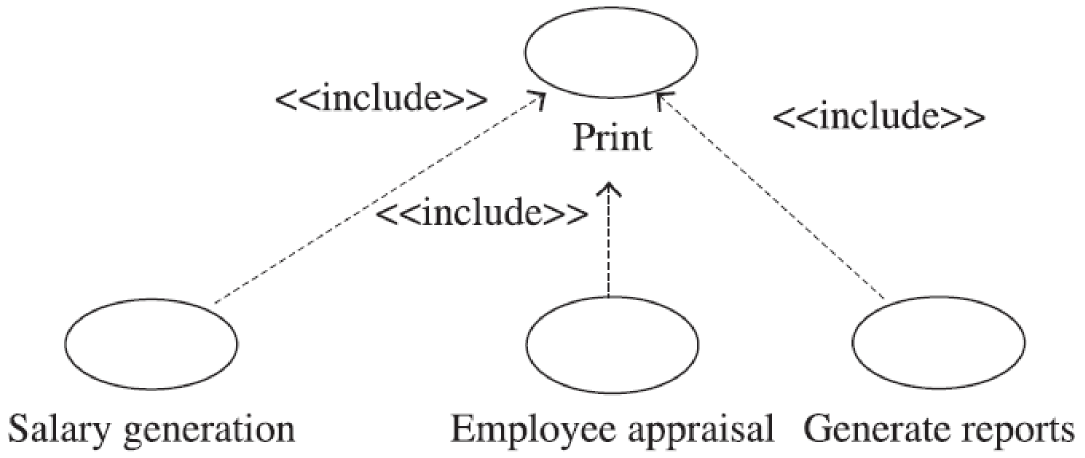
For e.g, a student returns a book to the library. It is possible that the book is late. In that case, we will need to calculate the fine for the book. Therefore, the **Return Book** use case will be extended by **Calculate Fine** Use case.



Include Relationship

Repeated functionalities in many use cases can be modelled into include relationship, using a singe use case.

For e.g, many use cases may require the **Print** function, so it can be converted into a separate use case and *included* in the other use cases.



Use Case Description

Should contain :

- Use Case Title
- Brief Description
- Actors involved
- Flow of events
  - Basic Flow
  - Alternate Flow
- Special Requirement
- Precondition
- Postcondition
- Extension point (related use cases)

Scenario Diagrams

Scenario is a particular path through the use case. In scenario diagrams:

- Basic flow is represented using straight lines.
- Alternate flows represented using curved lines
- Preconditions checked at the start, postconditions at the end.

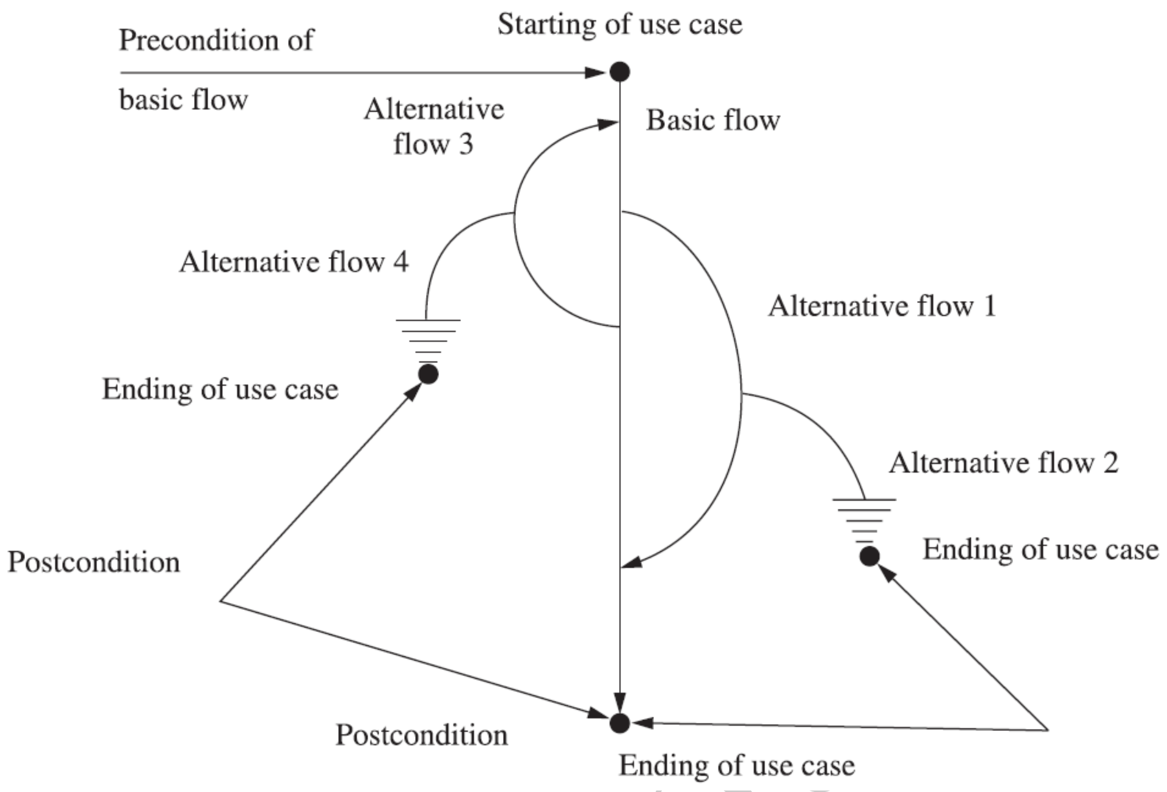


Figure 4: Scenario Diagram

Object Oriented Analysis

Identifies and defines the real world objects that are involved in interaction with the system.

Classes

Class is a collection of objects with common attributes and operations. It is a template that groups attributes and operations together.

Entity Classes

- Persist longer in the system.
- Stored and maintained for a long time
- Contain information needed to complete a task
- AKA domain classes.
- May be used in multiple use cases.

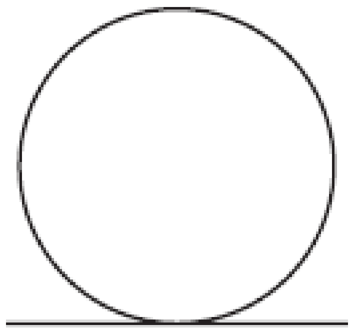


Figure 5: Entity Class

Interface Classes

- Handle interaction in the system.
- Provide interface between actors and the system.
- AKA boundary classes.
- Used to model windows, buttons, etc.
- Interface class usually lasts while the use case is active.
- Generally we need an interface class for each use case. Use cases may also have more than one interface class.
- Dependent on surroundings/environment of the system. Entity and control classes are not dependent on environment or surroundings.

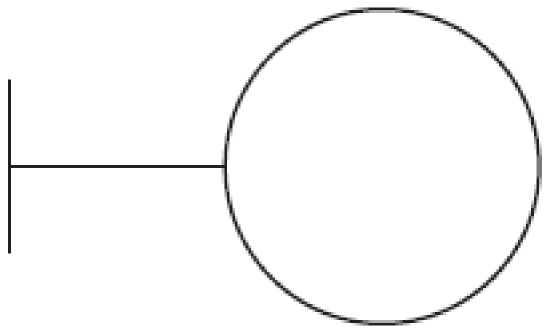


Figure 6: Interface Class

Control Class

- Coordinate and manage entity and interface classes.
- Puts together things so that a use case can be completed.
- Represents dynamics of the system.
- Handles tasks and sequence of events.
- A control object is created when the use case starts and is deleted when the use case finishes.
- Generally, each use case should have its own control class.

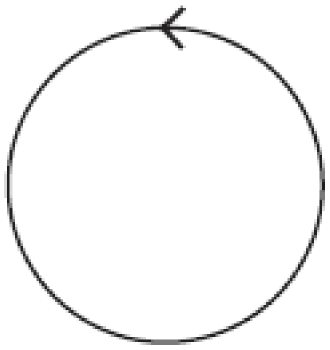
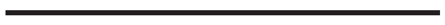
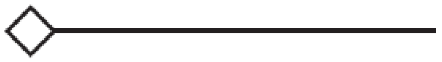
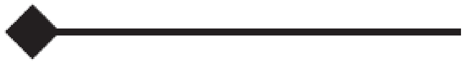
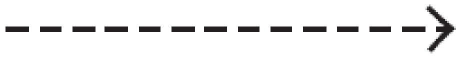
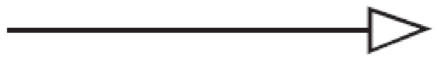


Figure 7: Control Class

Relationships between Classes

Name and Description	Notation
Association is a structural connection, usually bidirectional. Shows that two classes are linked in some way. May also have a name associated with it. For e.g, IssueBookController *manages* Transaction. *manages* is an association with a name.	
Aggregation shows a whole-part relationship, i.e, class B is part of class A. E.g - Book is part of Library. In the notation, the diamond should represent the library, i.e, the *whole* part.	
Composition is strong aggregation. In composition, class B is part of Class A, and *only Class A*. It cannot belong to another class, say C. In aggregation, a class can belong to/be part of 2 classes. For e.g, book can be part of Library as well as Computer Department. But Computer Department can belong to only one University.	
Dependency - represents that a class depends on another class. Unidirectional. It means that one class is affected by changes in the other class, because it uses that class. If class A depends on class B, we show dependency through a dotted arrow from A to B.	

Name and Description	Notation
Generalization - Relationship between parent and child class. Represents inheritance (is a) relationships. For example, Professor is an Employee, SecurityGuard is an Employee, Clerk is an Employee, etc. Represented by an arrow from child class towards parent class.	

Class Diagram

Class diagrams should contain details about each classes as well as the relationships between them. For each class, we need to write:

- the class name
- Type (entity/interface/control)
- Attributes and their types
- Operations

Good Programming Practices

- Meaningful variable names
- Documentation
- Use global data rarely.
- Use consistent formatting
- Use error messages
- Complicated if statements must be avoided. Keep nesting of if statements to minimum.

Software Testing

Testing is the process of executing a program with the intent of finding faults. It's the process of verifying the outcomes of every phase of development and validating the program by executing it with the intention of finding faults.

Verification	Validation
Process of reviewing documents and programs with the intention of finding faults.	Process of evaluating a system at the end of development to see if it satisfies the requirements.
Static Testing	Dynamic testing
Program Not Executed	Program is executed
Inspections, walkthroughs, reviews.	Black/White box testing, non-functional testing
Before validation	After verification
Prevention of Errors	Detection of Errors
Are we building the product right?	Are we building the right product
About the process	About the product.

Verification Techniques

Peer Reviews

- Simple, old and informal
- Applicable to every document and program
- We give the document/program to someone else and ask them to review it and find faults (if any)

- Faults may be reported using a formal report or verbally.
- Effective for small size documents/programs.

Walkthrough

- Group activity
- More formal
- Author presents the document/program to a group of 2-7 people.
- No prior preparation is expected.
- Group can ask questions and make observations, etc. Observations are discussed and changes are suggested.
- The author prepares a detailed report as per changes.

Inspections

- Formal, systematic and effective.
- AKA formal review, formal technical review, etc.
- Group of 3-6 people is there. An independent presenter prepares and understands the document and presents it. The group is given the document earlier and given time to prepare for the inspection.
- Impartial moderator (a senior person) should be the head of the group.
- The meeting is recorded.
- Observations, suggestions, potential faults, etc, are taken and noted.
- Checklist may be used.
- Moderator prepares a report and circulates it to participants. People may suggest changes in the report.

Checklist

Checklist contains important information that a particular document must contain. We can verify the document against our checklist.

It helps identifying duplicate information, missing information, and wrong information.

Every document may have a different checklist.

E.g.

- SRS
  - Are objectives properly stated?
  - Is SRS approved by customer?
  - Is layout of screens well designed?
  - Has IEEE standard been followed?
  - Are major functions stated?
- OO Analysis
  - Are all types of classes identified?
  - Is any class missing?
  - Are relationships between classes identified correctly?
  - Are classes well named?
- OO Design
  - Are all objects in diagrams identified correctly?
  - Are interaction diagrams present for each use case?
  - Are all messages passing correct parameters?

etc.

Functional Testing

- Checks functionality of the system.
- Inputs are given and observed output is compared to expected output.
- Program is treated as a black box. Only functionality is tested, not internals.

Boundary Value Analysis

- Focuses on values close to the boundary of the input domain, because those tests have more chances of failing.
- Checks
  - Min value
  - Max value
  - Just above min value
  - Just below max value
  - Avg value

For e.g, for two inputs $100 \le a \le 1000$ and $200 \le b \le 1200$, boundary value test cases are:

S.No	a	b
1	100	700
2	101	700
3	550	700
4	999	700
5	1000	700
6	550	200
7	550	201
8	550	1199
9	550	1200

Only tests a single fault at once. i.e, a and b both won't simultaneously have boundary conditions. Total number of cases $= 4n + 1$.

Robustness Testing

Also tests *just below minimum* and *just above maximum*, as well as all the test cases tested in boundary value. The program should identify these values as invalid.

Total number of cases $= 6n + 1$

Reference : Object Oriented Software Engineering - Yogesh Singh,Ruchika Malhotra.

Worst Case Testing

We don't assume single fault here. Many failures can occur together. Any combination of inputs may have:

- min value
- max value
- just above min
- just below max
- average value

Total number of cases $= 5^n$

Robust Worst Case

We add *just below min* and *just above max* to worst case testing. Total number of cases $= 7^n$.

Equivalence Class Testing

- Divide input domain into many classes (groups). Any test case in the class should be equivalent to testing the entire class.
- For e.g, for a,b example above, create classes :
  - $I_1 = \{100 \le a \le 1000 \textit{ and } 200 \le b \le 1200\}$. (a and b both valid)
  - $I_2 = \{100 \le a \le 1000 \textit{ and } b < 200\}$. (a valid b invalid)
  - $I_3 = \{100 \le a \le 1000 \textit{ and } b > 1200\}$. (a valid b invalid)

  - $I_4 = \{a < 100 \; and \; 200 \leq b \leq 1200\}$. (a invalid b valid)
  - $I_5 = \{a > 1000 \; and \; 200 \leq b \leq 1200\}$. (a invalid b both valid)
  - Similarly 4 more cases will be there when both inputs are invalid (a<100,b<200), (a>1000,b<200),(a<100,b>1200),(a>10

Structural Testing

The entire program is evaluated. We go into the internal structure of the program to test it. This is white box testing.

Path Testing

- For each line in the code, draw a node. Connect it to the next line. Do the same for loops.
- For if conditions create branches.
- Merge lines that don't have branching into a single node.

Eg:

Code:

```
1. int a;
2. input(a);
3. print("you entered",a);
4. if (a%2==0)
5. {
6.   print("a is even");
7. }
8.else
9. {
10. print("a is odd");
11. }
12. print("Goodbye")
```

The graph will look like:

```
      1
      |
      v
      2
      |
      v
      3
      |
      v
      4
    /   \
   5     8
   |     |
   v     v
   6     9
   |     |
   v     v
   7     10
   |     |
   v     v
   12<--11
```

Remove sequential nodes.

```
    S
    |
    v
    N1
    |
    v
    N2
  /   \
 N3   N4
  \   /
    N5
    |
    v
    N6
```

This is the final DD graph

Independent Paths in DD graphs are paths that introduces at least one new node or edge in its sequence.

Cyclomatic number= max number of independent paths.

It depends only on the decision structure of G. Addition/removal of functional statements doesn't affect cyclomatic number.

V(G) = Cyclomatic Number = e-n+2P

e = edges

n= nodes

P = number of connected components

V(G)= number of regions of program graph.

For activity diagrams,

V(G) = Transitions - activities/branches + 2P

Class Testing

- Done by developers who coded the class
- Each operation is tested
- Stubs or drivers may be written to execute the test.
- Instances of classes are created and tested.
- Unspecified behaviour cannot be tested, so it should never be implemented.
- Testing is done by identifying pre and post conditions for each operation in the class, and testing if they are valid on execution.

Mutation Testing

- We make small changes to the program, such as replacing < by <= , changing (a+b) to (b+c), removing a line, adding a line, changing public to private,etc. Only one small change is made (Generally)
- Changed programs are called mutants
- We run all our test cases against the mutants.
- If the mutant fails *any* test case, it is a killed mutant. If the mutant passes all test cases, it is a live mutant.
- We calculate mutation score as $\frac{Mutants\_Killed}{Total\_Mutants}$
- Mutation score should be as high as possible.
- For every live mutant, we add a new test case to the test suite that will ensure the mutant is killed.

Levels of Testing

Unit Test

- Testing a single class or operation of the class.
- Generally we choose a class as a unit.
- Testing classes that have parents is done by *flattening* the class, which means merging the parent class and child class. This is done so we can test the attributes and operations present in the parent class. Flattening must be undone after testing.
- Instance of class is created and operations are called with appropriate parameters.

Integration Test

- Test various combinations of different units.
- We don't have hierarchical control structure in OO systems, so top-down, bottom-up integration testing isn't valid here.
- Integration testing in OO systems is basically interclass testing.
- The most popular way is thread based testing. We integrate classes that work together to respond to some input. Such classes make up a thread. There can be many threads. Expected vs Observed output of each thread is calculated and compared.
- Another way is to test basic and alternative flow of each use case. A path may require one or more classes.
- Third way is cluster testing. Classes are combined to show one collaboration.

System Testing

- Performed after unit and integration testing.
- The software is tested with its environment.
- System = s/w + h/w + other associated parts.
- Non-functional requirements are also tested here.

Acceptance Testing

- This is carried out by customers.
- May be adhoc or systematic.
- If software isn't developed for particular systems, we identify potential customers and perform testing.
  - If the testing is done at the developer's site under their supervision, in a closed environment, this is called alpha testing.
  - Another approach is giving the software to limited amount of people and asking them to use it. This is beta testing.