PS02CMCA52 Software Engineering

References

- Roger S. Pressman, Software Engineering, A Practitioner's Approach, 6th Edition, TATA Mc-Graw Hill Publication, 2011.
- Pankaj Jalote, An Integrated Approach to Software Engineering, Third Edition, Narosa Publishing House, 2008.
- Rajib Mall, Fundamentals of Software Engineering, 2nd Edition, Prentice-Hall of India, 2006.
- Ian Sommerville : Software Engineering, 9th edition, Pearson Education, 2011.
- Waman S Jawadekar, Software Engineering Principles and Practice, 2nd Reprint,Tata McGraw Hill, 2008.

❖ SOFTWARE

- **Software** is (1) instructions (***computer programs***) that when executed provide desired features, function, and performance; (2) ***data structures*** that enable the programs to adequately manipulate information; and (3) ***documents*** that describe the operation and use of the programs.
  (*Reference : Roger S. Pressman : Software Engineering, A Practice Approach*)

- Software is a set of instructions or computer programs, and data used to operate computers and execute specific tasks. Software is a generic term used to refer to applications, scripts and programs that run on a device.
- To gain an understanding of software, it is important to examine the ***characteristics*** of software.
- Software is a ***logical*** rather than a physical system element.
- Software ***doesn't "wear out"***.
- Software is ***developed*** or ***engineered***; it is not manufactured in the classical sense.
- Although the industry is moving toward **component-based construction**, most software continues to be ***custom built***.

❖ Software refers to a collection of computer programs, procedures, associated documentation and data pertaining to the operation of a computer system.

- The IEEE definition of software lists the following components of software :
  - Computer program(s)
  - Procedures
  - Documentation
  - Data necessary for operating the software system

Software Applications can be found in various domains including business, education, government, agriculture, healthcare, engineering, etc.

Characteristics of a Good Software

- Correct
- Maintainable
- Easy to modify
- Well modularized with well-designed interface
- Reliable and robust
- Having a good user-interface
- Well documented
    * Internal documentation for maintenance and modification
    * External documentation for end users
- Efficient
    * Efficient usage of system resources
    * Optimized data structures and algorithms

System software versus application software : They both differ in terms of their purpose and design. System software is meant to administer the system resources. It also serves as a kind of platform for running the application software. On the other hand, application software is meant to enable the user to carry out some specific set of tasks or functions.

Examples of System Software

- Operating Systems
- Editors
- Compilers
- Interpreters
- Assemblers
- Loaders
- Linkers
- Device Divers
- Programming Language Translators
- Firmwares
- Utilities

Examples of Application Software

- Library Management System
- Online Shopping System
- ERP software
- Application software for the domain of Healthcare
- Application software for the domain of Agriculture, like Soil Healthcard System
- Word Processors
- Database Software
- Multimedia Software
- Education and Reference Software
- Graphics Software
- Web Browsers

#

Software Engineering - meaning, goal, challenges and approach

Software engineering is defined as the systematic approach to the development, operation, maintenance, and retirement of software.

(*Reference : Pankaj Jalote : A Concise Introduction to Software Engineering*)

Besides delivering software, high quality, low cost, and low cycle time are also **goals** which software engineering must achieve. In other words, the systematic approach must help achieve a high quality and productivity (Q&P). In software, the three main factors that influence Q&P are people, processes, and technology. That is, the final quality delivered and productivity achieved depends on the skills of the people involved in the software project, the processes people use to perform the different tasks in the project, and the tools they use.

- The **IEEE** has developed the following definition :
- **Software Engineering** is defined as

the application of a systematic, disciplined, and quantifiable **approach** to the ***development, operation, and maintenance of software***.

- Software engineering is the **science** and **art** of building ***high quality software*** systems
  - On ***time***
  - Within ***budget***
  - With ***correct operation***
  - With ***acceptable performance***

- ***Software engineering is the establishment and use of sound engineering principles in order to obtain economically a software that is reliable and works efficiently on real machines.***

  - Fritz Bauer
- This definition serves as a basis for discussion. However, it says little about the technical aspects of ***software quality***; it does not directly address the need for ***customer satisfaction*** or ***timely delivery*** of a product; it omits mention of the importance of ***measurement and metrics***; it does not state the importance of an ***effective process***.

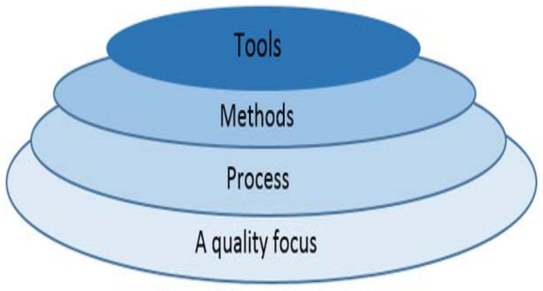
Goals of Software Engineering

- To produce software that is ***absolutely correct***.
- To produce software with ***minimum effort***.
- To produce software at the ***lowest possible cost***.
- To produce software in the ***least possible time***.
- To produce software that can be ***easily maintained and modified***.

- The challenge of software engineering is to see how close we can get to achieving these goals. The art of software engineering - balancing these goals for a particular project.

<u>**Software Engineering – A Layered Technology**</u>

(*Reference : Roger S. Pressman : Software Engineering, A Practitioner's Approach*)

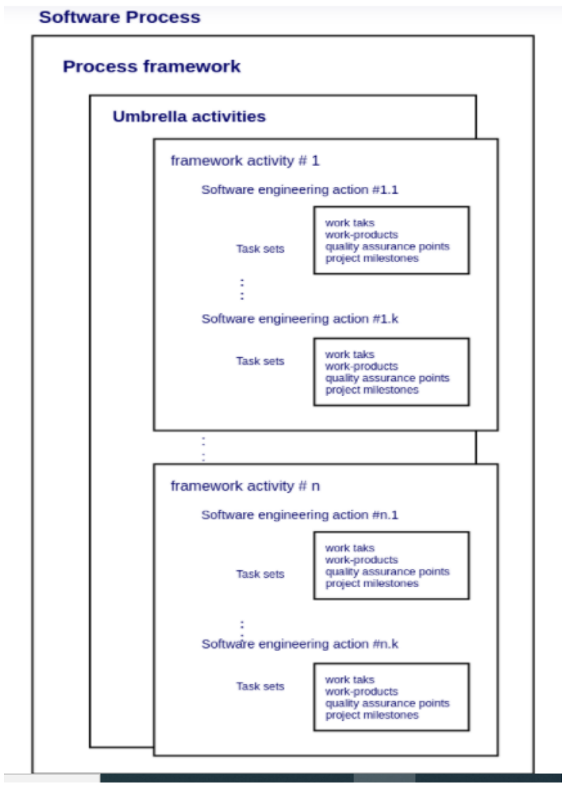


(Figure: Software Engineering Layers)

- Software engineering is a layered technology.
- Any engineering approach (including software engineering) must rest on an organizational commitment to <u>***quality***</u>.
- Total Quality Management, Six Sigma, and similar philosophies foster a ***continuous process improvement*** culture, which ultimately leads to the development of increasingly more effective approaches to software engineering.
- The foundation for software engineering is the <u>***process layer***</u>.
- Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software.

- **Process** defines a framework that must be established for effective delivery of software engineering technology.
- The <u>**software process**</u> forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.
- <u>**Software engineering methods**</u> provide the technical "how to's" for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support.
- Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.
- <u>**Software engineering tools**</u> provide automated or semi-automated support for the process and the methods.
- When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.

A Process Framework



(*Source : Roger S. Pressman : Software Engineering, A Practitioner's Approach*)

Software Process

- A software process comprises activities performed to create a software product. It deals with the technical and management aspects of software development.
- A Software process includes various tasks, actions, and activities.
- A process framework for software engineering defines five **framework activities**. Framework activities include communication, planning, modeling, construction and deployment. Each framework activity includes a set of engineering actions and each action defines a set of tasks that incorporates work products, project milestones and software quality assurance (SQA) points that are required. **Umbrella activities** are carried throughout the process.

Process Framework Activities

- **Communication** – Communicate with stakeholders and customers to obtain objectives of the system and requirements for the software.
- **Planning** – Software project plan has details of resources needed, tasks and risk factors likely to occur, schedule.

- **Modeling** – Architectural models and design to better understand the problem and for work towards the best solution.
- **Construction** – Generation of code and testing of the system to rectify errors and ensuring all specified requirements are met.
- **Deployment** – Entire software product or partially completed product is delivered to the customer for evaluation and feedback.

Umbrella activities

- Activities that occur throughout a software process for better management and tracking of the project.
- **Software project tracking and control** – Compare the progress of the project with the plan and take steps to maintain a planned schedule.
- **Risk management** – Evaluate risks that can affect the outcome and quality of the software product.
- **Software quality assurance (SQA)** – Conduct activities to ensure the quality of the product.
- **Technical reviews** – Assessment of errors and correction done at each stage of activity.
- **Measurement** – All the measurements of the project and product features.
- **Software configuration management (SCM)** – Controlling and tracking changes in the software.
- **Reusability management** – Back up work products for reuse and apply the mechanism to achieve reusable software components.
- **Work product preparation and production** – Project planning and other activities used to create work product are documented.

Software Development Process Models

In the software development process, we focus on the activities directly related to production of the software, for example, design, coding, and testing. As the development process specifies the major development and quality control activities that need to be performed in the project, the development process really forms the core of the software process. The management process is decided based on the development process. Due to the importance of the development process, various models have been proposed.

(*Reference : Pankaj Jalote : An Integrated Approach to Software Engineering*)

Some of the well-known software development process models are given below :

* Waterfall Model
* Prototyping
* Iterative Development
* Timeboxing Model

(*Reference : Pankaj Jalote : An Integrated Approach to Software Engineering*)

<u>**Waterfall Model**</u>

- The simplest process model, which states that the phases are organized in a linear order.
- The model was originally proposed by Royce, though variations of the model have evolved depending on the nature of activities and the flow of control between them.
- In this model, a project begins with feasibility analysis.
- Upon successfully demonstrating the feasibility of a project, the requirements analysis and project planning begins.
- The design starts after the requirements analysis is complete, and coding begins after the design is complete.
- Once the programming is completed, the code is integrated and testing is done.
- Upon successful completion of testing, the system is installed.
- After this, the regular operation and maintenance of the system takes place. The model is shown in the Figure.

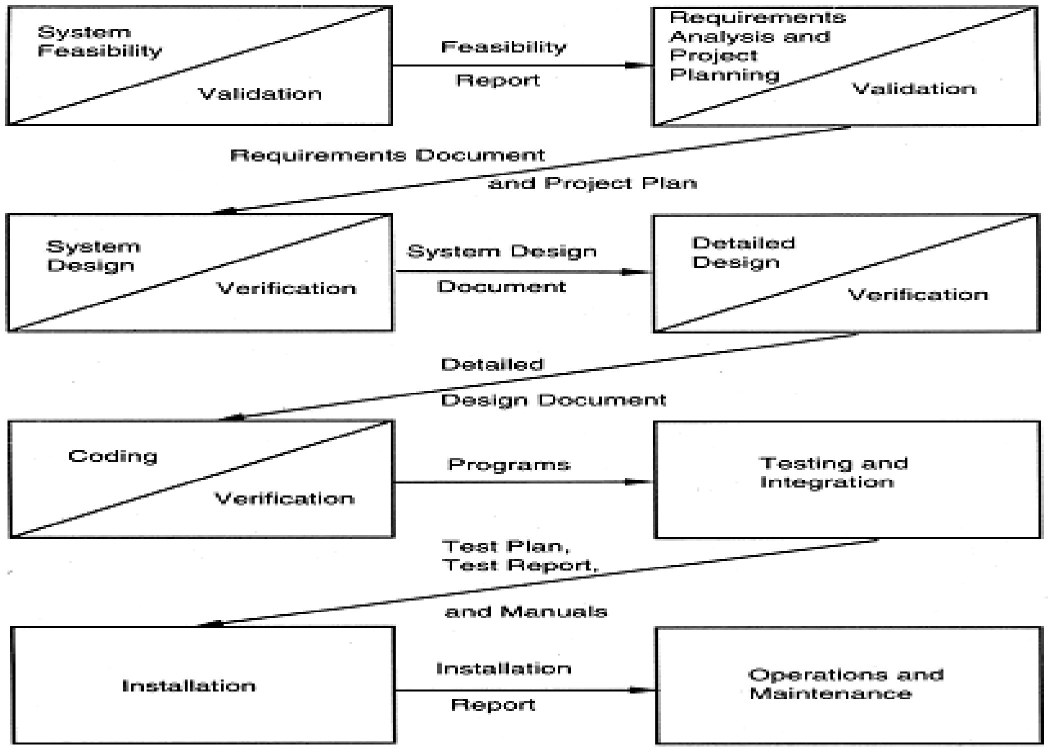


Figure : The waterfall model
(*Source : "An Integrated Approach to Software Engineering" by Pankaj Jalote*)

The requirements analysis phase is mentioned as “analysis and planning”. Planning is a critical activity in software development. A good plan is based on the requirements of the system and should be done before later phases begin. However, in practice, detailed requirements are not necessary for planning. Consequently, planning usually overlaps with the requirements analysis, and a plan is ready before the later phases begin. This plan is an additional input to all the later phases.

Linear ordering of activities has some important consequences. First, to clearly identify the end of a phase and the beginning of the next, some certification mechanism has to be employed at the end of each phase. This is usually done by some verification and validation means that will ensure that the output of a phase is consistent with its input (which is the output of the previous phase), and that the output of the phase is consistent with the overall requirements of the system. The consequence of the need for certification is that each phase must have some defined output that can be evaluated and certified. That is, when the activities of a phase are completed, there should be some product that is produced by that phase.

The outputs of the earlier phases are often called ***work products*** and are usually in the form of documents like the requirements document or design document. For the coding phase, the output is the code. Though the set of documents that should be produced in a project is dependent on how the process is implemented, the following documents generally form a reasonable set that should be produced in each project :

- Requirements document
- Project plan
- Design documents (architecture, system, detailed)
- Test plan and test reports
- Final code
- Software manuals (e.g., user, installation, etc.)

In addition to these work products, there are various other documents that are produced in a typical project. These include review reports, which are the outcome of reviews conducted for work products, as well as status reports that summarize the status of the project on a regular basis. Many other reports may be produced for improving the execution of the project or project reporting.

One of the main advantages of this model is its simplicity. It is conceptually straight-forward and divides the large task of building a software system into a series of cleanly divided phases, each phase dealing with a separate logical concern. It is also easy to administer in a contractual setup – as each phase is completed and its work product produced, some amount of money is given by the customer to the developing organization.

Limitations of the Waterfall Model

The waterfall model, although widely used, has some limitations. Some of the key limitations are :

1. It assumes that the requirements of a system can be frozen (i.e., baselined) before the design begins. This is possible for systems designed to automate an existing manual system. But for new systems, determining the requirements is difficult as the user does not even know the requirements. Hence, having unchanging requirements is unrealistic for such projects.

2. Freezing the requirements usually requires choosing the hardware (because it forms a part of the requirements specification). A large project might take a few years to complete. If the hardware is selected early, then due to the speed at which hardware technology is changing, it is likely that the final software will use a hardware technology on the verge of becoming obsolete. This is clearly not desirable for such expensive software systems.

3. It follows the "big bang" approach – the entire software is delivered in one shot at the end. This entails heavy risks, as the users do not know until the very end what they are getting. Furthermore, if the project runs out of money in the middle, then there will be no software. That is, it has the "all or nothing" value proposition.

4. It is a document-driven process that requires formal documents at the end of each phase.

Despite these limitations, the waterfall model has been the most widely used process model. It is well suited for routine types of projects where the requirements are well understood. That is, if the developing organization is quite familiar with the problem-domain and the requirements for the software are quite clear, the waterfall model works well.

Spiral Model

- This is a recent model that has been proposed by Boehm.
- As the name suggests, the activities in this model can be organized like a spiral.
- The spiral has many cycles.
- The radial dimension represents the cumulative cost incurred in accomplishing the steps done so far and the angular dimension represents the progress made in completing each cycle of the spiral.
- Each cycle in the spiral begins with the identification of objectives for that cycle, the different alternatives that are possible for achieving the objectives, and the constraints that exist.
- The next step in the cycle is to evaluate these different alternatives based on the objectives and constraints.
- The focus of evaluation in this step is based on the risk perception for the project.
- The next step is to develop strategies that resolve the uncertainties and risks. This step may involve activities such as benchmarking, simulation, and prototyping.
- Next, the software is developed, keeping in mind the risks. Finally the next stage is planned.

- As it can also be seen in the diagram, **the spiral model** is divided into four major quadrants. Therefore, apart from the loop divisions, the spiral model is also divided into quadrants which further divide and categorize these loops and each of these divisions contains a set of activities that are performed while the software development. Each of the quadrants of the spiral model performs the following functions:
- **First quadrant:**
  Sets the objective of the software and analyses all the risks associated with the software.
- **Second Quadrant:**
  This quadrant deals with the complete analysis of each of the risks analyzed in the first quadrant. Apart from that, the risk reduction is also taken care of here.
- **Third quadrant:**
  This quadrant includes all the development and validation part which includes coding, testing, and other stuff.
- **Fourth quadrant:**
  The fourth quadrant deals with the final results that we are getting from the software. It involves the review, planning, and maintenance of the software.

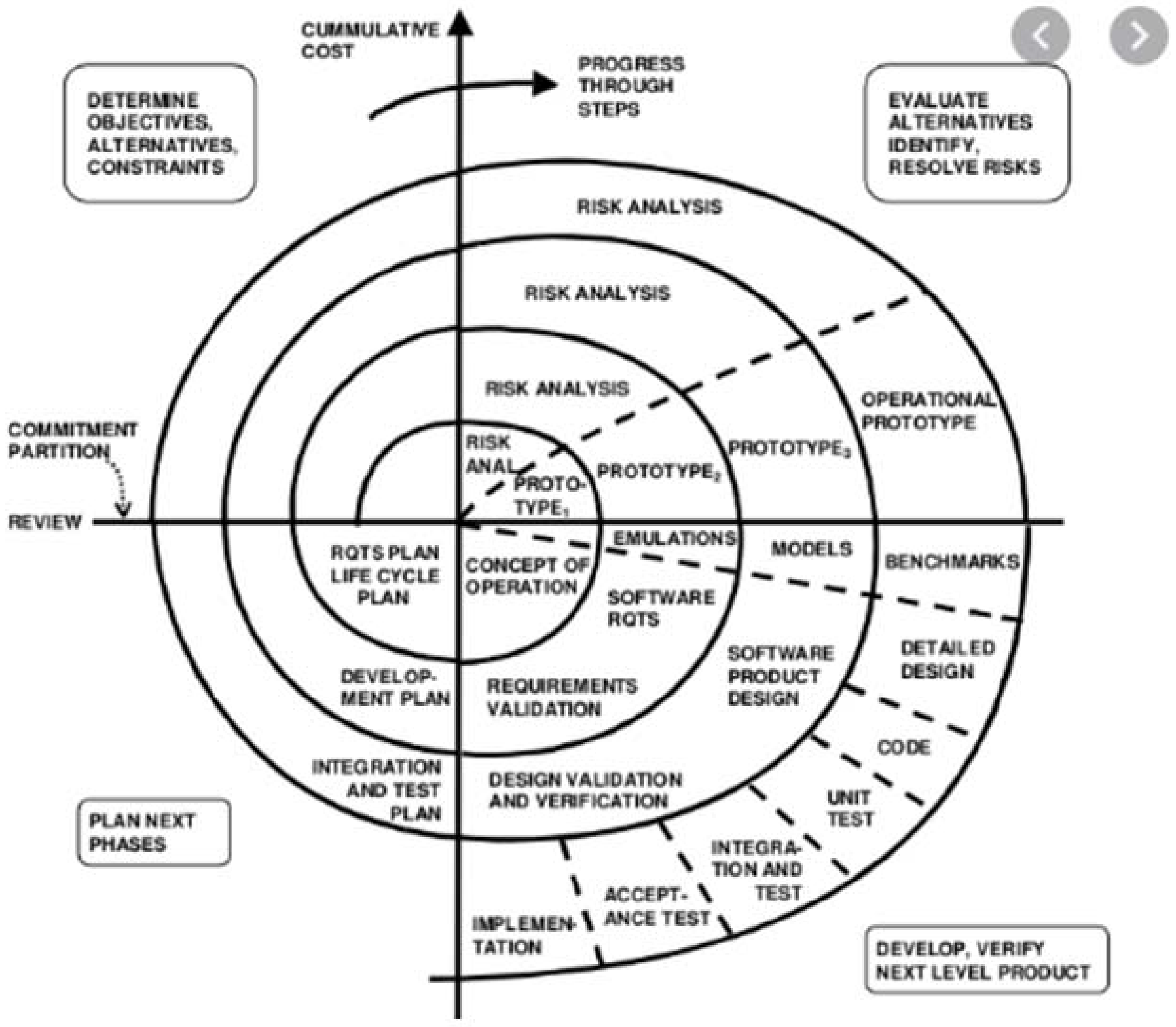


Figure : The spiral model

Unit 3

Software Design

The design of a system is a plan for a solution such that if the plan is implemented, the implemented system will satisfy the requirements of the system and will preserve its architecture. The design activity is a two-level process. The first level produces the system design which defines the modules needed for the system, and how the components interact with each other. The detailed design refines the system design, by providing more description of the processing logic of components and data structures. A design methodology is a systematic approach to creating a design. Most design methodologies concentrate on system design. During system design a module view of the system is developed, which should be consistent with the component view created during architecture design.

Design approaches

- Function-oriented design
- Object-oriented design

Function-Oriented Design

Design principles

- Problem partitioning and hierarchy
- Abstraction
- Modularity
- Top-down and bottom-up strategies

Module-level concepts

- Coupling
- Cohesion

Criteria for Quality of Software Design

Thumb Rules for design Evaluation

Quality Criteria

- **Correctness**
- **Completeness** – Implements all the specifications
- **Verifiable** design
- **Traceability** – All design elements can be traced to some requirements
- **Efficiency** – Concerned with the proper use of scarce/expensive resources
- **Simplicity / Understandability**- Produce designs that are simple to understand
- **Maintainability** – Requires thorough understanding of different modules and their interconnectivities
- **Best possible design** – There can be many correct designs. Select the best one.

Design Principles

Problem Partitioning and Hierarchy

- Solving large/complex problems
- "Divide and conquer"
- Divide into smaller pieces, so that each piece can be conquered separately.
- Goal : Divide the problem into manageably small pieces that can be solved separately.
- Basic rationale / belief : If the pieces of a problem are solvable separately, the cost of solving the entire problem is more than the sum of the costs of solving all the pieces.
- As the #components increases, the cost of partitioning also increases and the cost of added complexity increases.
- When the cost of added complexity is greater than the savings achieved by partitioning, we should stop partitioning.

Abstraction

- An abstraction of a component describes the external behavior of that component without worrying about the internal details that produce the behavior.
- A tool that permits the designer to consider a component at an abstract level without worrying about the details of implementation of the component.
- Any component / system – provides some services to its environment.
- Allows the designer to concentrate on one component at a time using the abstraction of other components.

Modularity

- A system is considered modular if it consists of discrete components so that each component can be implemented separately and a change to one component has minimal impact on other components.
- Modularity helps in system debugging – Isolating a system problem to a component is easier if the system is modular.
- Modularity helps in system building – A modular system can be easily built by putting its modules together.
- A system is partitioned into modules so that **modules are**
  - solvable separately
  - modifiable separately
  - compilable separately
  - implementable separately.

- Software is divided into separately named and addressable components (modules)

- Why do we prefer modularization ?

* To ease planning for implementation (coding)
* To easily accommodate changes
* To effectively test and debug programs
* To conduct long-term maintenance without serious side-effects

Module-Level Concepts

Coupling and Cohesion

Coupling :

- Coupling between modules is the strength of interconnections between modules or a measure of interdependence among modules.
- In general, the more we must know about module A in order to understand module B, the more closely connected A is to B.
- "Highly coupled" modules are joined by strong interconnections.
- "Loosely coupled" modules have weak interconnections.
- Independent modules have no interconnections.
- **To solve and modify a module separately, we would like a module to be loosely coupled with other modules.**
- The coupling between modules is largely decided during system design.
- Coupling increases with the complexity and obscurity of the interface between modules.

Factors affecting coupling

- Interface complexity
- Type of connection
- Type of communication

Coupling	**Interface complexity**	**Type of connection**	**Type of communication**
Low	Simple/Obvious	To module By name	Data Control
High	Complicated /Obscure	To internal elements	Hybrid

- Coupling is reduced when elements in different modules have little or no bonds between them.
- Coupling can be reduced if relationships among elements in different modules is minimized.
- Coupling increases if a module is used by other modules via indirect/obscure interface – like using shared variables.

- Complexity of an interface : refers to the #items being passed as parameters and the complexity of items.
- An interface is used to support communication between modules.

Cohesion of a module:

- Cohesion of a module refers to the relationship between elements of the same module. It indicates how closely the elements of a module are related to each other.
- Cohesion of a module represents how tightly bound the internal elements of a module are to one another.
- It is possible to strengthen the bonds between elements of the same module, by maximizing the relationship between elements of the same module.
- Cohesion and coupling are related.
- Usually, the greater the cohesion of each module, the lower the coupling between modules.

Types of Cohesion

- **Functional Cohesion:** Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' payroll displays functional cohesion.
- **Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
- **Communicational Cohesion:** A module is said to have communicational cohesion, if all the elements of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.
- **Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.
- **Temporal Cohesion:** When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up, shut-down of some process, etc. exhibit temporal cohesion.
- **Logical Cohesion:** The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.
- **Coincidental Cohesion:** The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

Types of Coupling

- Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

Types of Coupling:

- **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent to each other and communicating through data. Module communications don't contain tramp data. Example-customer billing system.
- **Stamp Coupling** Two modules are stamp coupled, if they communicate using a composite data item, such as a structure in C or a record in Pascal.
- **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.
- **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
- **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses and reduced maintainability.
- **Content Coupling:** In a content coupling, one module can modify the data of another module or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

Design Concepts for Object-Oriented Design

Information Hiding:

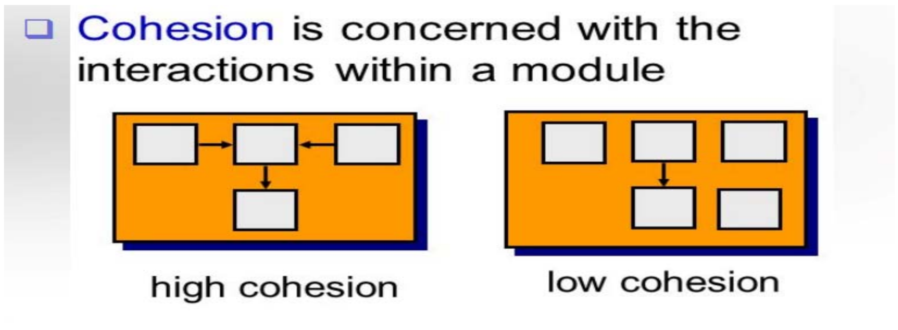
- Suggests that **software components** should be designed in such a way that **information** (data structure and algorithms) contained within a software component is directly **not accessible to other software components**.
- Implies that effective modularity can be achieved by defining a set of independent software components that communicate with one another only necessary information.

Consider the following example :

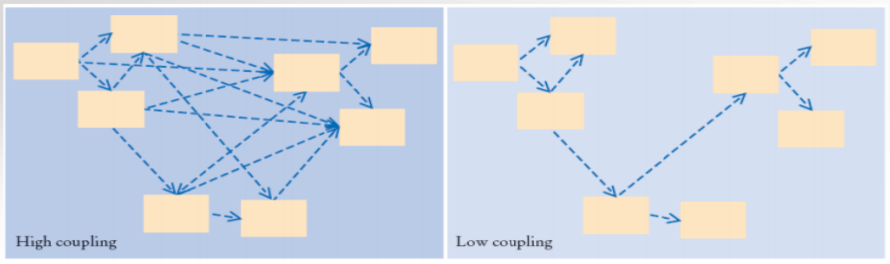
```
Class Time
  {
    public :
              void Display();   // member function –declared public, accessible to all
    private :
             int ticks;  // data member- declared private, external users cannot access it
  };
```

Functional Independence:

- Achieved by developing independent modules and aversion to excessive interaction with other modules.
- Module independence is measured by two quality criteria : cohesion and coupling.
- Cohesion is an indication of relative functional strength of a module. A cohesive module should ideally do just one thing.
- Coupling is an indication of relative inter-dependence among modules.
- Coupling depends on the interface complexity between modules (the point at which entry/reference is made to a module). It depends on what data pass across the interface.



❑ Coupling:



High and Low Coupling between Modules

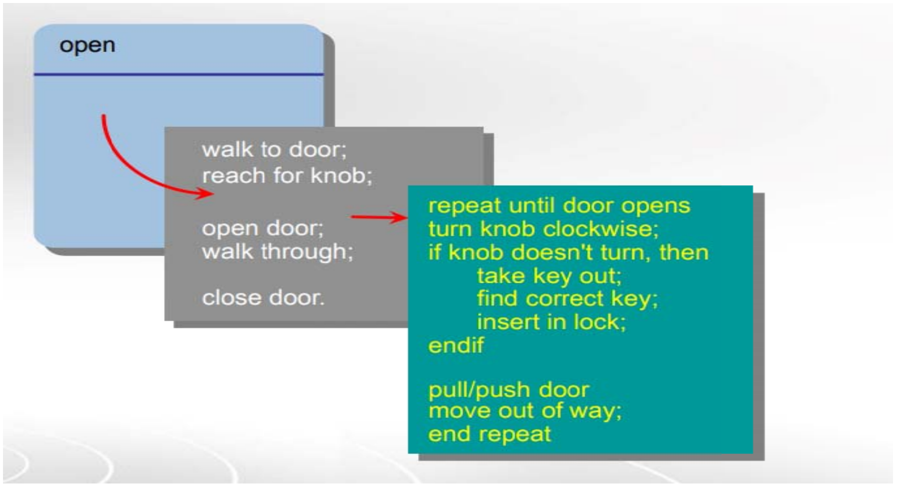
Refactoring :

- A process of changing a software system in such a way that it does not alter the external behavior of the code(design) yet improves the internal structure (Fowler, 1999).
- When a software is refactored, the existing design is examined for
  - * redundancy
  - * unused design elements
  - * inefficient/unnecessary algorithms
  - * poorly constructed/inappropriate data structures
  - * or any other design failure that can be corrected to yield better design
- Result : Software that is easy to integrate/test/maintain

Refinement :

- A top-down design strategy where a program is developed by successively refining levels of procedural details in a hierarchical manner.
- It is an elaboration process that begins at a high level of abstraction.
- Abstraction and refinement are complementary.

While abstraction hides unnecessary details from outsiders, refinement helps to reveal low-level details and design progresses.



Object-oriented approach : Some important concepts and terms

Object-Oriented Design

Overview of important concepts

Objects :

- In the object-oriented approach, ***a system*** is designed as ***a set of interacting objects***.
- An Object : may represent ***a tangible real-world entity***, such as an employee, a book, a library member, etc. An object sometimes may also represent ***some conceptual entity***, e.g. a scheduler, a controller, etc.
- When a system is analyzed, developed, and implemented in terms of natural objects occurring in it, it becomes easier to understand the design and implementation of the system.
- Each object consists of some ***data*** that are **private** to the object and ***a set of functions (or operations)*** that operate on these data.
- An object cannot directly access the data internal to another object. However, an object can indirectly access the internal data of other objects by invoking the operations (i.e. methods) supported by those objects. This mechanism is popularly known as ***data abstraction***.
- Data abstraction means that each object hides from other objects the exact way in which its internal information is organized and manipulated. It only provides a set of methods, which other objects can use for accessing and manipulating this priivate information of the object.

Class :

- Similar objects constitute a class.
- Objects possessing similar attributes and displaying similar behavior constitute a class.
- For example, a set of employees can constitute a class in an employee pay-roll system.
- A class serves as a template for object creation.
- Since each object is created as an instance of some class, classes can be considered as abstract data types (ADTs).

Methods and Messages :

- The **operations supported by an object** are called its ***methods***.
- Methods are the only means available to other objects for accessing and manipulating the data of another object.
- The methods of an object are invoked by sending messages to it.
- The set of valid messages to an object constitutes its protocol.

Inheritance :

- The inheritance feature allows us to define a new class by extending or modifying an existing class.
- The original class is called the ***base class (or superclass)*** and the new class obtained through inheritance is called the ***derived class (or subclass)***.
- The base class contains only those properties that are common to all the derived classes.
- Each derived class is a specialization of its base class because it modifies or extends the basic properties of the base class in certain ways.

- In addition to inheriting all the properties of the base class, a derived class can define new properties. That is, a derived class can define new data and methods.
- A derived class can even give new definitions to methods which already exist in the base class.
- Redefinition of the methods which existed in the base class is called method overriding.

Multiple inheritance

- A mechanism by which a subclass can inherit attributes and methods from more than one base classes.

Polymorphism

Polymorphism literally means poly (many) morphism (forms).

- The same message can result in different actions when received by different objects. This is also referred to as static binding. This occurs when multiple methods with the same name exist.
- When we have an inheritance hierarchy, an object can be assigned to another object of its ancestor class. A method call to the ancestor object results in the invocation of the appropriate method of the object of the derived class. Since the exact method to which a method call would be bound cannot be known at compile time, and is dynamically decided at the runtime, this is also known as dynamic binding.

Unified Modeling Language (UML)

- UML is a modeling language
  - Not a system design or development methodology
- Used to document object-oriented analysis and design
- Independent of any specific design methodology.
- Developed in early 1990s
  - To standardize the large number of object-oriented modeling notations that existed.

Different software development houses were using different notations to document their object-oriented designs

- Adopted by Object Management Group (OMG) as a de facto standard in 1997.
  OMG is an association of industries which tries to facilitate early formulation of standards.
- Provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create models of systems.
- Models - very useful in documenting the object-oriented analysis and design results obtained using some methodology.
- UML contains an extensive set of notations and suggests construction of many types of diagrams.

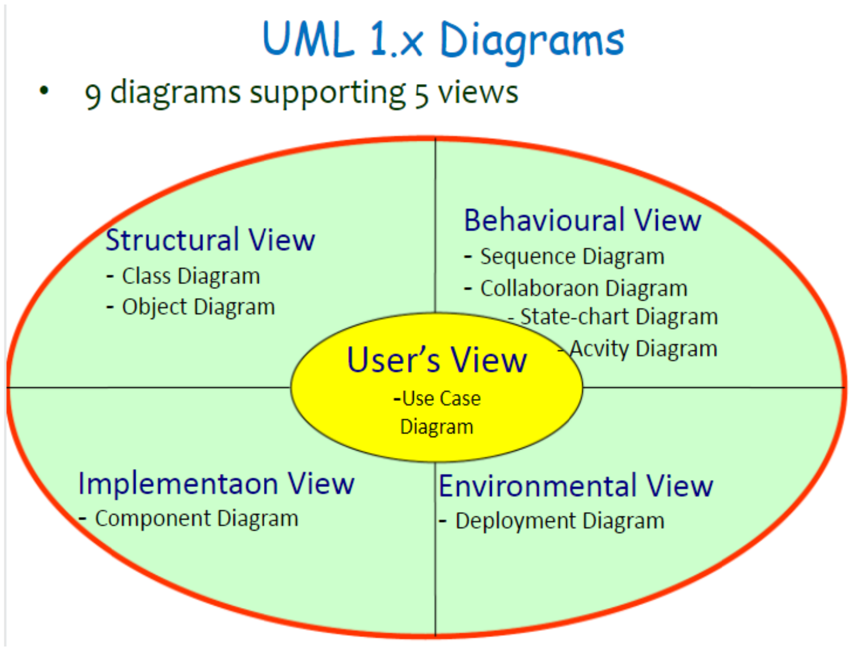
What is a model ?

- A model captures aspects important for some applications while omitting (or abstracting) the rest.
- A model in the context of software development can be ***graphical, textual, mathematical, or program code-based***.
- Graphical models – very popular, easy to understand and construct.
- It helps in managing complexity.
- Can be used for a variety of purposes during software development, including the following :
  Analysis, Specification, Design, Code generation, Testing, Understanding the problem, and Working of a system.

UML Diagrams

- UML can be used to construct ***9 different types of diagrams*** to capture different views of a system.
- Different UML diagrams ***provide different perspectives*** of the software system to be developed.
- ***Facilitate comprehensive understanding of the system***.
- The UML diagrams can capture the following views of a system :

  * Structural view
  * Behavioral view
  * Implementation view
  * Environmental view



Use Case Model

- The use case model for any system consists of a set of “use cases”.
- Use cases represent different ways in which a system can be used by the users.
- A simple way to find all the use cases of a system :

  Ask : “What the users can do using the system ?”
- The use cases partition the ***system behavior*** into ***transactions***, such that each transaction performs ***some useful action*** from the ***user’s point of view***. Each transaction may involve either a single message or multiple ***message exchanges*** between the ***user and the system*** to complete itself.

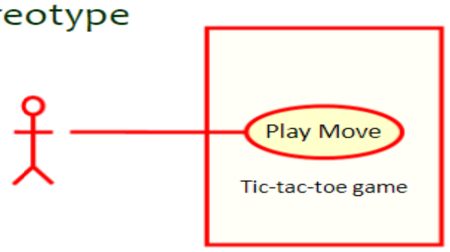
- The use case model is an important ***analysis and design*** artifact.
- Other UML models must conform to the use case model in any ***use case-driven*** or ***user-centric analysis and development approach***.
- The use case model represents a ***functional or process model*** of a system.

- ***Use cases*** can be represented by drawing a ***use case diagram*** and writing an ***accompanying text*** elaborating the drawing.
- Each ***use case*** is ***represented by*** an ***ellipse*** with the name of the use case written inside the ellipse.
- ***All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary.***
- The name of the system being modeled (e.g. Library Information System) appears inside the rectangle.
- The ***different users*** of the system are represented by using the ***stick person icon***.
- Each ***stick person icon*** is normally referred to as an ***actor***.

- An actor is a ***role played by a user*** with respect to the system use.
- It is possible that the same user may play the role of multiple actors.
- Each actor can participate in one or more use cases.
- The ***line*** connecting the ***actor*** and the ***use case*** is called the communication relationship.
- It indicates that ***the actor makes use of the functionality provided by the use case.***
- Both the ***human users*** and ***external systems*** can be represented by ***stick person icons.***
- When a stick person icon represents an external system, it is annotated by the stereotype <<external system>>.
- ***Stereotyping*** can be used to give a ***special meaning*** to any ***basic UML construct***.

Representation of Use Cases

- Represented in a use case diagram
  - A Use Case is represented by an ellipse
  - System boundary is represented by a rectangle
  - Users are represented by stick person icons (actor)
  - Communication relationship between actor and Use Case by a line
- External system by a stereotype



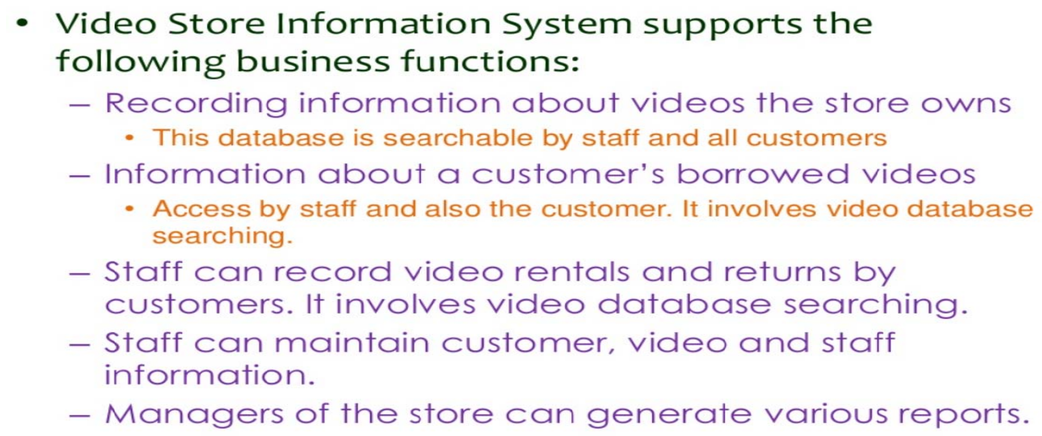
- A ***use case*** typically represents ***a sequence of interactions between a user and a system.***
- These interactions consist of one mainline sequence.
- Mainline sequence : represents the normal interaction between a user and a system. The mainline sequence is the most frequently occurring sequence of interactions.
- For example, in the ***mainline sequence*** of the ***withdraw-cash***, the ***use case*** supported by a ***bank ATM*** would be :
  - * insert a card
  - * enter a password
  - * select the amount withdraw option
  - * enter the amount to be withdrawn
  - * complete the transaction
  - * get the amount

- Several variations to the mainline sequence may also exist.
- A ***variation*** from the mainline sequence occurs ***when some specific conditions hold***.
- These ***variations*** are also called ***alternate scenarios*** or ***alternate paths***.
- For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the account balance.
- A use case can be viewed as a set of related scenarios tied together by a common goal.
- ***The mainline sequence and each of the variations are called scenarios or instances of the use case.***
- Each scenario is a single path of user events and system activity through the use case.

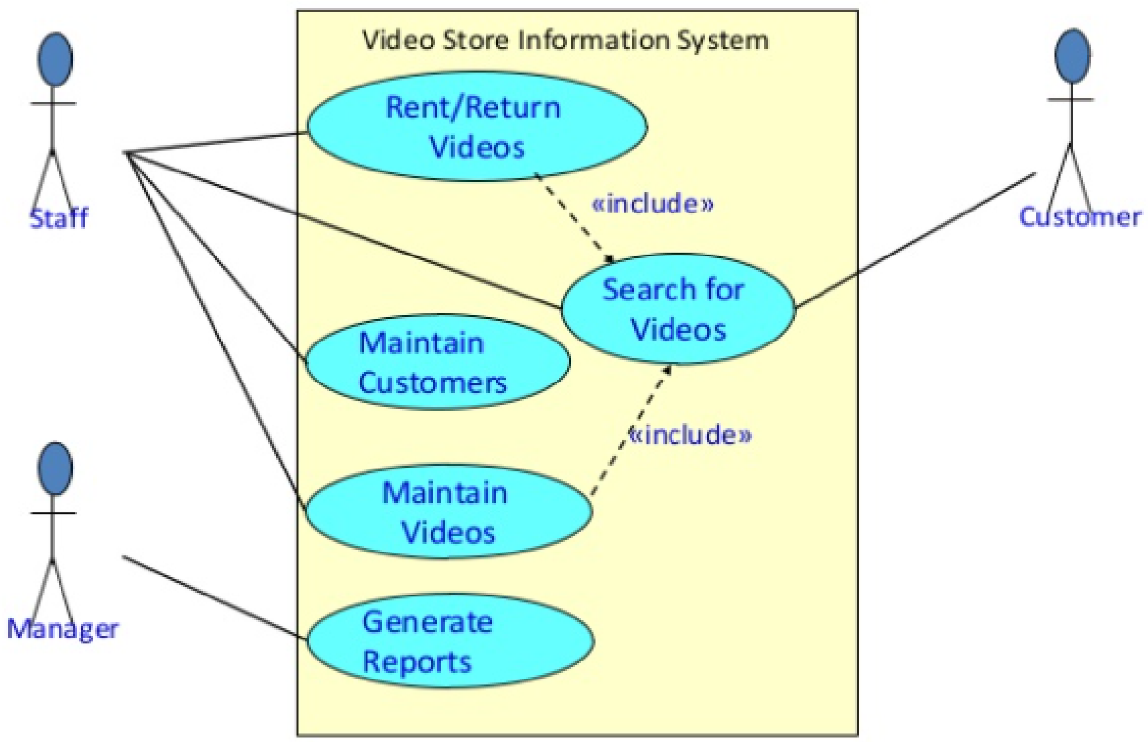
Use Case terms

Actor : A person or a system which uses the system being built for achieving some goal.
Primary actor : The main actor for whom a use case is initiated and whose goal satisfaction is the main objective of the use case.
Scenario : A set of actions that are performed to achieve a goal under some specified conditions.
Main success scenario: describes the interaction if nothing fails and all steps in the scenario succeed.
Extension scenario: Describe the system behavior if some of the steps in the main scenario do not complete successfully.

Example



- Video Store Information System supports the following business functions:
  - Recording information about videos the store owns
    - This database is searchable by staff and all customers
  - Information about a customer's borrowed videos
    - Access by staff and also the customer. It involves video database searching.
  - Staff can record video rentals and returns by customers. It involves video database searching.
  - Staff can maintain customer, video and staff information.
  - Managers of the store can generate various reports.

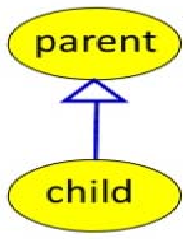


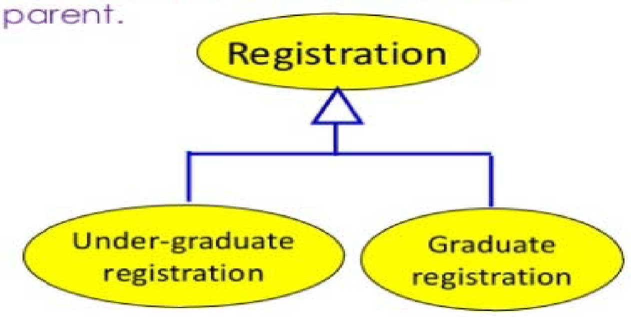
Factoring Use Cases

- Two main reasons for factoring:
  - Complex use cases need to be factored into simpler use cases
  - To represent common behaviour across different use cases
- Three ways of factoring:
  - Generalization
  - Include
  - Extend

Generalization

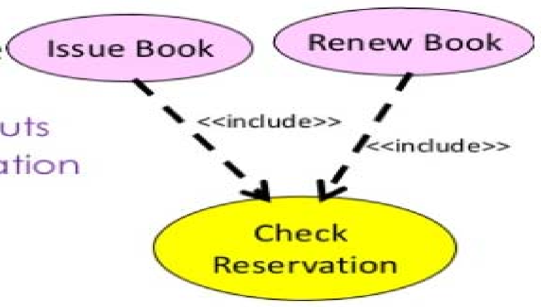
- The child use case inherits the behaviour of the parent use case.
  - The child may add to or override some of the behavior of its parent.

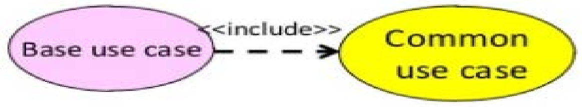




Include

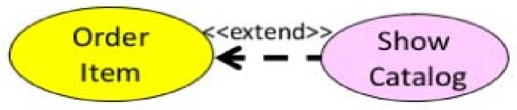
- When you have a piece of behaviour that is similar across many use cases
  - Break this out as a separate use-case and let the other ones "include" it
- Examples of use case include
  - Validate user interaction
  - Sanity check on sensor inputs
  - Check for proper authorization





Extends

- Use when a use-case optionally can do a little bit more:
  - Capture the normal behaviour
  - Capture the extra behaviour in a separate use-case
  - Create extends dependency
- Makes it a lot easier to understand



Class Diagrams

- A ***class diagram*** describes the ***static structure*** of a system.
- It ***shows how a system is structured*** rather than *how it behaves.*
- The static structure of a system consists of a number of class diagrams and their *dependencies*.
- The main constituents of a class diagram are ***classes and their relationships*** : generalization, aggregation, association and various kinds of dependencies.
- Concepts necessary to understand the UML syntax for representation of the classes and their relationships :

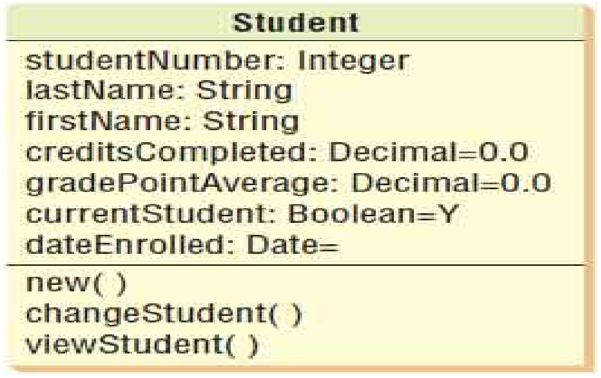
  * Classes * Attributes * Operation * Association
  * Aggregation * Composition * Inheritance * Dependency
  * Constraints * Object diagrams

- Object-oriented methodologies work to discover ***classes, attributes, methods, and relationships among classes***. Because programming occurs at the class level, defining classes is one of the most important object-oriented analysis tasks. ***Class diagrams*** show the ***static features*** of the system and do not represent any particular processing. A class diagram also shows the ***nature of the relationships between classes***.

- **A class** is represented by a ***rectangle*** in a class diagram.
- In the simplest format, the ***rectangle*** may include only the ***class name***, but may also include the ***attributes*** and ***methods***. Attributes are what the class knows about characteristics of the objects, and methods (also called operations) are what the class knows about *how to do things*. Methods are small sections of code that work with the attributes.

Example : Representation of a Student Class

An extended Student class that shows the type of data, and in some cases, its initial value or default value.

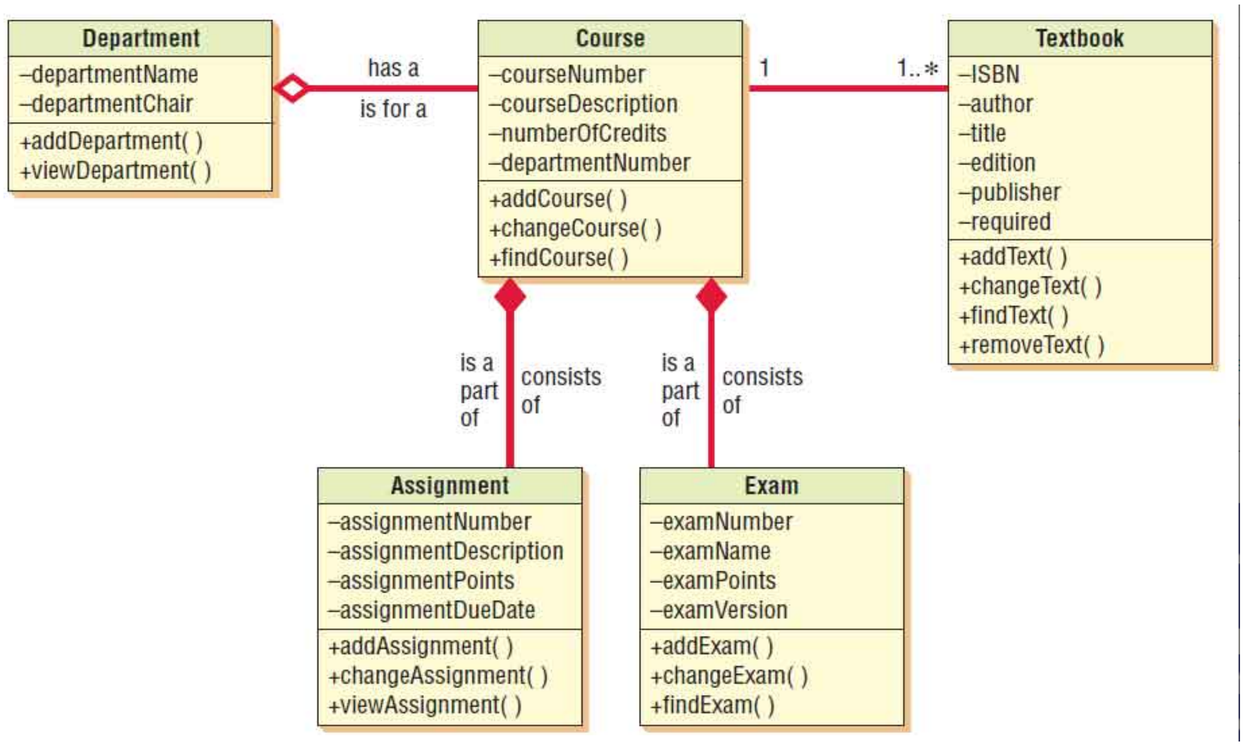


<u>Class Diagrams</u>

- A class in the diagram may show just the ***class name***; or the ***class name and attributes***; or the ***class name, attributes, and methods***. Showing only the class name is useful when the diagram is very complex and includes many classes. If the diagram is simpler, attributes and methods may be included.
- When attributes are included, there are *three ways* to show the ***attribute information***. The simplest is to include only the ***attribute name***, which takes the least amount of space.
- The ***type of data*** (such as string, double, integer, or date) may also be included on the class diagram. The most complete descriptions would include an equal sign (=) after the type of data followed by the ***initial value for the attribute.*** If the attribute must take on one of a finite number of values, such as a student type with values of F for full-time, P for part-time, and N for non-matriculating, these may be included in curly brackets separated by commas: **studentType:char{F,P,N}**.

- The <u>***attributes***</u> (or properties) are usually designated as <u>***private***</u>, or only available in the object. This is represented on a class diagram by a <u>***minus sign***</u> in front of the attribute name.
- On a class diagram, <u>***public***</u> messages (and any <u>***public attributes***</u>) are shown with a <u>***plus sign (+)***</u> in front of them.
- Attributes may also be ***protected***, indicated with a different symbol. These attributes are hidden from all classes except immediate subclasses. Under rare circumstances, an attribute is public, meaning that it is visible to other objects outside its class.
- Making attributes private means that the attributes are available to outside objects through the class methods only, a technique called ***encapsulation***, or ***information hiding***.

A class diagram for course offerings.
The filled-in diamonds show aggregation
and the empty diamond shows a whole-part relationship.



- The figure illustrates a ***class diagram for course offerings***. Notice that the ***name is centered at the top of the class, usually in boldface type***. The area directly below the name shows the ***attributes***, and the ***bottom portion lists the methods***. The class diagram shows data storage requirements as well as processing requirements.
- Meaning of the diamond symbols
- UML syntax for representation of the classes and their relationships.

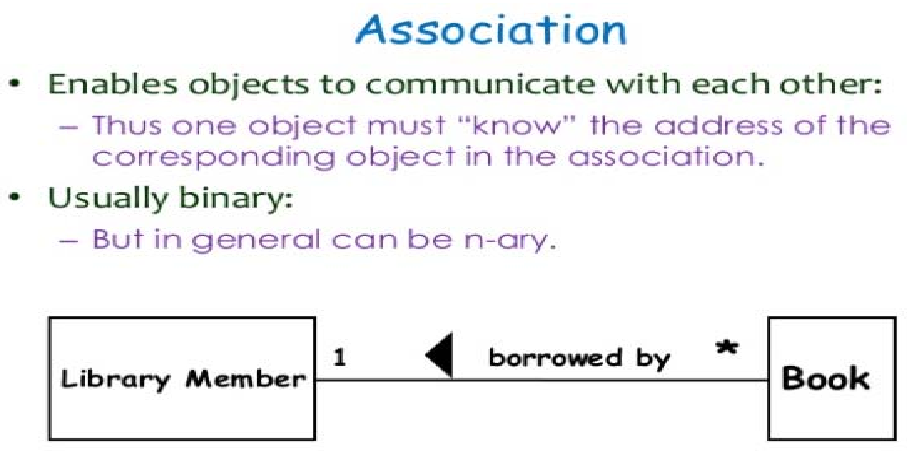
Classes

- The ***classes*** represent entities with common features, i.e. attributes and operations.
- ***Classes*** are represented as ***solid outline rectangles*** with ***compartments***.
- Classes have a mandatory name compartment where the name is written centered in boldface.
- The ***class name*** is usually written using mixed case convention and ***begins with an uppercase***.
- The class names are usually chosen to be singular nouns.

Association

- Associations are needed to enable objects to communicate with each other.
- An ***association*** describes a ***connection between classes***.
- The relation between two objects is called object connection or link.
- For example, suppose Mahesh has ***borrowed*** the book Compiler Design. Here, borrowed is the ***connection*** between the objects Mahesh and the Compiler Design book.
- ***Association between two classes*** is represented by drawing a ***straight line*** between the ***concerned classes***. A graphical representation of the association relation is illustrated in the figure.

- The name of the association is written on the association line.
- An ***arrowhead*** may be placed on the association line to indicate the reading direction of the association.
- On each side of the association relation, the ***multiplicity*** is noted as ***an individual number*** or as a ***value range***.
- The ***multiplicity*** indicates ***how many instances of one class are associated with the other***.
- ***Value ranges*** of multiplicity are noted by specifying the minimum and maximum value, separated by two dots. For example, 1..8.
- An ***asterisk*** is a ***wild card*** and means ***many (zero or more)***.



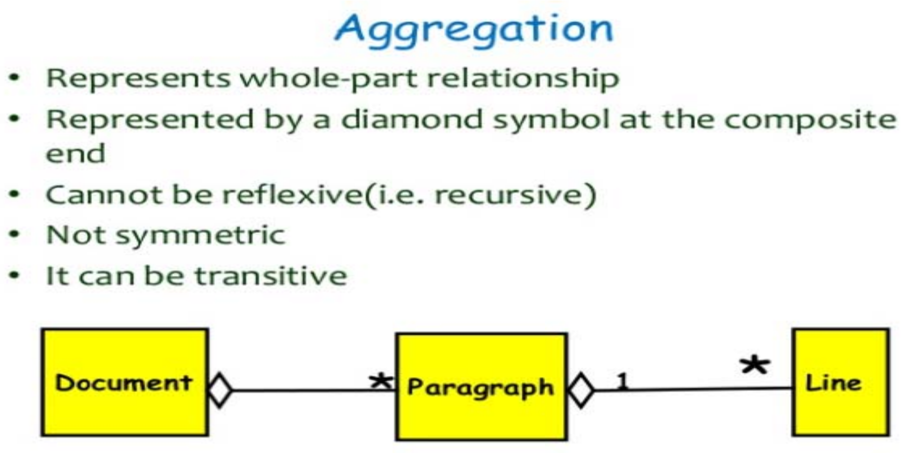
Association

- Enables objects to communicate with each other:
  - Thus one object must "know" the address of the corresponding object in the association.
- Usually binary:
  - But in general can be n-ary.

Example : "Many books may be borrowed by a Library Member."

Aggregation

- Aggregation is a special type of association where the involved classes represent a ***whole-part relationship***.
- The aggregate takes the responsibility of forwarding messages to the appropriate parts.
- If an instance of one class contains instances of some other classes, then aggregation (or composition) relationship exists between the ***composite object*** and the ***component object***.
- ***Aggregation is represented by the diamond symbol at the composite end of a relationship.***
- The number of instances of the component class aggregated can also be shown.
- The aggregation relationship ***cannot be reflexive (i.e. recursive)***. That is, ***an object cannot contain objects of its own class.***
- The aggregation relation is ***not symmetric***. ***That is, two classes A and B cannot contain instances of each other.***
- However, the aggregation relationship ***can be transitive***.

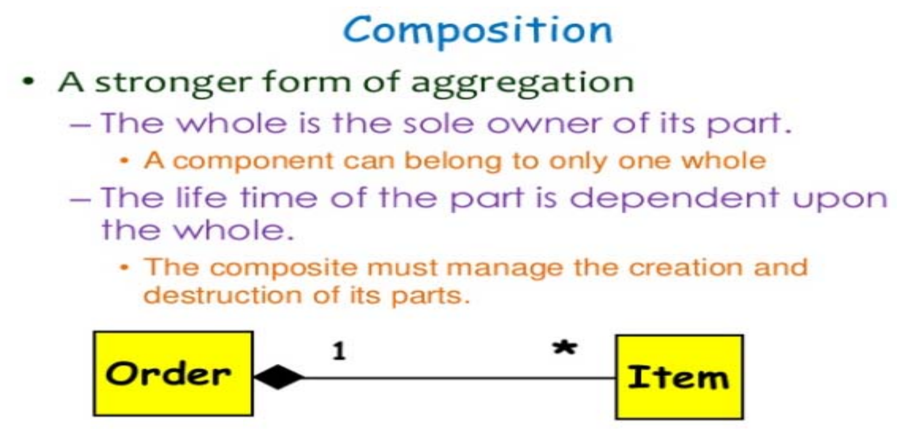


Aggregation

- Represents whole-part relationship
- Represented by a diamond symbol at the composite end
- Cannot be reflexive(i.e. recursive)
- Not symmetric
- It can be transitive

Composition

- ***Composition*** is a stricter form of aggregation, in which ***the parts are existence-dependent on the whole***.
- The life of each part is closely tied to the life of the whole.
- ***When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed.***
- The ***composition relationship*** is represented as ***a filled diamond drawn at the composite end***.
- Example : an invoice object with invoice items.
  As soon as the invoice object is created, all the invoice items in it are created and as soon as the invoice object is destroyed, all invoice items in it are also destroyed.



Composition

- A stronger form of aggregation
  - The whole is the sole owner of its part.
    - A component can belong to only one whole
  - The life time of the part is dependent upon the whole.
    - The composite must manage the creation and destruction of its parts.

Order
1
*
Item

Dependency

- Dependency is a form of association between two classes.
- A ***dependency relation between two classes*** shows that ***a change in the independent class requires a change to be made to the dependent class***.
- ***Dependencies*** are shown as ***dotted arrows***.
- Dependencies may have various causes.
- Important reasons for dependency among classes :

  * A class invokes the methods provided by another class.

  * A class uses a specific interface of another class. If the properties of the class that provides the interface are changed, then a change becomes necessary in the class that uses that interface.

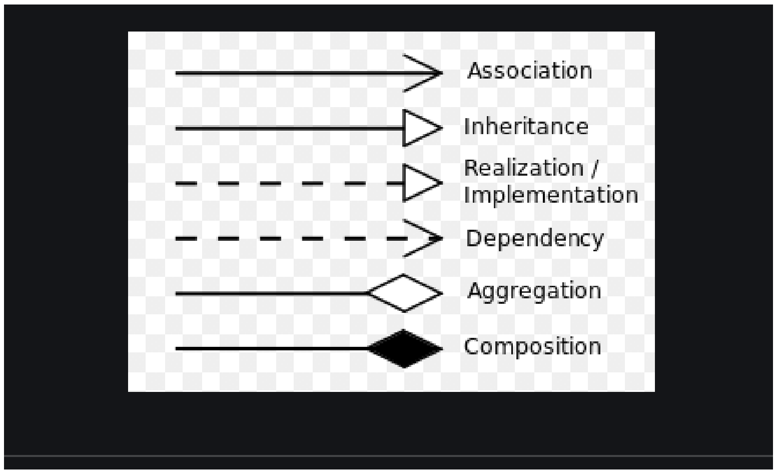
Constraints

- ***Constraints*** describe ***a condition*** or ***an integrity rule***.
- It can describe the ***permissible set of values of an attribute***, specify ***pre- and post-conditions for operations***, define a ***specific order***, etc.
- For example, **{Constraint}** UML allows us to use any free form expression to describe constraints.
- ***Enclosed between braces.***
- Constraints ***can be expressed*** using ***informal English***.
- UML also provides ***Object Constraint Language (OCL)*** to specify constraints. [Rumbaugh 1999]

Inheritance

- The ***inheritance relationship*** is represented by means of an empty arrow pointing from the subclass to the superclass.
- ***The arrow is directly drawn from the subclass to the superclass.***
- Alternatively, the inheritance arrows from the subclasses may be combined into a single line.

Class Diagrams



Interaction Diagrams

- ***Interaction diagrams*** are models that ***describe how groups of objects collaborate to realize some behavior***.
- Typically, each interaction diagram realizes the behavior of a single use case.
- An interaction diagram ***shows*** a ***number of example objects*** and ***messages*** that are *passed between the objects* ***within the use case.***
- There are two kinds of interaction diagrams :

  * ***sequence diagrams***

  * ***collaboration diagrams***
- One diagram can be derived automatically from the other.
- They ***portray different perspectives of behavior of the system***.

Sequence Diagrams

- A sequence diagram ***shows interaction among objects*** as a two-dimensional chart.
- The chart is **read from top to bottom**.
- The ***objects participating in the interaction*** are ***shown at the top of the chart as boxes attached to a vertical dashed line***.
- ***Inside the box*** the ***name of the object*** is written with a ***colon*** separating it from the ***name of the class***. Both the ***name of the object and the class are underlined***.
- The objects appearing at the top signify that the objects already existed when the use case execution was initiated.
- However, if some object is created during the execution of the use case, and participates in the interaction (e.g. a method call), then that object should be shown at an appropriate place in the diagram where it was created.

- The ***vertical dashed line*** is called the ***object's lifeline***.
- The lifeline indicates the existence of the object at any particular point of time.
- The rectangle drawn on the lifeline is called the activation symbol and indicates that the object is active as long as the rectangle exists.
- Each message is indicated as an arrow between the lifelines of two objects.
- The messages are shown in chronological order from the top to the bottom.
- Reading the diagram from the top to the bottom shows the sequence in which the messages occur.
- Each message is labeled with a message name. Some control information can also be included.
- Each message is labeled with a message name. Some control information can also be included.
- Two types of control information are particularly valuable.

* A condition (e.g. [invalid]) indicates that a message is sent, only if the condition is true.
* An iteration marker (*) shows that a message is sent many times to multiple receiver objects as would happen when you are iterating over a collection or the elements of an array. You can also indicate the basis of the iteration, for example, [for every book object].

- **Unified Modeling Language (UML)** is a modeling language in the field of software engineering which aims to set standard ways to visualize the design of a system. UML guides the creation of multiple types of diagrams such as interaction , structure and behavior diagrams.
- A **sequence diagram** is the most commonly used interaction diagram.
- **Interaction diagram**

An interaction diagram is used to show the **interactive behavior** of a system. Since visualizing the interactions in a system can be a cumbersome task, we use different types of interaction diagrams to capture various features and aspects of interaction in a system.

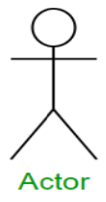
Sequence Diagrams

- A sequence diagram simply <u>depicts interaction between objects in a sequential order i.e. the order in which these interactions take place.</u> We can also use the terms event diagrams or event scenarios to refer to a sequence diagram.
- Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.

<u>Sequence Diagram Notations</u>

- **Actors :** An actor in a UML diagram represents a ***<u>type of role</u>*** where it interacts with the system and its objects. It is important to note here that an actor is always outside the scope of the system we aim to model using the UML diagram.

We use actors to depict various roles including human users and other external subjects. We represent an actor in a UML diagram using a stick person notation. We can have multiple actors in a sequence diagram



- **Lifelines –** A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a sequence diagram is represented by a lifeline.
- Lifeline elements are located at the top in a sequence diagram.
- The standard in UML for naming a lifeline follows the following format – Instance Name : Class Name.
- We display a lifeline in a rectangle called head with its name and type. The head is located on top of a vertical dashed line (referred to as the stem) as shown here. If we want to model an unnamed instance, we follow the same pattern except now the portion of lifeline's name is left blank.

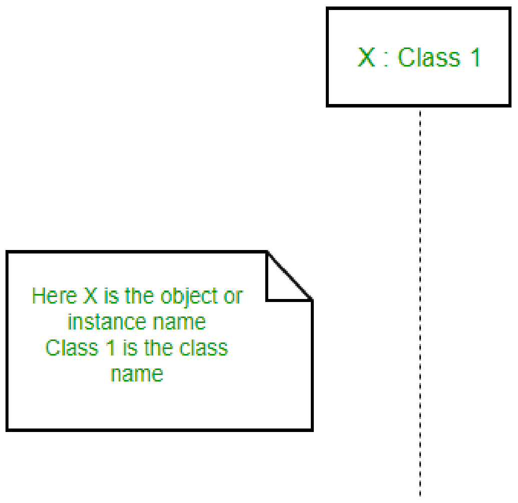

X : Class 1
Here X is the object or instance name Class 1 is the class name

Figure – lifeline

- **Messages –** Communication between objects is depicted using messages.
- The messages appear in a sequential order on the lifeline.
- We represent messages using arrows. Lifelines and messages form the core of a sequence diagram.
- Messages can be classified into various **categories** :

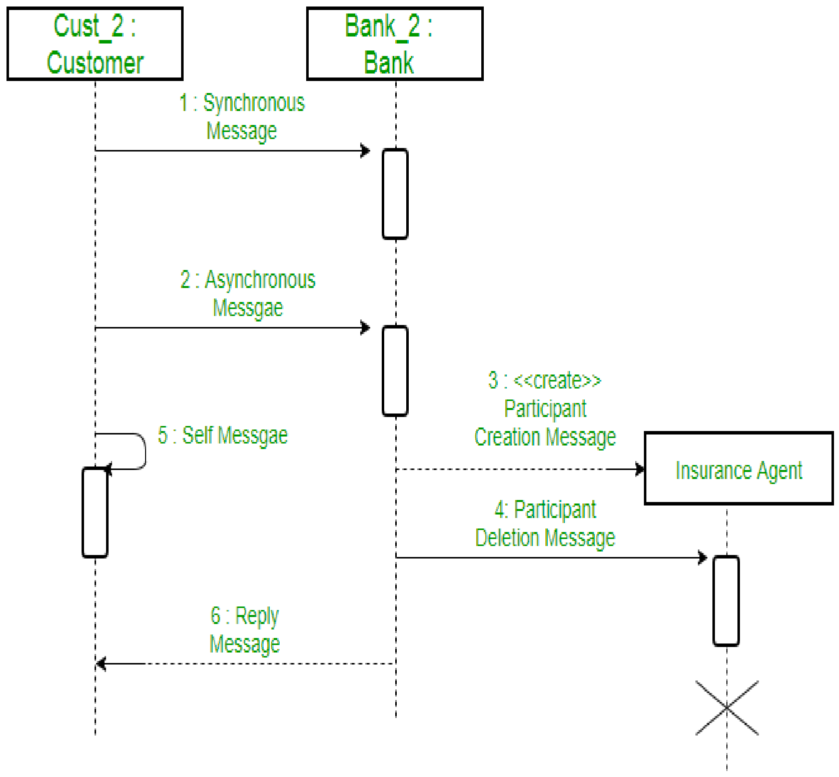


Figure – a sequence diagram with different types of messages

(***Source :https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams***/)

Synchronous messages –

- ***A synchronous message waits for a reply before the interaction can move forward***. The sender waits until the receiver has completed the processing of the message.
- The caller continues only when it knows that the receiver has processed the previous message i.e. it receives a reply message.
- A large number of calls in object oriented programming are synchronous.
- We use a ***solid arrow head*** to represent a synchronous message.

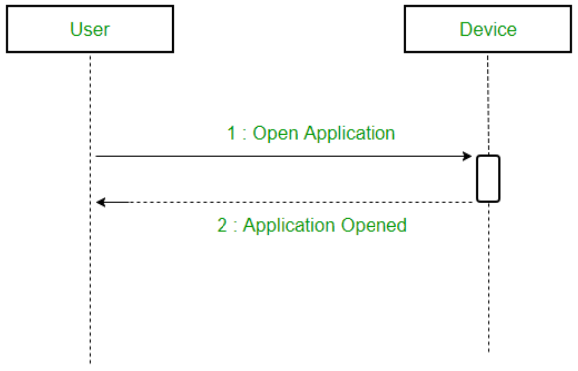
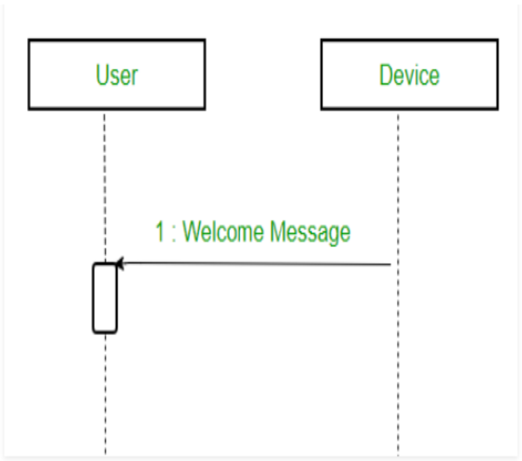


Figure – a sequence diagram using a synchronous message

Asynchronous Messages :

- An asynchronous message does not wait for a reply from the receiver.
- The interaction moves forward irrespective of the receiver processing the previous message or not.
- We use a ***lined arrow head*** to represent an asynchronous message.



Create message – We use a Create message to instantiate a new object in the sequence diagram. There are situations when a particular message call requires the creation of an object.

- It is represented with a ***dotted arrow*** and create word labeled on it to specify that it is the create Message symbol.
- For example – The creation of a new order on an e-commerce website would require a new object of Order class to be created.

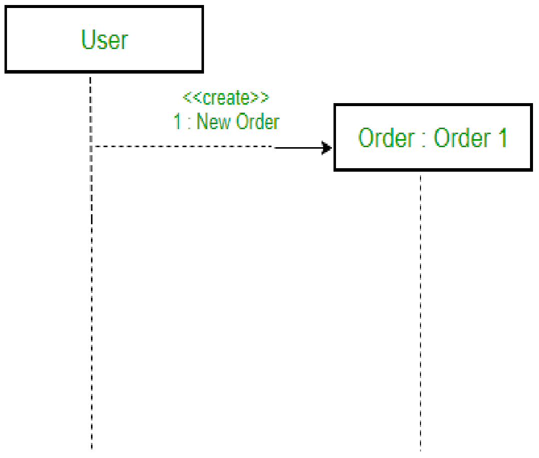


Figure - a situation where create message is used

(***Source :https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams***/)

Delete Message :

- We use a Delete Message to delete an object. When an object is deallocated memory or is destroyed within the system we use the Delete Message symbol.
- It destroys the occurrence of the object in the system.
- It is represented by an arrow terminating with a x.
- For example – In the scenario shown here when the order is received by the user, the object of order class can be destroyed.

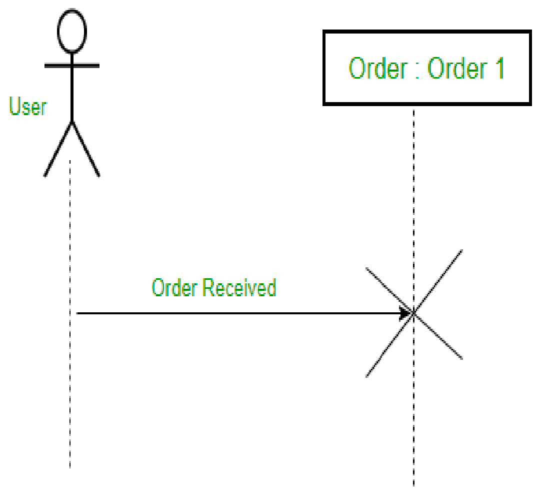


Figure – a scenario where delete message is used

(***Source :https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/***)

Self Message :

- Certain scenarios might arise where the object needs to send a message to itself. Such messages are called Self Messages and are represented with a U shaped arrow.

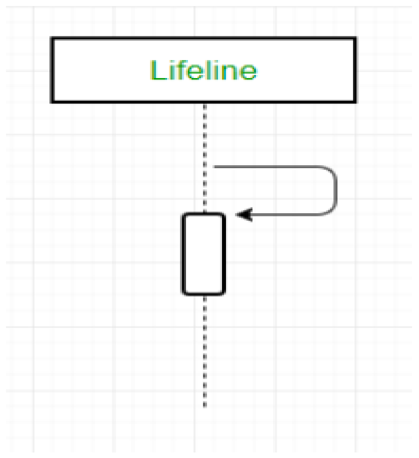


Figure – self message

Reply Message :

- Reply messages are used to show the message being sent from the receiver to the sender.
- We represent a return/reply message using an open arrowhead with a dotted line.
- The interaction moves forward only when a reply message is sent by the receiver.

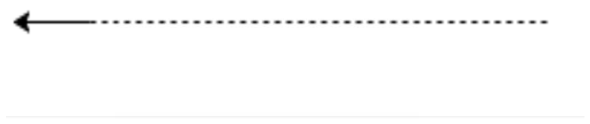


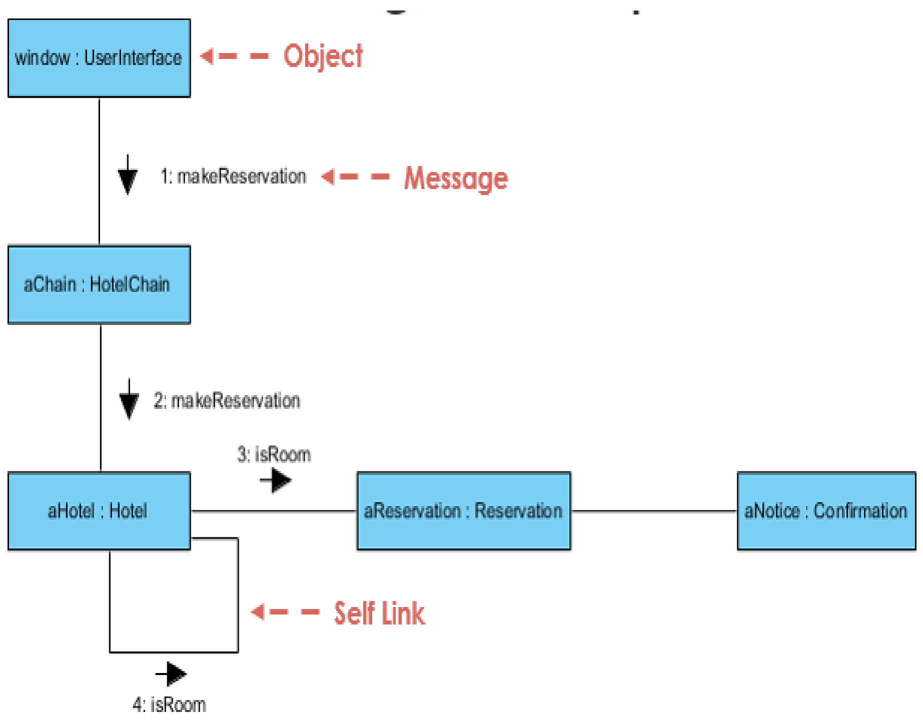
Figure – reply message

A sequence diagram with different types of messages
(***Source :https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams***/)

<u>Collaboration Diagrams</u>

- ***A <u>collaboration diagram</u> shows both the <u>structural and behavioral aspects</u> explicitly***. This is unlike a sequence diagram which shows only the behavioral aspects.
- The ***structural aspect*** of a collaboration diagram consists of ***objects*** and the ***links*** existing between them.
- In this diagram, an ***object*** is also called a ***collaborator***. The ***behavioral aspect*** is ***described by the set of messages exchanged among the different collaborators***.
- The ***<u>link</u>*** between objects is shown as a ***<u>solid line</u>*** and can be used to send messages between two objects.
- The ***<u>message</u>*** is shown as a ***<u>labeled arrow placed near the link</u>***. Messages are ***prefixed with sequence numbers*** because that is the only way to describe the relative sequencing of the messages in this diagram.
- It helps us in ***determining which classes are associated with which other classes.***

An Example of a **Collaboration Diagram :**



Activity Diagrams

- The activity diagram is possibly one modeling element which was not present in any of the predecessors of UML. No such diagrams were present either in the works of Booch, Jacobson, or Rumbaugh.
- The activity diagram focuses on ***representing activities or chunks of processing*** which may or may not correspond to the methods of classes.
- An ***activity*** is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity has more than one outgoing transitions, then these must be identified through conditions.

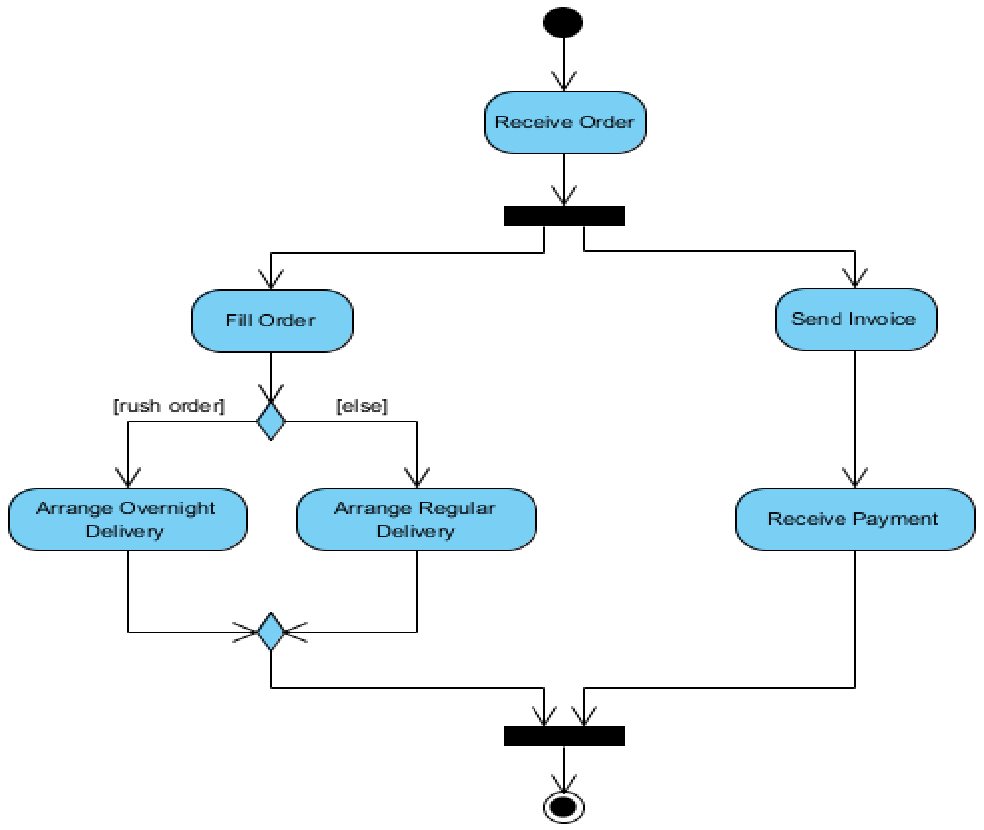
- Activity diagrams are similar to the procedural flow charts. The difference is that ***activity diagrams support description of parallel activities and synchronization aspects involved in different activities.***
- An interesting feature of the activity diagrams is the ***swim lanes***. Swim lanes ***enable you to group activities based on who is performing them***, for example, academic department vs. hostel office. Thus swim lanes ***subdivide activities based on the responsibilities of some components***. The activities in swim lanes can be assigned to some model elements, for example, classes or some component, etc.

- Activity diagrams can be ***very useful to understand complex processing activities involving many components.***
- ***Activity diagrams*** are normally employed in ***business process modeling***. This is carried out during the initial stages of requirements analysis and specification.
- Later these diagrams can be used to develop interaction diagrams which help to allocate activities (responsibilities) to classes.

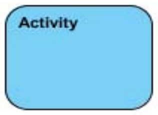
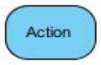
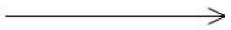
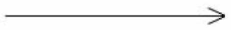
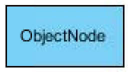
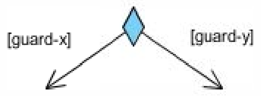
Example : Activity Diagrams

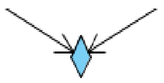
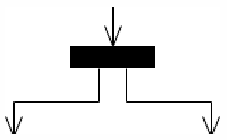
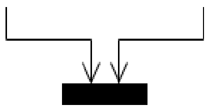
Process Order - Problem Description

- Once the order is received, the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing.
- On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed.
- Finally the parallel activities combine to close the order.



Activity Diagram Notation Summary

Notation Description	UML Notation
Activity Is used to represent a set of actions	
Action A task to be performed	
Control Flow Shows the sequence of execution	
Object Flow Show the flow of an object from one activity (or action) to another activity (or action).	
Initial Node Portrays the beginning of a set of actions or activities	
Activity Final Node Stop all control flows and object flows in an activity (or action)	
Object Node Represent an object that is connected to a set of Object Flows	
Decision Node Represent a test condition to ensure that the control flow or object flow only goes down one path	

Merge Node Bring back together different decision paths that were created using a decision-node.	
Fork Node Split behavior into a set of parallel or concurrent flows of activities (or actions)	
Join Node Bring back together a set of parallel or concurrent flows of activities (or actions).	

State Chart Diagrams

- A state chart diagram is normally used ***to model how the state of an object changes in its lifetime***.
- State chart diagrams are good at ***describing how the behavior of an object changes across several use case executions***.
- However, if we are interested in modeling some behavior that involves several objects collaborating with each other, the state chart diagram is not appropriate.
- State chart diagrams are ***based on the finite state machine (FSM) formalism***.
- An FSM consists of a ***finite number of states*** corresponding to those of the object being modeled. The **object undergoes state changes when specific events occur**. The FSM formalism existed long before the object-oriented technology, and has since been used for a wide variety of applications. Apart from modeling, it has even been used in theoretical computer science as a generator for regular languages.
- A state chart is a hierarchical model of a system and introduces the concept of a composite state (also called a nested state).
- Actions are associated with transitions and are considered to be processes that occur quickly and are not interruptible.
- Activities are associated with states and can take longer.
- An activity can be interrupted by an event.

The basic elements of the state chart diagram are as follows :

- ***Initial state*** : It is represented as a ***filled circle***.
- ***Final state*** : It is represented by a ***filled circle inside a larger circle***.
- ***State*** : It is represented by a ***rectangle with rounded corners***.
- ***Transition*** : A transition is shown as

an ***arrow between two states***. Normally, the ***name of the event*** which causes the transition is placed along side the arrow. You can also assign a ***guard*** to the transition. A guard is a ***Boolean logic condition***. The transition can take place only if the guard evaluates to true. The ***syntax for the label of the transition*** is shown in three parts :

event[guard]/action

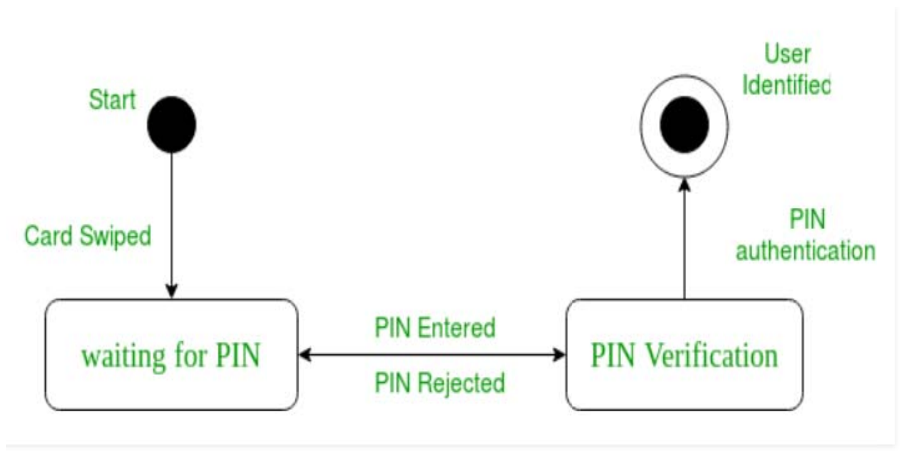


Figure – a state diagram for user verification

Coding

- Goal : To implement the design in the best possible manner.
- Coding activity : affects both testing and maintenance profoundly.
- Time spent in coding : a small percentage of the total software cost, while testing and maintenance consume the major percentage.
- Goal during coding : should not be to reduce the implementation cost, but the ***goal should be to reduce the cost of later phases***. Goal during the coding phase is not to simplify the job of the programmer. Rather, the ***goal should be to simplify the job of the tester and the maintainer***.
- During coding, the programs should not be constructed so that they are easy to write, but so that they are ***easy to read and understand***.

- Criteria for judging a ***program*** :

  - readability and understandability
  - size of the program
  - execution time
  - required memory

Programming Principles and Guidelines

- How to write a high quality code ?
- How to avoid errors ?
- Writing code quickly
- Good programming (producing correct and simple programs) is a practice independent of the target programming language.

Common Coding Errors

Much of effort in developing software goes in identifying and removing bugs. How to reduce occurrence of bugs ? Educate programmers about most common types of errors.

1.1 Memory leaks

* A situation where the memory is allocated to the program which is not freed subsequently.

* occurs frequently in languages which do not support automatic garbage collection (like C, C++)

1.2 Freeing an already freed resource

This error occurs when a programmer tries to free the already freed resource.

```
main()
{
    char  *str;
    str = (char *) malloc (10);
    if  (global == 0)
        free (str);
    free (str);  /* str is already freed */
}
```

1.3 NULL Dereferencing

* This error occurs when we try to access the contents of a location that points to NULL.
* An attempt to access uninitialized memory.
* A commonly occurring error. It may bring the software system down.
* Sometimes NULL dereferencing may occur only in some paths and only under certain situations. Often improper initialization in different paths leads to the NULL reference statement.
* It can also be caused because of aliases. For example, two variables refer to the same object, and one is freed and an attempt is made to dereference the second.

```
switch (i)

{

     case  0 :  s = OBJECT_1;  break;

 case  1 :  s = OBJECT_2;  break;

}

return (s);    /*  s  is  not  initialized for values other than 0 and 1*/
```

1.4 Lack of unique addresses

* Aliasing may create many problems.
* For example, in the string concatenation function, we expect source and destination addresses to be different.

```
strcat (src, dest)
```

In the above function, if src is aliased to dest, then we may get runtime error.

1.5 Synchronization error

* In a parallel program, where there are multiple threads possibly accessing some common resources, synchronization errors are possible.

* Different categories of synchronization errors

- **Deadlocks :** The threads in a deadlock wait for resources which are in turn locked by some other thread.
- **Race conditions :** occur when two threads try to access the same resource and the result of execution depends on the order of execution of the threads.
- **Inconsistent synchronization :** A situation in which there is a mix of locked and unlocked accesses to some shared variables.

1.6 Array Index Out of Bounds

* Array index values should not exceed their bounds and should not be negative.

1.7 Arithmetic Exceptions

* These include errors like ***divide by zero*** and ***floating point exceptions***.
* Getting unexpected results or termination of the program.

1.8 Off by One

* This is one of the most common errors which can be caused in many ways.
* For example, starting at 1 when we should start at 0 or vice versa, writing <= N instead of < N or vice versa, and so on.

1.9 Enumerated data types

* Overflow and underflow errors can easily occur while working with enumerated types. Care must be taken when assuming the values of enumerated data types.

Example :

```
typedef  enum  {A, B, C, D}  grade;
  void  f1 (grade  x)
  {
     int  i, j;
     i = GLOBAL_ARRAY[x-1];    /* Underflow possible */
     j = GLOBAL_ARRAY[x+1];    /* Overflow possible */
  }
```

1.10 Illegal use of & instead of &&

* This bug arises if non short circuit logic (like & or |) is used instead of short circuit logic (&& or ||). Non short circuit logic will evaluate both sides of the expression. But short circuit operator evaluates one side, and based on the result, it decides if it has to evaluate the other side or not.

Example : if (object != null & object.getTitle() != null)

/ Here second operation can cause a null dereference */*

1.11 String handling errors :

There are a number of ways in which string handling functions like strcpy, sprintf, gets etc. can fail.

Examples : one of the operands is NULL, the string is not NULL terminated,
the source operand may have greater size than destination, etc.

1.12 Buffer overflow :

* It is a frequent cause of software failures.
* It is a security flaw – can be exploited by a malicious user for executing arbitrary code.
* By giving a large input, a malicious user can overflow the buffer. The return address can get rewritten to whatever the malicious user has planned. So, when a function call ends, the control goes to a place planned by the malicious user, where typically there may be some malicious code to take control of the computer or do some harmful action.

Example :

```
void  mygets(char *str) {
           int  ch;
           while (ch=getchar()  != '\n'  &&  ch != '\0')
                  *(str++)=ch;
           *str='\0';
 }
 void main()  {  char  ss[5];   mygets(ss);   }
```