Software Engineering Class

University of Anbar

Computer Science Department

Hussein K. Almulla

# Table of Contents

# 1. Syllabus

## 1.1. Course objective

Software engineering is concerned with the development and maintenance of high-quality software in a systematic and efficient approach in regard of the cost and schedule of the development process. In this course, students will learn about the different software development lifecycle (SDLC) phases used in developing, delivering, and maintaining software products. Students will also acquire basic software development skills and understand common terminology used in the software engineering profession. Student will learn the major phases of SDLC: analysis (requirements), design, implementation, and testing; in addition to deployment. A long with that, student will learn some tool and technologies that used to simplify the process and increase the productivity. Also, student will gain knowledge about the Agile methodology that used to manage development process.

## 1.2. Learning outcomes:

1- Students will gain knowledge of the SDLC and how they can be applied on real project.
2- Students will be able to analyze user need to formulate user story and use cases that can be implemented.
3- Students will be able to build a basic design (Visual, Object Oriented, Database, System) base one the gathered requirements.
4- Students will be able to transform the requirement and design into code based on their previous knowledge.
5- Students will be able to write unite test to their code to test the functionality.
6- Students will understand the Agile way of managing projects.

The course includes the following major topics:

1- Introduction to software engineering
2- Software requirement
3- Software design
4- Software implementation
5- Software testing
6- Agile methodology

## 1.3. Course grading

The total grade of the course in 40 points which will be divided as follow:

1- Project (10 points)
2- Homework (10 points)
3- Quizzes (10 points)
4- Exam (10 points)

## 1.4. Project

Students have to deliver project of their own choice. The project can be written in any language, framework, library, or tool. The students will apply the steps of the SDLC on the project to gain practical experience of these concepts. Each three students (max) can collaborate on single project. The have to come up with an idea that they want to build. The idea cannot be change after being selected. Students are NOT allowed to copy or use other people projects.

## 1.5. Schedule

All the submission will be through Google Classroom

| Date | Submission |
|---|---|
| **3/3/2023** | Project idea |
| **10/3/2023** | Homework 1 |
| **17/3/2023** | Project analysis (gather requirements) |
| **24/3/2023** | Homework 2 |
| **31/3/2023** | Project design |
| **7/4/2023** | Homework 3 |
| **28/4/2023** | Project implementation |
| **5/5/2023** | Homework 4 |
| **12/5/2023** | Project Unit testing |

Table 1.1: Submission deadlines

## 1.6. Books

1- Rod Stephens, Beginning Software Engineering, Second Edition, 2023

2- Nico Loubser, Software Engineering for Absolute Beginners Your Guide to Creating Software Products, 2021
3- Ronald J. Leach, Introduction to Software Engineering Second Edition, 2016

# 2. Introduction to Software Engineering

It's easy to sit down at the keyboard and write a working program with no previous design or planning. Those types of programs are fine if you're the only one using them and then for only a short while. To produce applications that are effective, safe, and reliable, you can't just sit down and start typing. Because of a software is the combination of program(s), database(s), and documentation in a systemic suite, and with the sole purpose of solving specific system problems and meeting predetermined objectives that includes one or more team(s) to work on it, it need a plan, for this reason, software engineering is founded to formulate the way to develop a software.

Software engineering requires planning, and it goes through cycle similar to other engineering project. However, software engineering is knowledge base which depends highly on the understanding of the user needs which can change over time. For that, the software life cycle needs to be flexible to accommodate any changes. The flexibility granted to software by its virtual nature is both a good and a bad. It's a good because it lets you refine the program during development to better meet user needs, add new features to take advantage of opportunities discovered during implementation, and make modifications to meet evolving business requirements. This type of flexibility isn't possible in other types of engineering. It is bad because the flexibility that allows you to make changes throughout a software project's life cycle also lets you mess things up at any point during development. Adding a new feature can break existing code or turn a simple, elegant design into confusing parts. Constantly adding, removing, and modifying features during development can make it impossible for different parts of the system to work together. In some cases, it can even make it impossible to tell when the project is finished.

Producing a software application is relatively simple in concept: Take an idea and turn it into a useful program. Unfortunately for projects of any real scope, there are countless ways that a simple concept can go wrong. Software engineering includes techniques for avoiding the many pitfalls that otherwise might send your project down the road to failure. It ensures that the final application is effective, usable, and maintainable. It helps you meet milestones on schedule and produce a finished project on time and within budget. Perhaps most importantly, software engineering gives you the flexibility to make changes to meet unexpected demands without completely obliterating your schedule and budget constraints.

## 2.1. Software Development Life Cycle

Developing a software will go through the following phases.

1. Requirements Gathering

One of the first steps in a software project is figuring out the requirements. You need to find out what the customers want and what the customers need. Depending on how well-defined the user's needs are, this chore can be time-consuming.

2. Design

it includes such things as decisions about what platform to use (such as desktop, laptop, tablet, or phone), what data design to use (such as direct access, 2-tier, or 3-tier), and what interfaces with

other systems to use (such as external purchasing systems or a payroll system hosted in the cloud). It includes information about the project architecture at a relatively high level. You should break the project into the large chunks that handle the project's major areas of functionality. Also, it includes the database relational information, object-oriented design, and visual interfaces.

3. Development

At this point developers start using the requirements and designs to write the code that will provide the users with what they need. During this step, requirements and design are revisited and they can be refined and modified if needed.

4. Testing

During implementation phase, developers doing their best to build a product that work correctly. However, making a mistake in any software is inevitable which certainly lead to bugs. Mostly there are two way the bug can introduce; programmers assume their code is correct they don't always test it as thoroughly as they should, and different parts of code does not work together. To address these issues, testing phase is introduced in which testers who didn't write the code test it. After a piece of code seems to work properly, it is integrated into the rest of the project, and the whole thing is tested to see if the new code broke anything. Any time a test fails, the programmers go back into the code to find bug and fix it. After any repairs, the code goes back into the queue for retesting.

5. Deployment

It is the phase in which the product is ready to hand over to the users (customers). This phase could be as easy as installing .exe file or it can take few days to prepare resources and configure these resources.

6. Maintenance

After deploying a software, any found issue/bugs need to be fixed by and redeployed. This step can trigger a whole cycle (requirement, design, coding, testing, and deployment) or it can just fix small issue in the code (but still need to be retested).

## 2.2. Project Initiation

Before starting any project, there are some studies need to be made in order to decide the benefit of the project and its scop. There are three main points need to be made:

1- Project goal: This section aims to state why this project being execute, what the ultimate goal is or outcome of the final results.
2- Problem definition: This section defines the problem that the project should solve.
3- Target user (audiences): who will benefit from this project, who the users/customers are.

# 3. Software Requirement

The requirements for a system are the descriptions of the services that a system should provide and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. Requirements is usually presented as the first stage of the software development process. However, some understanding of the requirements may have to be presented before a decision is made to go ahead with the development of a system. Throughout development, the requirements used to guide development and ensure that progress heading in the right direction. At the end of the project, the requirements used to verify that the finished application does what it's supposed to do.

## 3.1. Requirement's Properties

The requirements should have and satisfied the following properties:

1- Clear/ unambiguous: good requirements are clear, concise, and easy to understand. If the requirement is worded so that you can't tell what it requires, then you can't build a system to satisfy it. Example: if you are building app for cyclist and the requirement state "the app will find the best path between start and destination location. Here the word "best" is not clear and ambiguous. It should state whether the best is shortest, fastest, path with nice view, etc.

2- Consistent: A project's requirements must be consistent with each other. That means that they not only cannot contradict each other but that they also don't provide so many constraints that the problem is unsolvable. Example: if you are building software the manage appointments for mechanical shop, the requirements state the following:

- Reduce the appointment starting window to two hours.
- Meeting 90% of the scheduled appointments.

These two requirements are unlikely to be satisfied because shortening the time before the start of the appointment mean more likely to miss this appointment.

3- Prioritized: working on the project's schedule requires prioritizing requirements because project could reach point that need to be stop. At this point, project should deliver the most important features first. If the customer insists that all requirements have same priorities, then they should be willing to increase the fund if needed. One of the prioritization methods is MoSCoW:

1) M for Must have: The features that must have because they are the fundamental and backbone of the software.
2) S for Should have: The features that are important that should be include (if possible) because they provide benefits to user. If there are no time/budget they can be postpone to next release.
3) C for Could have: The features that desirable (not necessary and not important). They will not be done until the M and C finished. These features are postponed until having time and budget.

4) W for Will not have: Features that optional and will note be done. It could be included in the future release.

4- Verifiable: Requirements must be verifiable. If you can't verify a requirement, you can't know if you met the requirement and you can't prove to your customers that you've done it. Being verifiable means the requirements must be limited and precisely defined. They can't be open-ended statements such as, "Process more work orders per hour than are currently being processed." How many work orders is "more"?

5- Avoid using the following words:
   a. Comparative - faster, better, more, and shinier.
   b. Imprecise adjectives - robust, user-friendly, efficient, flexible, and glorious.
   c. Vague commands - minimize, maximize, improve, and optimize.

6- Mix project goal with requirement: Avoid mixing project goal and hope with requirements. Example: customers state "increase the purchasing by 50%". This the goal of the project and it is the marketing responsibility, and it should not be part of the requirements.

## 3.2. Requirement Types

There are different types of requirements:

1- User requirements: describe how the project will be used by the eventual end users. They show the steps users will perform to accomplish specific tasks, use cases, and prototypes.
2- Hardware Requirements: Define the properties expected from and constraints on the physical hardware in a system. Expectations on storage space, CPU, network latency, etc.
3- Software Requirements: Properties of software that being executed and its constraints. This includes details about how software will functional.

These requirements can be categorized into Functional and non-functional.

### 3.2.1. Functional Requirement

These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. It may also include what the system should not do. These requirements depend on the type of software being developed, and the expected users of the software. Example:

- Each user must have unique system ID that consist of 8 digits.
- A user should be able to search the patient's appointments lists for all doctors in the clinic.
- A patient should be able to see the available appointments slots.

### 3.2.2. Non-functional requirements

These requirements that are not directory linked or concerned with provided services. However, this type of requirement can be considered highly important because it can impact the life and goal

of the system. Example: reliability, response time, security level, memory use, etc. requirements are often more critical than individual functional requirements. Failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation. Also, non-functional requirement can generate multiple functional requirements. Example: security requirement that led implement authentication and authorization as requirements. Non-functional requirements, in contrast of functional, not always raise due to user need. It can be required by organization policy, country regulation, interoperability with other system, etc.

## 3.3. Requirement gathering

Collecting the requirements involve meeting with stockholders of different levels. This includes meeting with manager to get their vision on how the system will help them making improvement in the business. Meeting with users to find how the system can help with their tasks. Also, meeting with them to collect how the system will be used. In general, there are techniques that can be used to discover and collect requirements and process:

- Interview: meet with stockholders and ask them question related to their jobs, tasks, difficulties, hope, etc. These can provide good information about how the system should be. Interview can be closed (there are predefined questions) or opened (not predefined agenda and it is open discussion)
- Observation: Watching people doing their jobs to learn the sequence and artifacts. This can help when people can't really explain or provide details about their tasks because it became a habit, they do not understand how their work relate to others, etc.
- Stories and scenarios: sometime people find it easy to tell a story about their work. This can be helpful to provide good aspect of sequence of the task and their impact. It also can give information about to handle specific/difficult situation.
- Questionary: It can be helpful get users opinion of a requirement or a situation.
- Brainstorming: in which group of people sitting together sharing their ideas as it come to them discuss it to get it organized.

## 3.4. Requirement specification

It is a comprehensive technical that descript how the requirement will be states. The set of specifications is fully described what the software will do and when, what, and how it will be expected to perform.

Example: requirement state that "user must be authenticated to user the software".

Specifications:

- The user needs to create account with username and password.
- The length of password must be at least 8.
- Password must include number, letter, and symbol.
- Password should be hashed before being stored.

The specification should be structured and written in natural language that is clear and precise. It is better to use standard format to when gathering requirements. Example of requirement specification template:

- **ID:** Unique identifier
- **Description:** A description of the requirement or entity being specified.
- **Rationale:** Context that explains why this requirement was included.
- **Inputs:** describe the input that must be provided if this is a functional requirement.
- **Outputs:** describe the output that must be provided if this is a functional requirement.
- **Persistent Changes:** describe any changes to system state that will persist following this operation.
- **Related Requirements or Prerequisite:** Refer to the IDs of any related requirements or use cases, or any required information.
- **Test Cases (verification):** Tests that can be used to show that this requirement is met.

Figure 3.1: Requirements information template

- **ID:** 23
- **Description:** user need to login with his/her username and password to be able to use the system. User who has no login credential must not enter to the system.
- **Rationale:** It is important to prevent unauthorized users from using the system.
- **Inputs:** username: include any letter or number but must not include special character, with length of at least 4. Password must consist of digit, letter, and symbol mix, with length of at least 8.
- **Outputs:** if user login successfully, welcome message should appear and move to landing page; otherwise, error message will be displayed.
- **Persistent Changes:** user information should be store in session. User should see option based on its permission.
- **Related Requirements or Prerequisite:** requirement ID 22, user must have created account.
- **Test Cases (verification):** use correct user credential, user incorrect password to make sure unauthorize user no able to login.

Figure 3.2: Example of requirement information collected for login.

## 3.5. Interaction Models

All systems involve interaction of some kind. This can be user interaction, which involves user inputs and outputs; interaction between the software being developed and other systems in its environment; or interaction between the components of a software system. User interaction modeling is important as it helps to identify user requirements. Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability. This section discusses two related approaches to interaction modeling:

1- Use case modeling, which is mostly used to model interactions between a system and external agents (human users or other systems).
2- Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.

### 3.5.1. Use Cases

Use cases are a way of describing interactions between users and a system using a graphical model and structured text. Use cases are documented using a high-level use case diagram. The set of use cases represent all the possible interactions that will be described in the system requirements. Actors may be human or other systems. Each class of interaction is represented as a named ellipse. Lines link the actors with the interaction (figure 3.3).

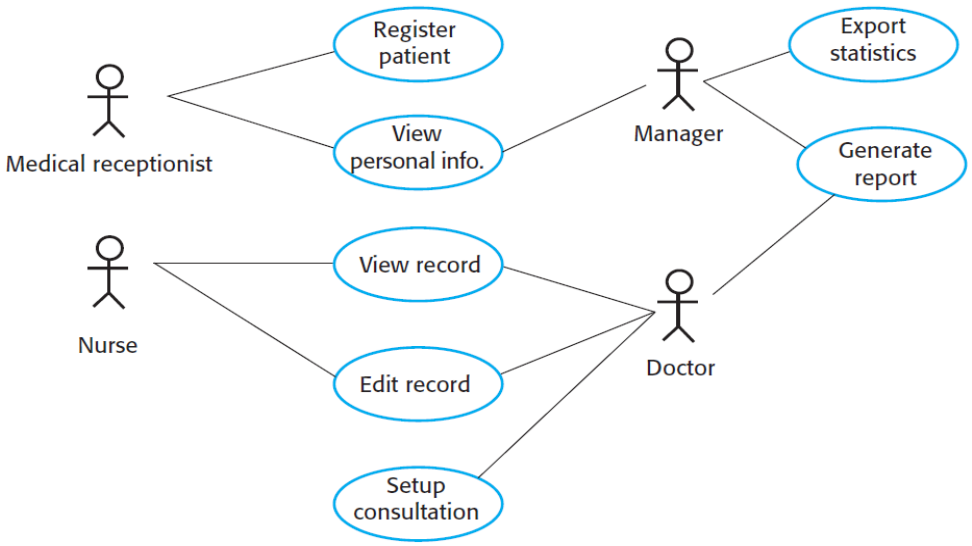



Figure 3.3: Use case example

Collecting use cases help the development team to capture the user's goal because it focuses on what user can achieve with system. To get and understand how the user will achieve these goals, team need to capture the sequences of user interactions which will describe the process. In use cases diagram, you can use relationships to describe the connection between different classes or actors (figure 3.4). The relationships as follow:

Association between actor and use case represented by line connecting actor to use case.

Generalization of an actor represented by arrow from on actor to another. This arrow means that the actor (arrow going from) can inherit the actor's role that arrow going to

Extend between two use cases represented by the text "<<**extend**>>" and it mean that the extending use case is depending on the extended use case (which is the base). This relationship can be base condition which make it optional.

Include between two use cases represented by text "<<**include**>>" which show that the included use case is part of the including use case (which is the base). The goal of include relationship is to reuse common class among different use cases.

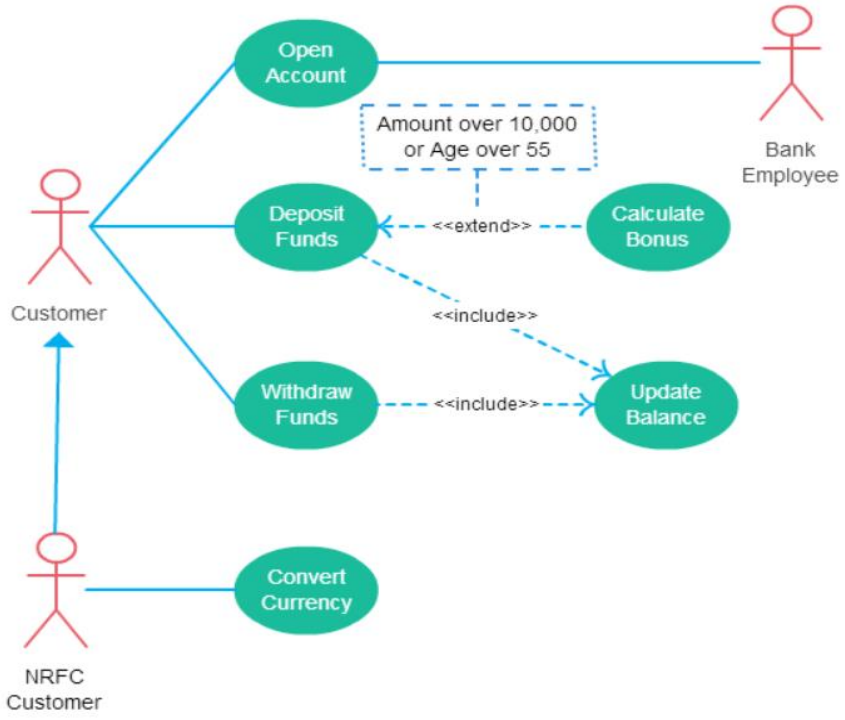



Figure 3.4: Example of use case with four types of relationships

### 3.5.2. Sequence Diagram

It is diagram that model the interaction between actors and objects and among objects themselves in the system. a sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance. The following example show the interaction of viewing patient information in the health system. The following is example of sequence diagram for ordering food in restaurant.

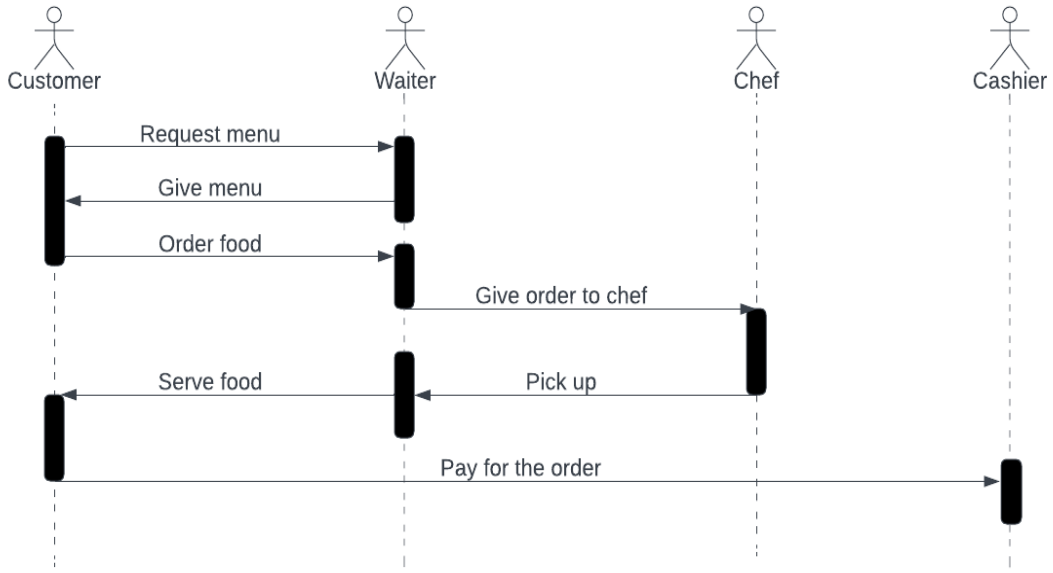



Figure 3.5: Example of sequence diagram of ordering food in restaurant

In the figure 3.6, example of the two features in patient record system. In these features, user try to transfer information from the database to patient system. The diagram shows the communication between actors and system objects. The sequence is as following:

1- User logs in to the system.
2- After login successfully two options are available (represent with “alt”). These options are:
    a. transfer of updated patient information from the database to the PRS and
    b. the transfer of summary health data from the database to the PRS.
3- In each option user credentials are check using PRS authorization.
4- Patient information is transfer to PRS after checking user and return Ok message.
5- User logout.

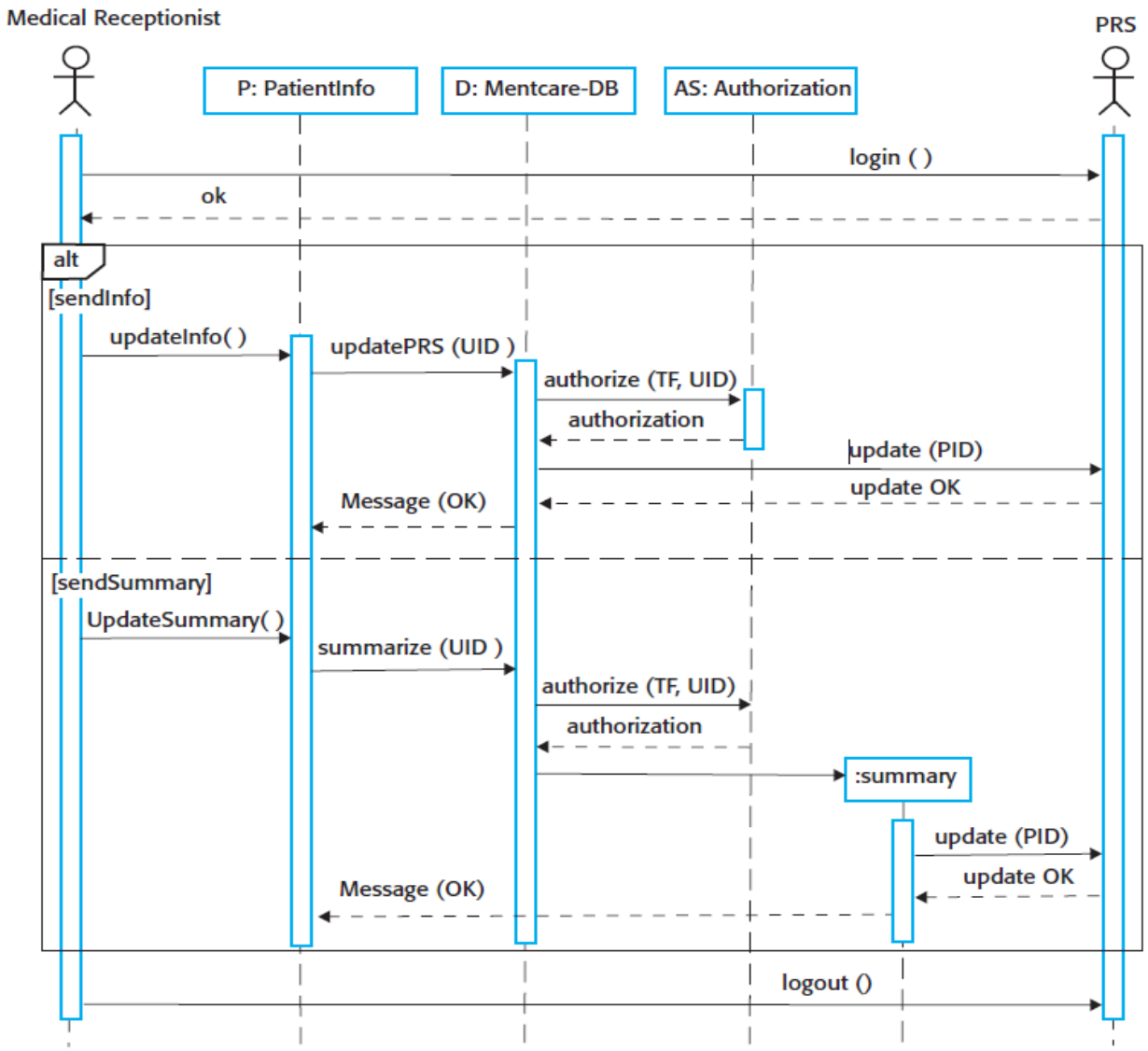



Figure 3.6: Example of sequence diagram of two feature in Clinic system

# 4. Software Design

Software design is the process of envisioning and defining software solutions. The objective of the software design process is the construction of a model that is representative of the required software system. The model must be accurate and comprehensive; it must conform to established software design and development standards. During this phase, there are many activities that need to be accomplished. The following are some of them:

1- **Architectural Design:** Architectural design relates to the subsystems and/or modules making up the system, as well as their interrelationships.
2- **Integration Design**: For each component (subsystem or module), its interface with other related components is designed and documented. This process is particularly useful in the scenario where the software system or component in being integrated into a larger software/information infrastructure.
3- **Object Structure Design:** Object structure design relates to the data structures used in the system—object types (information entities), relationships, integrity constraints, data dictionary, etc.
4- **User Interface Design:** relates to screen design, menu structure(s), input and output design and in some systems, a user interface language.
5- **Operations Design**
6- **Message Design**
7- **Security Design**

## 1.1. Architecture design

Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system. Architectural design is the first stage in the software design process which development team need to identify the main architectural components as these reflect the high-level features of the system. This phase is must do before starting major software development to understand how the parts of the software will work with each other's. The following figure (4.1) is example of architecture design of packing robot system design.

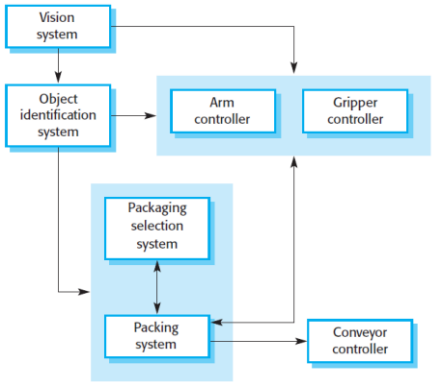



Figure 4.1: Example of architecture design

Architecture is important because it affects the performance, distributability, and maintainability of a system. In general, it has three main advantages:

1- Stakeholder communication
2- System analysis
3- Large-scale reuse

### 1.1.1. Architectural Design Patterns

The goal of patterns as a way of presenting, sharing, and reusing information about software under development. You can think of an Architectural pattern as a stylized, abstract description of good practice, which has been tried and tested in different systems and environments. So, an Architectural pattern should describe a system organization that has been successful in previous systems. There are several types of patterns that can be used, the following some of them:

### 1.1.2. Layered

The notions of separation and independence are fundamental to architectural design because they allow changes to be localized. Separating elements of a system allows these elements to change independently. The architecture is also changeable and portable. If its interface is unchanged, a new layer with extended functionality can replace an existing layer without changing other parts of the system (figure 4.2.). It usually consists of four layers:

1- **Presentation layer** (**UI**)
2- **Application layer** (**service**)
3- **Business logic layer**
4- **Data access layer**

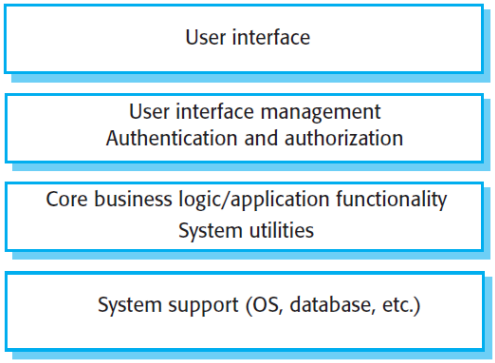



Figure 4.2: A generic layered architecture

### 1.1.3. Client-Server

This pattern is commonly used runtime organization for distributed systems. This pattern consists of set of services.

and associated servers, and clients that access and use the services. The server component will provide services to multiple client components. Clients request services from the server and the server provides relevant services to those clients.

### 1.1.4. Model-view-controller (MVC) pattern

This pattern is the basis of interaction management in many web-based systems and is supported by most language frameworks. It consists of three parts:

1- **Model** — contains the core functionality and data.
2- **View** — displays the information to the user (more than one view may be defined)
3- **Controller** — handles the input from the user.

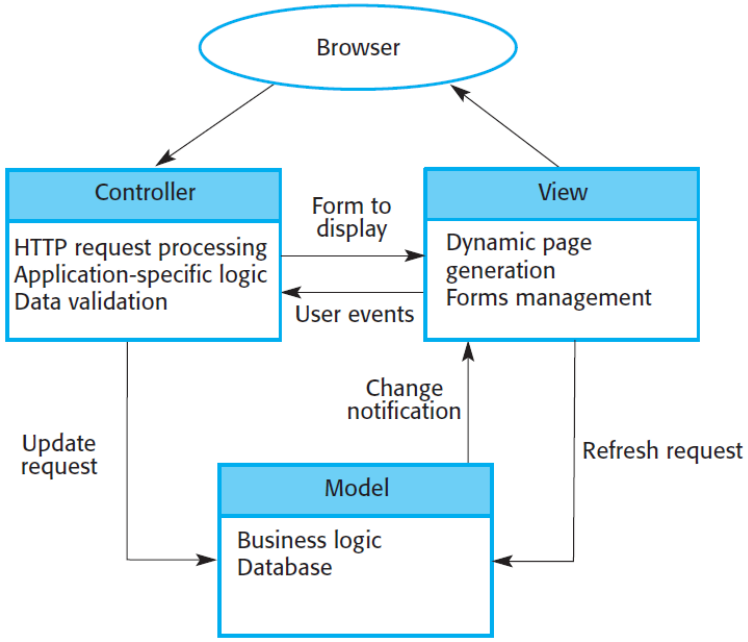



Figure 4.3: MVC Pattern

## 1.2. Object-Oriented Design

In this approach, a system is viewed as being made up of a collection of objects (i.e., entities). Each object is associated with a set of functions that are called its *methods*. Each object contains its own data and is responsible for managing it. The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object. Example of objects are employees, departments, items, etc.

If we take a Pay-roll system in an object-oriented design, the employee data, such as the names of the employees, their code numbers, basic salaries, etc., are distributed among different employee objects of the system.

### 1.2.1. Basic Terms

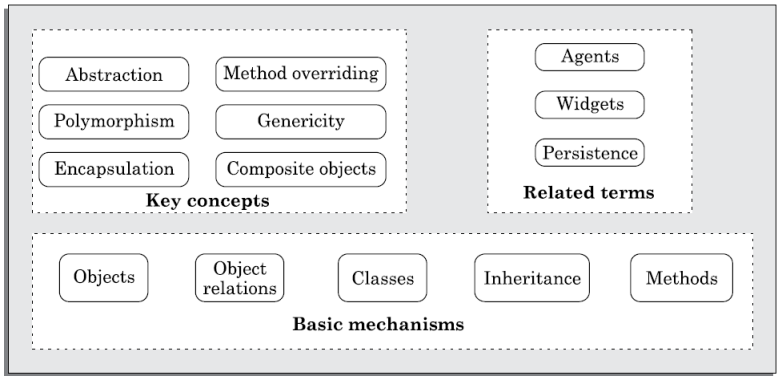



Figure 4.4: Concepts of Object-Oriented

This section will provide some definition to most used terms in object-oriented design:

1. **Object**: it is usually any tangible real-world entity that it can be interacted with in real life. Example in library system objects can be Books, Rooms, Employees, etc. Also, Object can be something like Controller, Schedular, etc. even if they are not tangible. advantage of considering a system as a set of objects is an excellent decomposition of the system into parts that have low coupling and high cohesion. Each object essentially consists of some data that is private to the object and a set of functions.
   Example: Library member is an object the has private data such as (name of the member, membership number, address, phone number, e-mail address, etc.) and operations such as (issue-book, find-books-outstanding, return-book, etc.).
2. **Class**: All the objects possessing similar attributes and methods constitute a class. all library members would constitute the class LibraryMember because each library member object has the same set of attributes and same operations.
3. **Method**: The operations (such as create, issue, return, etc.) supported by an object are implemented in the form of *methods*. An operation is a specific responsibility of a class, and the responsibility is implemented in the form of a method. However, it is at times useful to have multiple methods to implement a single responsibility.
4. **Class Relationships**:
   a. **Inheritance**: The inheritance feature allows one to define a new class by extending the features of an existing class. The original class is called the *base class*, *superclass*, or *parent class*, and the new class obtained through inheritance is called the *derived class*, *subclass*, or a *child class*.
   b. **Association and link**: it happen when two classes are taking each others' help (i.e., invoke each others' methods) to serve user requests. Example, Student and Subject objects they can have association relationship because Student can register in Subject and can invoke SubjectName method and access Subject information.
   c. **Composition**: represent part/whole relationships among objects. Objects which contain other objects are called *composite objects*. Example, Book object is composed of Chapter objects.

d. **Dependency**: one class depend on another class. Example, class C1 take an object from class C2 (class C1 dependent on C2), so any change on C2 would require change to class C1.

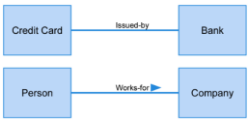
a. **Abstract Class**: Classes that are not intended to produce instances of themselves. In other words, an abstract class cannot be instantiated into objects. Abstract classes exist so that behavior common to a variety of classes can be factored into one common location, where they can be defined once. Example, "Issuable" is an abstract class and "Book" and "Journal" are concert class. Issuable can have the common method like "issue", "return", etc. which they have to be overridden in the concrete class.

5. **Encapsulation**: The data of an object is encapsulated within its methods. To access the data internal to an object, other objects have to invoke its methods, and cannot directly access the data. Because object do not directly change each other's data, they are weakly coupled.

### 1.2.2. Unified Modelling Language (UML)

UML is a language for documenting models. A model is a simplified version of a real system. The model aims to capture the important aspect with ignoring the internal details. Big and complex project can have multi-level of design and different models. To create these models, UML language used to create MUL diagrams. In UML, there are different association and relationship which can be used:

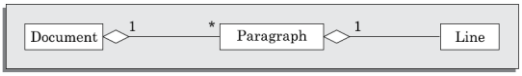
1- conceptual connection between classes




2- Direction/bidirectional on an association indicates control.

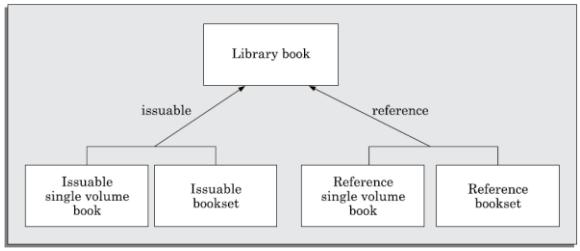

3- Multiplicity

4- Aggregation




5- Composition**:** in which the parts are existence-dependent on the whole. If components can dynamically be added to and removed from the aggregate, then the relationship is expressed as *aggregation.* If the components are not required to be dynamically added/delete, then the components have the same lifetime as the composite. In this case, the relationship should be represented by *composition*.


6- Inheritance




7- Dependency


## 1.2.3. OO Design Process

To start the process of the designing the class, we need first to identify the classes and their data and methods. After that we establish based on the requirement the relationship. So we need to identify the inheritance relation and if there is a need of abstraction, dependency, aggregation, and identify the multiplicity weather it is “zero or more” or “1 or more”, etc. UML Can be used to represent objects as showing in the figure 4.5, in which each class (such as WeatherStation), represented by rectangle, has private data and methods.

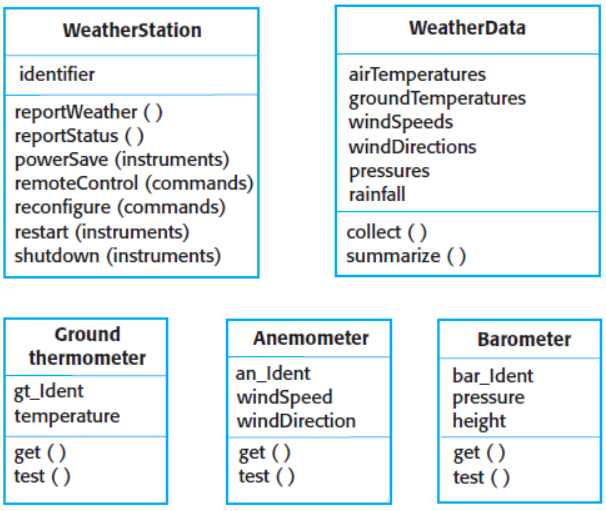



Figure 4.5: Example of UML object representation

Also, part of the object-oriented design, using UML to create use cases and sequence diagram that can be used to represent the sequence of using objects and method invoking. The following figure is example of sequence diagram for weather app base on the objects in the figure 4.5.

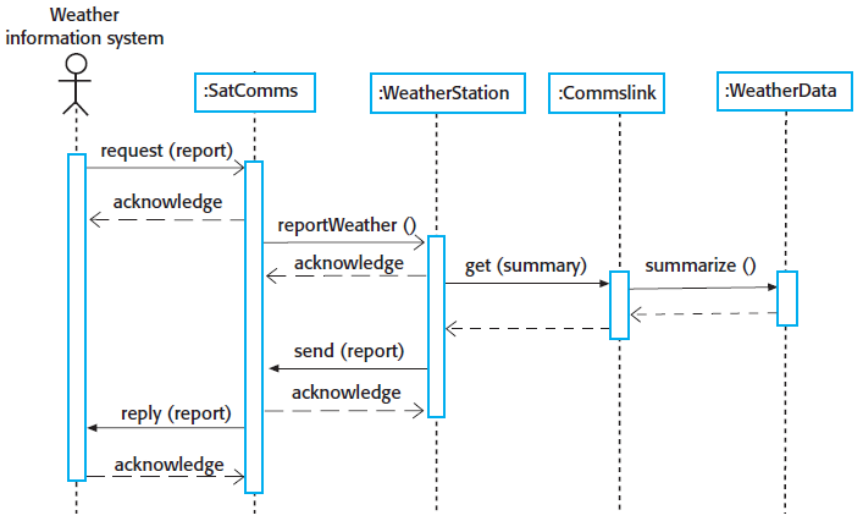



Figure 4.6: Example of sequence diagram for data collection in weather app

The following figures represent a class diagram for "Ordering System" and "Invoice and Payment" which show the objects and class with relationship among them.

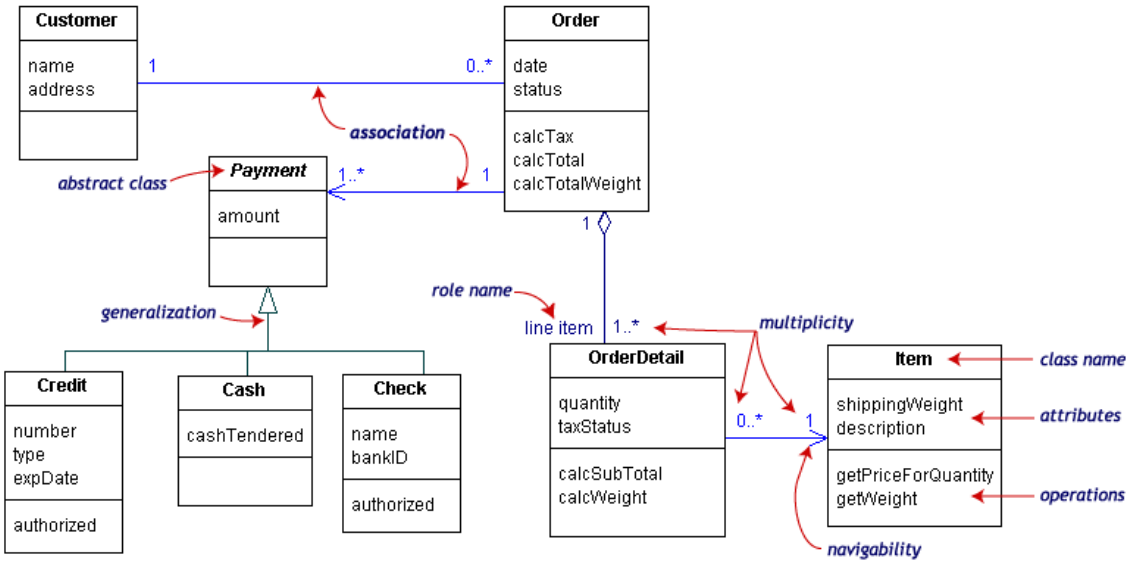



Figure 4.7: UML Class diagram example

## 1.2.4. Creational Design Patterns

Creational patterns provide different object creation mechanisms which help in increase reuse of the system code. The following the most common patterns:

1- Builder: it aims to separate the construction of a complex object from its representation. This type of pattern let you construct object step by step with different type of representation. Imagine you are creating function that make pizza with 10 optional toping. You could create different override constructors for different combination of the optional parameter (telescoping constructor) which unreasonable and to much work.

```
class Pizza {
    Pizza(int size) { ... }
    Pizza(int size, boolean cheese) { ... }
    Pizza(int size, boolean cheese, boolean pepperoni) { ... }
    // ...
```

Instead, we create builder for making pizza. In which, we have method or constructor that set attribute the object and return the same object after the change.

```
class PizzaBuilder{

    private Dough dough;
    private Cheese cheese;
    private HashSet<Topping> toppings = new HashSet<Topping>();

    public PizzaBuilder(){

    }
    public PizzaBuilder setDough(Dough dough){
        this.dough = dough;
        return this;
    }
    public PizzaBuilder setCheese(Cheese cheese){
        this.cheese = cheese;
        return this;
    }
    public PizzaBuilder addTopping(Topping topping){
        this.toppings.add(topping);
        return this;
    }
    public Pizza buildPizza(){
        return new Pizza(this.dough, this.cheese, this.toppings);
    }

}
```

```
Pizza myPizza = new PizzaBuilder()
        .setDough(Dough.WHITE)
        .setCheese(Cheese.WHITE)
        .addTopping(Topping.Kittens)
        .addTopping(Topping.Salami)
        .buildPizza();
```

2- Factory method: define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.

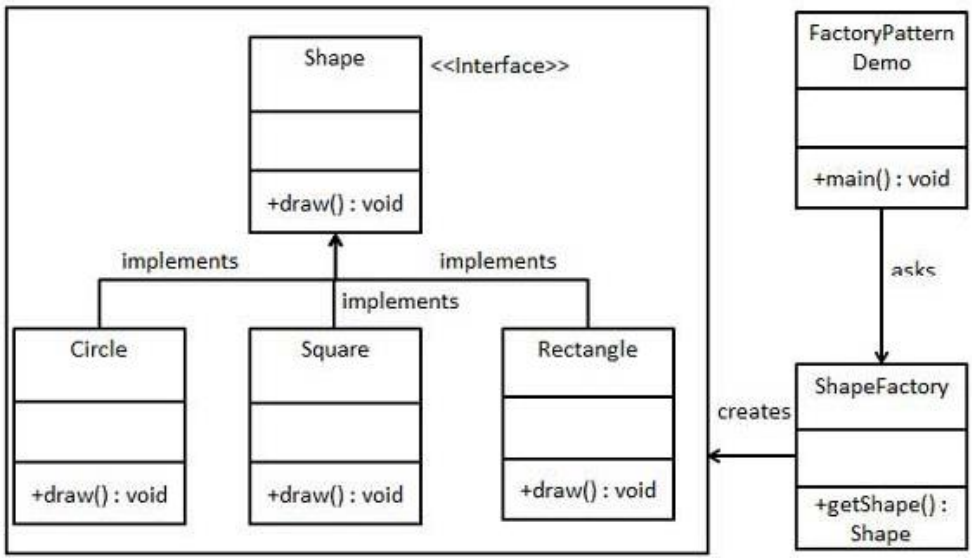



Figure 4.8: Example of factory method.

The following is example code of create factory method. In the code, we have interface Shape which is implemented by two classes. The ShapeFactory is returning an object of type Shape which later can be used to call a function.

```
public interface Shape {
  void draw();
}
public class Rectangle implements Shape {
  @Override
  public void draw() {
    System.out.println("Inside Rectangle.");
  }
}

public class Square implements Shape {
  @Override
  public void draw() {
    System.out.println("Inside Square.");
  }
}

public class ShapeFactory {
  public Shape getShape(String shapeType){
    if(shapeType == null){
      return null;
    }
   if(shapeType.equalsIgnoreCase("RECTANGLE")){
      return new Rectangle();
    } else if(shapeType.equalsIgnoreCase("SQUARE")){
      return new Square();
    }
    return null;
  }
}

public class FactoryPatternDemo {
  public static void main(String[] args) {
    ShapeFactory shapeFactory = new ShapeFactory();.
    Shape shape1 = shapeFactory.getShape("RECTANGLE");
    Shape1.draw();
    Shape shape2 = shapeFactory.getShape("SQUARE");
    Shape2.draw();
  }
}
```

3- Abstract factory pattern: it provides an interface for creating families of related objects without specifying their classes. In the figure 4.8, the abstract factory for vehicle is invoke another factory based on the type provided, such as car. The car factory will create object based on the type of car type provided. So factory is creating and return object.

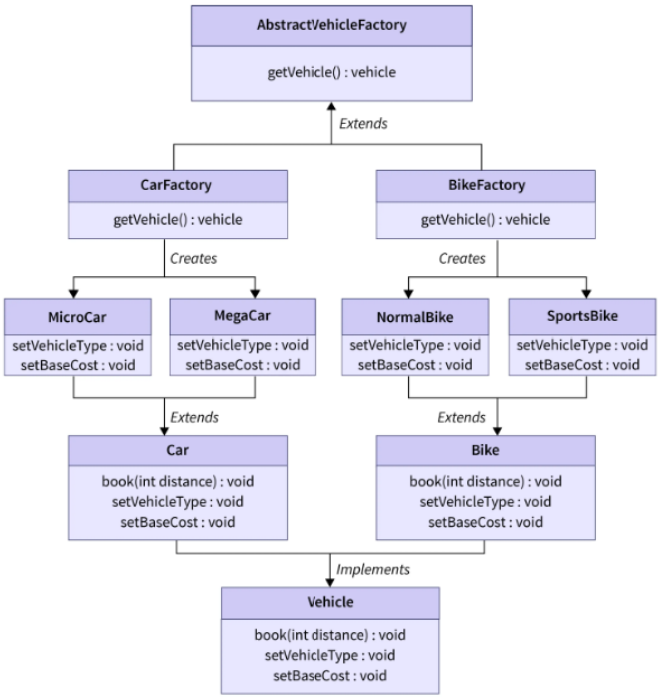


Figure 4.9: Example of abstract factory

Another short example is shape creating factory. In which, Factory producer return factory either rounded or shape. The shape factory will return object such as rectangle. The return object can be used and called as normal as, see main function.

```
public class FactoryProducer {
  public static AbstractFactory getFactory(boolean rounded){
    if(rounded){
      return new RoundedShapeFactory();
    }else{
      return new ShapeFactory();
    }
  }
}

public abstract class AbstractFactory {
  abstract Shape getShape(String shapeType) ;
}

public class RoundedShapeFactory extends AbstractFactory {
  @Override
  public Shape getShape(String shapeType){
    if(shapeType.equalsIgnoreCase("RECTANGLE")){
      return new RoundedRectangle();
    }else if(shapeType.equalsIgnoreCase("SQUARE")){
      return new RoundedSquare();
```

```
    }
    return null;
  }
}

public class ShapeFactory extends AbstractFactory {
  @Override
  public Shape getShape(String shapeType){
    if(shapeType.equalsIgnoreCase("RECTANGLE")){
      return new Rectangle();
    }else if(shapeType.equalsIgnoreCase("SQUARE")){
      return new Square();
    }
    return null;
  }
}

public interface Shape {
  void draw();
}

public class Rectangle implements Shape{
  @Override
  public void draw() {
    System.out.println("Inside Rectangle.");
  }
}

public class Square implements Shape{
  @Override
  public void draw() {
    System.out.println("Inside Square.");
  }
}

public class RoundedRectangle implements Shape{
  @Override
  public void draw() {
    System.out.println("Inside RoundedRectangle.");
  }
}

Class demo {
   public static void main(String[] args) {
   //get factory of type shape
        AbstractFactory shapeFactory = FactoryProducer.getFactory(false);
   // gat shap object from the factory
        Shape shape1 = shapeFactory.getShape("RECTANGLE");
```

```
   // call method from the returned object
      shape1.draw();

      AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);
      Shape shape3 = shapeFactory1.getShape("RECTANGLE");
      shape3.draw();
   }
}
```

4- Singleton: it ensures that a class has only one instance, while providing a global access point to this instance.

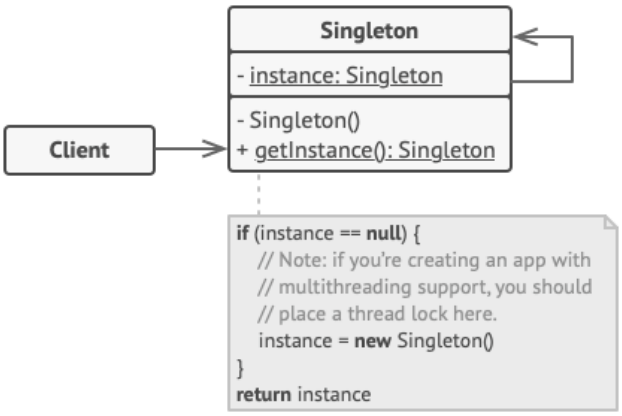
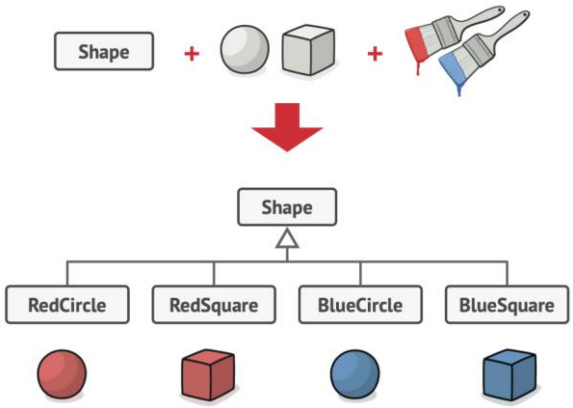



Figure 4.10: example of singleton

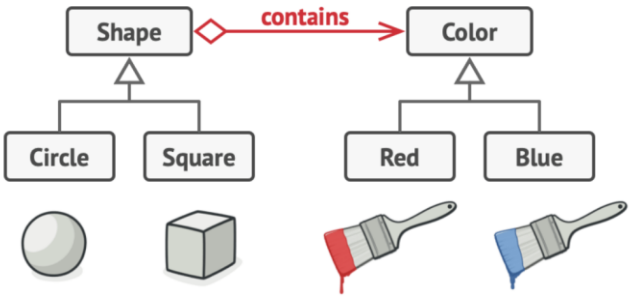
## 1.2.5. Structural Design Pattern

1- Adapter: it allows objects with incompatible interfaces to collaborate.
Example: Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user. Later, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format. You could change the library to work with XML. However, this might break some existing code that relies on the library. And worse, you might not have access to the library's source code in the first place, making this approach impossible. To solve this issue, you can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.

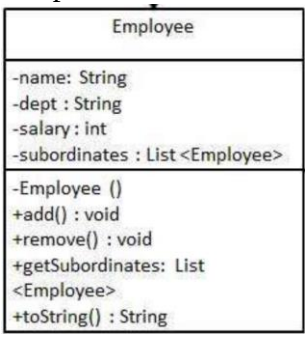
2- Bridge: lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.
Example: Say you have a geometric Shape class with a pair of subclasses: Circle and Square. You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses. However, since you already have two subclasses, you'll need to create four class combinations such as BlueCircle and RedSquare. Adding new shape types and colors to the hierarchy will grow it exponentially.



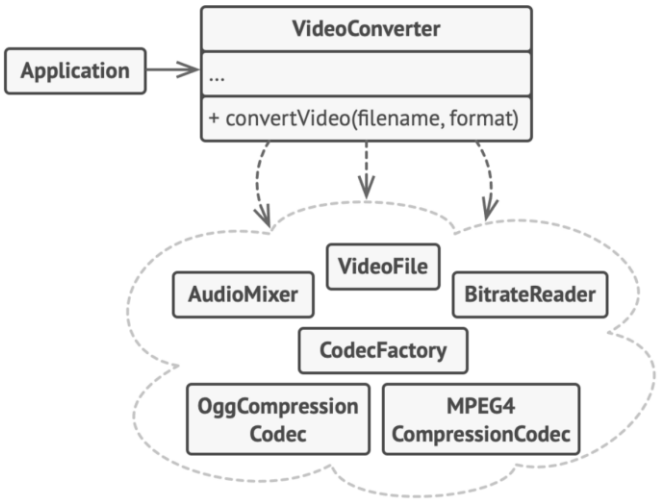

The Bridge pattern attempts can solve this problem by switching from inheritance to composition. This means, you extract them into a separate class hierarchy instead of having all of its state and behaviors within one class. Using this approach, we can extract the color-related code into one class with subclasses: Red and Blue. The Shape class gets a reference field pointing to one of the color objects. Now the shape can delegate any color-related work to the linked color object. That reference will act as a bridge between the Shape and Color classes. So, adding new colors won't need modifying the shape hierarchy, and vice versa.




3- Composite: it lets you compose objects into tree structures and then work with these structures as if they were individual objects.
For example, you have two objects: Employee which has sub-object (subordinates). Solution of this is using composite pattern to include object as tree.




4- Façade: it provides a simplified interface to a library, a framework, or any other complex set of classes.
Example: an app that uploads short videos with to social media could potentially use a professional video conversion library. You can interact with library and call each method/class as you need. A better option, creating a class with the single method encode(filename, format). After creating such a class and connecting it with the video conversion library, you'll get façade pattern as "encode" function will simplify all the complex work.



## 1.3. Database Design

There are many types of databases that can be used in a system. During the design, the type of database needs to be specified. Using relational database is common to deal with data. In this approach, data is stored in tables. These tables can be related to each other based on some constrained. Build a relational database help save space by reduce duplication, avoid deleting record by mistake and other advantage.

### 1.3.1. Normalization

To create a database, you need to normalize it to be in standard form. There are 3 most common level of normalizations.

1- First (1NF): some properties need to be met at this level:
   a. Each column must have a unique name.
   b. The order of the rows and columns doesn't matter.
   c. Each column must have a single data type.
   d. No two rows can contain identical values.
   e. Each column must contain a single value.
   f. Columns cannot contain repeating groups.

| NAME | WEAPON | WEAPON |
|---|---|---|
| Shelly Silva | Broadsword | |
| Louis Christenson | Bow | |
| Lee Hall | Katana | |
| Sharon Simmons | Broadsword | Bow |
| Felipe Vega | Broadsword | Katana |
| Louis Christenson | Bow | |
| Kate Ballard | Everything | |

Figure 4.10: example of table that violate first norm form.

2- Second (2NF): some properties need to be met at this level:
   a. It is a 1NF
   b. All non-key field depend on all key fields.

| TIME | GAME | DURATION | MAXIMUMPLAYERS |
|---|---|---|---|
| 1:00 | *Goblin Launch* | 60 mins | 8 |
| 1:00 | *Water Wizzards* | 120 mins | 6 |
| 2:00 | *Panic at the Picnic* | 90 mins | 12 |
| 2:00 | *Goblin Launch* | 60 mins | 8 |
| 3:00 | *Capture the Castle* | 120 mins | 100 |
| 3:00 | *Water Wizzards* | 120 mins | 6 |
| 4:00 | *Middle Earth Hold'em Poker* | 90 mins | 10 |
| 5:00 | *Capture the Castle* | 120 mins | 100 |

Figure: 4.11: table represent game schedules

The primary key in above table is Time+Game. The table meet the requirement for 1NF but it has issue with 2NF as following:

- Update—If you modify the Duration or MaximumPlayers value in one row, other rows containing the same game will be out of sync.
- Deletion—if you want to cancel the *Middle Earth Hold'em Poker* game at 4:00, so you delete that record. Then you've lost all the information about that game. You no longer know that it takes 90 minutes and has a maximum of 10 players.
- Insertion—You cannot add information about a new game without scheduling it for play.

3- Third (3NF):
   a. It is a 2NF
   a. It contains no transitive dependencies. Which means non-key field's value does not depend on another non-key field's value.

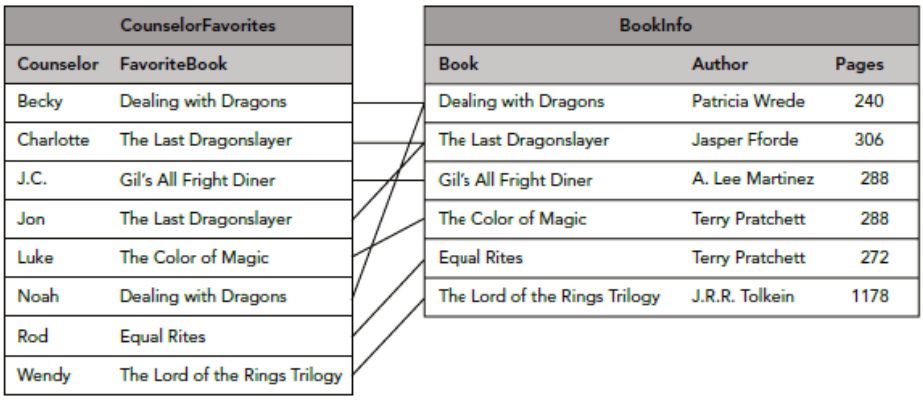
| COUNSELOR | FAVORITEBOOK | AUTHOR | PAGES |
|---|---|---|---|
| Becky | *Dealing with Dragons* | Patricia Wrede | 240 |
| Charlotte | *The Last Dragonslayer* | Jasper Fforde | 306 |
| J.C. | *Gil's All Fright Diner* | A. Lee Martinez | 288 |
| Jon | *The Last Dragonslayer* | Jasper Fforde | 306 |
| Luke | *The Color of Magic* | Terry Pratchett | 288 |
| Noah | *Dealing with Dragons* | Patricia Wrede | 240 |
| Rod | *Equal Rites* | Terry Pratchett | 272 |
| Wendy | *The Lord of the Rings Trilogy* | J. R. R. Tolkein | 1178 |

Figure 4.12: Counselors' favorite books

This table meet the requirement of 2NF but not 3NF. It has the following issues (which solve in the figure 4.13):

- Update: If you change the Pages value for Becky's row, it will be inconsistent with Noah's row as they have same favorite book. Also, if Luke changes his favorite book to different book, the table loses the data it has about "*The Color of Magic"*.
- Deletion: If you remove J.C. record from the table, you lose the information about *Gil's All Fright Diner*.

- Insertion: You cannot add information about a new book unless it's someone's favorite. Conversely, you can't add information about a person unless he declares a favorite book.



| CounselorFavorites | |
|---|---|
| Counselor | FavoriteBook |
| Becky | Dealing with Dragons |
| Charlotte | The Last Dragonslayer |
| J.C. | Gil's All Fright Diner |
| Jon | The Last Dragonslayer |
| Luke | The Color of Magic |
| Noah | Dealing with Dragons |
| Rod | Equal Rites |
| Wendy | The Lord of the Rings Trilogy |

| BookInfo | | |
|---|---|---|
| Book | Author | Pages |
| Dealing with Dragons | Patricia Wrede | 240 |
| The Last Dragonslayer | Jasper Fforde | 306 |
| Gil's All Fright Diner | A. Lee Martinez | 288 |
| The Color of Magic | Terry Pratchett | 288 |
| Equal Rites | Terry Pratchett | 272 |
| The Lord of the Rings Trilogy | J.R.R. Tolkein | 1178 |

Figure 4.13: Solution to 3NF problems

### 1.3.2. Design database process

The designing process start with identifying the entity or object that need in the database. This entity can be users, order, products, etc. Then defining what type of data that each table will hold which will be columns, this include specifying the datatype of each column. Then, start identifying the relationship among the table. The relationship can be any of the following:

- One-to-one (1:1) relationship
- One-to-many (1:M) relationship
- Many-to-one (M:1) relationship
- Many-to-many (M:M) relationship

After these process and with consideration of performing the normalization process, the Entity Relationship Diagram (ERD) should be created. ERD is created with table name and the columns that in the table with specifying the type and the constrain. This diagram help creating a big picture about how the database will look like and how the tables are related to each other's. There are many tool to create such diagram (dbdiagram[1], lucid app[2], or others).

[1] https://dbdiagram.io/home
[2] https://lucid.app/documents#/dashboard

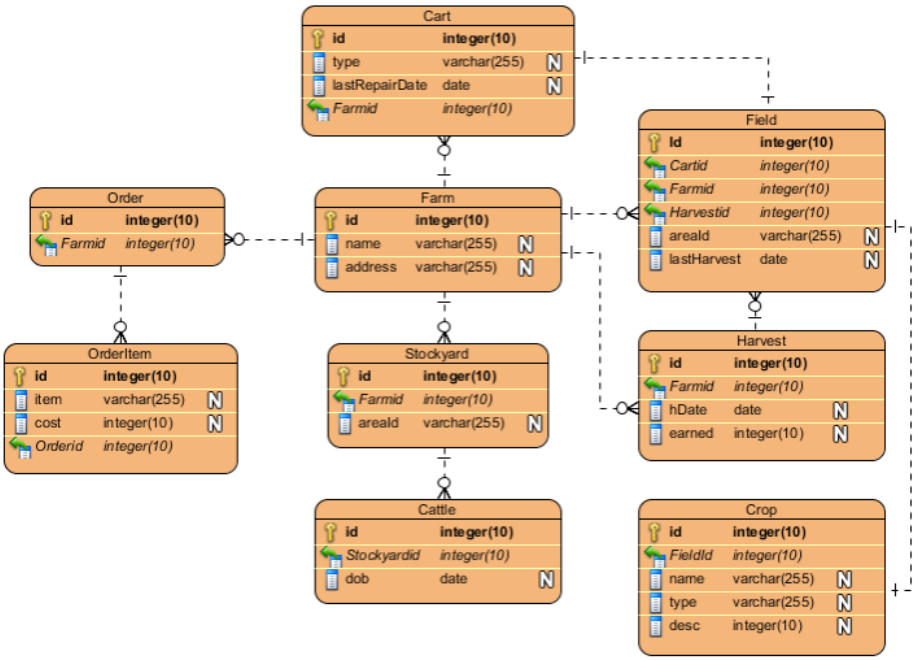



Figure 4.14: example of ERD

## 1.4. User Interface Design

A user interface is the portion of computer software that facilitates human–computer interaction (HCI), through which end users can access the software system. User interface is required, and every software should have some kind of interface.

### 1.4.1. Characteristics of User Interface

It is important to identify the characteristics that can differentiate good from bad user interface.

1- Easy to learn
2- Minimal Memory
3- Minimal Skilled Activities
4- Good Color Scheme: color scheme must not create pressure on the eye.
5- Minimal Assumptions
6- Alignment
7- System Documentation
8- System Menu
9- Change Confirmation
10- Responsiveness:

## 2. Software Project Management

Project of any type require organizing and managing to identify scheduling, process, teams, cost, and any other needed parts. The important of the project management can be summarize in few points:

- to deliver the software to the customer at the agreed time.
- to keep overall costs within budget.
- to deliver software that meets the customer's expectations.
- to maintain a coherent and well-functioning development team.

However, the software project is different from other type of project. The following are some of the differences:

- Invisibility: Software remains invisible, until its development is complete, and it is operational. In contrast, other project the progress can be seen. This invisibility makes controlling and assessing project difficult.
- Changeability: The software can be changed during the progress increase the complexity of managing the project.
- Complexity: Software has different part that are connected and communicated in order to provide the services. Each one of these parts can be develop separately but it needs to be fit in whole system.
- Uniqueness: Each software develop is specific and oriented to an organization. This makes the process of developing any software require continuous learning and adaptiveness.

### 2.1. Planning

After team agree to undertake the project, planning process should start. During this process project is breaking down the work into parts and assign them to project team members, anticipate problems that might arise, and prepare tentative solutions to those problems. The planning is done in three different stages:

1- **Proposal stage**: when you are bidding for a contract to develop or provide a software system. Planning helps you decide whether the company has the resources to complete the work and what the price.
2- **Startup phase**: planning who will work on the project, how the project will be broken down into increments, how resources will be allocated.
3- **Periodically throughout the project**: plan is updating to reflect new information. During the development team will learn more about the system and the capabilities of team. With the time requirements can change, the work breakdown has to be altered and the schedule extended, which all require altering the plan.

Planning phase includes identifying few main points that will help understand the general complexity of the project. One of these points is **Estimation** needs to be done. The following project attributes need to be estimated:

- **Cost**: How much is it going to cost to develop the software product?
- **Duration**: How long is it going to take to develop the product?
- **Effort**: How much effort would be necessary to develop the product?

The estimation is very important to effectively plan the upcoming activities are dependent on the accuracy the estimations.

- **Scheduling:** After estimation, the schedules for manpower and needed resources are developed to understand what and when each person and resource will be allocated.
- **Staffing:** Staff organization and staffing plans are made, to provide information about the team size and skills needed.
- **Risk management:** This includes risk identification and analysis.
- **Miscellaneous plans:** other plans such as quality assurance plan, and configuration management plan, etc.

## 2.2. Waterfall Methodology

The Waterfall Methodology is a linear and structured approach to project management. The software development life cycle (SDLC) consists of steps with formal hand-offs from one stage to the next.

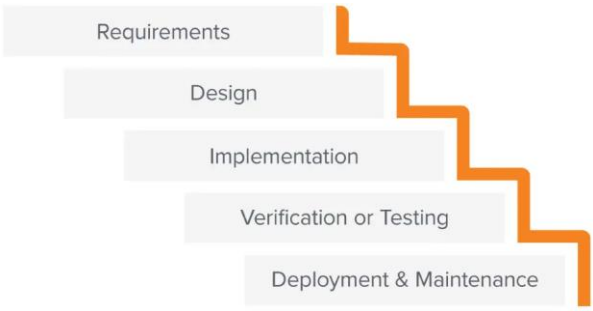



Figure 5.1: Waterfall process

The Waterfall methodology's insistence on upfront project planning and commitment to a certain definition of completion. Which mean, it is less flexible to changing. Changes that take place further in the process can be time-consuming and costly. The following is the challenges that comes with using Waterfall methodology in software project:

- Projects can take longer to deliver than with an iterative one, such as the Agile method.
- Clients often don't fully know what they need upfront, which results to request changes and new features later in the process.
- Clients are not involved in the design and implementation stages. Which resulting in less feedback during these stages.
- Deadline creep — when one phase in the process is delayed, all the other phases are delayed.

This does not mean Waterfall process inefficient. It is useful in project that:

- Doesn't have ambiguous requirements.
- Offer a clear picture of how things will proceed (clear picture about all the steps).
- Has clients who seem unlikely to change the scope of the project.

## 2.3. Agile Methodology

Agile methods of software development are iterative approaches where the software is developed and delivered to customers in increments. Unlike plan-driven approaches, the functionality of these increments is not planned in advance but is decided during the development. Agile projects consist of a number of smaller cycles. Each one of them is a project in miniature: it has a backlog and consists of design, implementation, testing and deployment stages within the pre-defined scope of work. The fundamental principle of Agile is covered by Agile Manifesto which are:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan.

The aspect Agile can summarized in the following point:

- Flexibility: The scope of work may change according to new requirements.
- Work breakdown: The project consists of small cycles.
- Value of teamwork: The team members work closely together and have a clear vision about their responsibilities.
- Iterative improvements: There is frequent reassessment of the work done within a cycle to make the final product better.
- Cooperation with a client: A customer is closely engaged in the development and can change the requirements or accept the team's suggestions.

## 2.4. Agile Frameworks

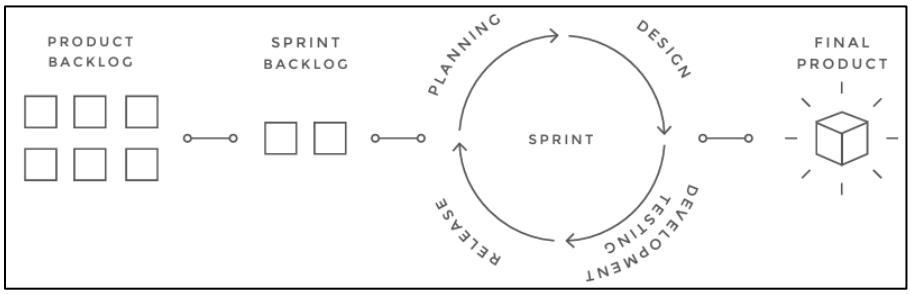
Agile is general term for a variety of frameworks and techniques that share the principles and values of Agile methodology. The most popular frameworks are:

### 2.4.1. Scrum Framework

It is the dominant agile framework that used by most company to manage their software development project. In Scrum, there are three main roles:

- **Scrum Master:** he/she is responsible on eliminating all the obstacles that might prevent the team from working efficiently.
- **Product Owner**: he/she is actively involved throughout the project, conveying the global vision of the product, and providing timely feedback on the job done after every Sprint.

- **Scrum Team: it** is a cross-functional and self-organizing team that is responsible for the product implementation. It should consist of up to seven team members, to stay flexible and productive.




5.2: General development Cycle

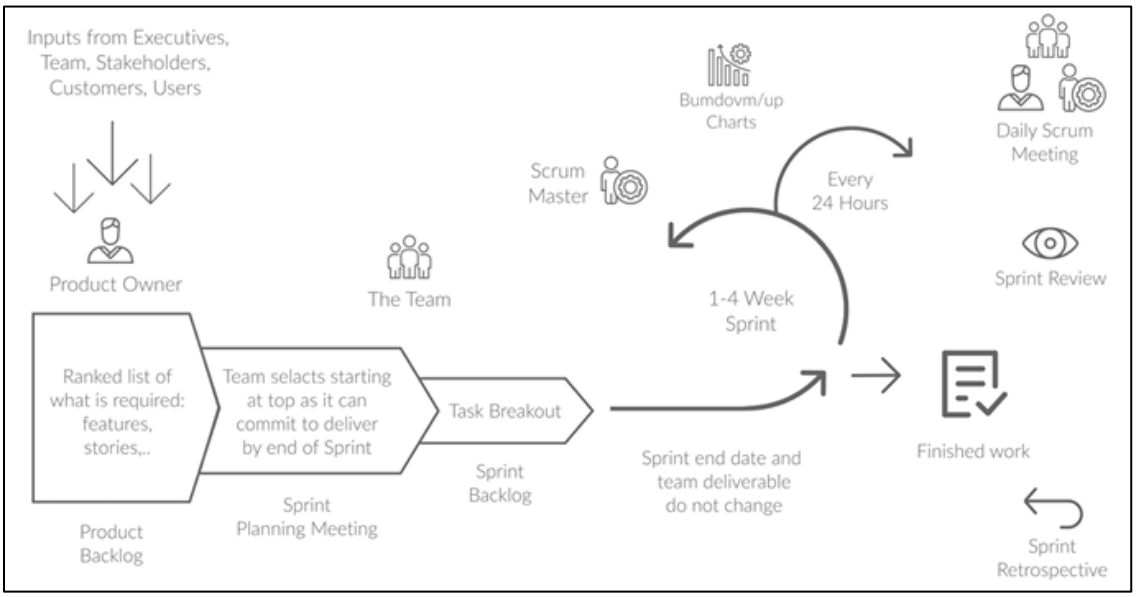
A basic unit of work in scrum, **Sprint**, is a short development cycle that is needed to produce a shippable product increment. A Sprint usually is between **one** and **four** weeks long. This makes it easier to plan and track progress. Scrum relies on three main **artifacts** which are used to manage the requirements and track progress:

1- the Product Backlog: is an ordered list of feature items that might be needed in the project's final product. The product Backlog updates as new requirements, fixes, features, and details are being changed or added.
2- the Sprint Backlog: is a list of tasks that the team must complete to deliver an increment of functional software at the end of each Sprint.
3- the Sprint Burndown Chart: is an illustration of the work remaining in a Sprint. It shows progress on a day-to-day basis and can predict whether the Sprint goal will be achieved on schedule.

The process is formalized through numbers of recurring meetings:

1- Daily Scrum (Standup): is a timeboxed meeting, during which a Development Team coordinates its work and sets a plan for the next 24 hours. The event lasts 15 minutes and should be held daily at the same place and time. Usually, each team member answer three main questions:
    a. What have done last day?
    b. What will do today?
    c. Is there any obstacle preventing the progress?
2- Sprint Planning: Everyone involved in the Sprint (a Product Owner, a Scrum Master, and a Development Team) participates in this event. They answer two key questions: which work can be done in the next Sprint and how this work will be done. The Sprint Planning lasts no longer than eight hours for a one-month Sprint.
3- Review/Demo: During this informal meeting, the team shows the work completed. The Sprint Review is a four-hour timeboxed meeting for one-month Sprints.

4- Retrospective meetings: to reflect on their work during the Sprint. Participants discuss what went well or wrong, find ways to improve, and plan how to implement these positive changes. The Sprint Retrospective is held after the Review and before the next Sprint Planning. It has duration is three hours for one-month Sprints.

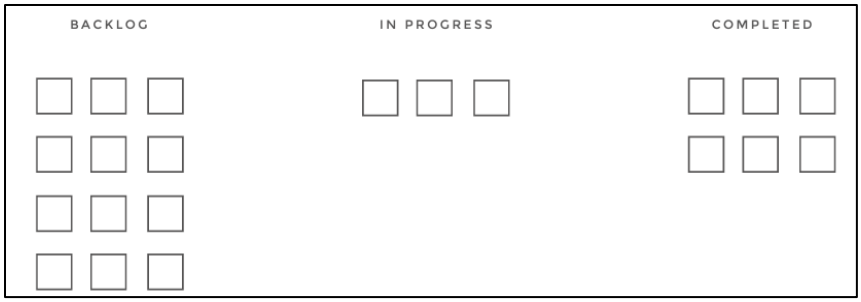



5.3: Scrum framework

Scrum **works well** for long-term, complex projects that require stakeholder feedback, which may greatly affect project requirements. So, when the exact amount of work can't be estimated, and the release date is not fixed, Scrum may be the best choice.

### 2.4.2. Kanban

Kanban focuses on the visualization of the workflow and prioritizes the work in progress (WIP), limiting its scope to match it effectively to the team's capacity. As soon as a task is completed, the team can take the next item from the pipeline. Thus, the development process offers more flexibility in planning, faster turnaround, clear objectives, and transparency. By using Kanban, teams can do small releases and adapt to changing priorities. Unlike Scrum, there are no sprints with their predefined goals. Kanban is focused on doing small pieces of work as they come up.




### 2.4.3. Extreme Programming (XP)

Its focus on technical aspects of software development. Extreme Programming is a set of certain practices, applied to software engineering in order to improve its quality and ability to adapt to the changing requirements. Most commonly used XP practices are:

- Test-Driven Development (TDD): it is an advanced engineering technique that uses automated unit tests to propel software design process. In the regular development cycle, the tests are written after the code. In TDD, a test is written first. This means that the unit tests are written prior to the code itself. According to this approach, the test should fail first when there is no code to accomplish the function. After that, the engineers write the code focusing on the functionality to make the test pass.
- Refactoring: it is a process of a constant code improvement through simplification and clarification. The process is solely technical and does not call for any changes in software behavior.
- Continuous Integration: is another practice agile teams rely on for managing shared code and software testing. Instead of doing short iterations, developers can commit newly written parts of a code several times a day. This way, they constantly deliver value to users.
- Pair Programming: this technique requires two engineers working together. While one of them is actually writing the code, the other one is actively involved as a watcher, making suggestions, and navigating through the process.

XP provides tools to decrease risks while developing a new system, especially when developers must write code within strict timeframes. It’s essential to know that XP practices are designed for small teams that don’t exceed 12 people.

# 6. Implementation/Coding

After the designing phase, the output (design document) will be the input to the next phase, implementation/coding. During this phase, different modules that identified in the design document are built according to their respective module specifications.

## 6.1. Coding Standards and Guidelines

Good software development organizations require their programmers to adhere to their well-defined standard style of coding which is usually called their *coding standard*. The standard can increase the understandability of the written code and uniform the appearance. Some of the rules are organization specific and some others are well-known standards. We will list the most common ones.

1. **Standard headers for different modules:** The header of different modules should have standard format and information.
   - Name of the module
   - Date on which the module was created.
   - Author's name
   - Modification history
2. **Naming conventions for global variables, local variables, and constant identifiers:** A popular naming convention is that variables are named using mixed case lettering. The most popular ones are:
   - Pascal case: The first letter of each word is capital, such as FirstName
   - camel case: the letters of first word are small and rest of the words start with capital, such as firstName.
   - snake case: The words are separated with underscore, such as first_name.
   - kebab case: the words are separated with dash, such as first-name. this is popular with URL naming and not in programming language because the dash is confused with minus sign.

   The most common naming conventions are, global variable, methods, and classes names is Pascal case (e.g., GlobalData) and local variable names is Camal case (e.g., localData). Constant names are all capital (e.g., CONSTDATA). However, this also depend on the language and the organization. The most important rule is that each variable should be given a descriptive name indicating its purpose.

3. **Length of the code line**: some organization specified how many characters should the line have before going to the next line.
4. **Do not use an identifier for multiple purposes, as mush as you can,** because it make enhancement and debugging more difficult.
5. **Code should be well-documented:** As a rule of thumb, there should be at least one comment line on the average for every three source lines of code.
6. **Specifying the length of functions:** A long function is usually very difficult to understand as it probably has many variables and carries out many different types of computations.

## 6.2. Code Review

Code review are an effective defect removal technique, which has been acknowledged to be cost-effective in removing defects. Code review for a module (that is, a unit) is undertaken after the module successfully compiles. That is, all the syntax errors have been eliminated from the module. Obviously, code review does not target to detect syntax errors in a program, but is designed to detect logical, algorithmic, and programming errors.

## 6.3. Code Documentation

## 6.4. Source Control

Source control is the practice of managing as well as tracking changes to your codebase. Nowadays, software is involved or embedded in every aspect of our life. Almost everything is running based on some type of software from space rockets to simple calculators. Every software goes through cycles and series of development and changes to reach a final and accepted case. Managing these changes is a challenging task for the software development team and managers. So, finding a way that can ease this change become a priority for researchers and developers. In the 1970s, the first system that can provide the needed functionality was developed by IBM. From that day, the journey of the source control system started.

In short term, it is tracking and managing changes to code. In long term, it is a system that is responsible for managing and tracking any changes that occur on targeted code without the need to store duplicated documents or code. We say document because source control can track changes on any type of text document.

### 6.4.1. Why do we need Source control?

Whether you are building a simple application or a large system, modifications are inevitable. Source control is not just for managing changes but also useful for collaborative software development due to the ability to merge different changes made to the main document. The most company consider it a vital component in their software development life cycle, that because it provides following benefits:

1. History revision of the code/document. The system stores versions of code on every change made. The version is not a complete copy of the code, but it is just what change since the last version. Every time developer commits changes, VSC creates a new version of the corresponding document.
2. Ability to revert to the previous version. If something happened that breaks the code, the developer easily can revert to any previous version of the code without losing any information. This issue may not happen a lot, but it will be a life saver when a problem arises. At the same time, developers can use it to compare between versions to see the differences and troubleshoot the issue.
3. Track code changes, what changes, and who made the changes. Most source control tools provide the ability to see what has been removed or added and who made the changes. This allows the team leads to review changes to approve or send back for improvement.

4. Collaboration on code: multiple team members can work on the same project at an isolation level until the code is ready to publish. So, any member who made changes to the project will not impact what other members do until they merge into the main version of the code.

Automate the process with DevOps. Using control sources consider the best practice for automated software development life cycle with DevOps. Having a reliable repository make another testing, packaging, and development tool have a reliable source (repository) that depends on to get the latest version of the code, with minimum human interaction.

### 6.4.2. Structure of Source Control

The basic principle of VSC is storing files with all their information and versions associated with them in the repository. The repository includes the latest versions of each document and all history of modifications unit the current stage, in addition to information related to these versions such as date and time, author, and description. During the history of the development of the VSC system, VSC was used in four different repository models:

1. Local: at an early stage of use VSC repository was kept locally and only the owner the person who can access the machine can interact with it. This limits the team to collaborate on a single project.
2. Shared folder: to overcome the limitations, the team stored a repository in a shared folder where the team on the same local network can access it.
3. In a large company using a shared folder is challenging. Client/server used to address this challenge. In which, repositories are stored on a server and every client can access (with permission)
4. Distributed repository: in which every user/developer has his/her local repository that they constantly updated and mostly keep synchronized with a remote online repository which shared with others.

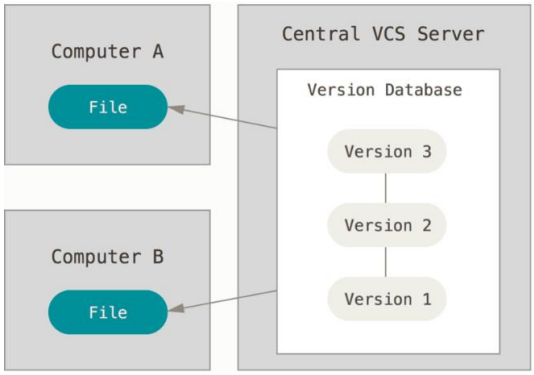
From the above, you can conclude that there are two types of VSC systems; centralized and distributed. In recent years, the distributed VSC has gained more attention because it allows collaboration without the need for a central repository.

### 6.4.3. Centralized Version Control

A centralized type was developed to overcome the challenge that developers faced when multiple developers work on the same systems. In this type, there is a single machine that stores the main copy of file versions history and keeps track of all changes and their information. It is called centralized because there is a central server or computer that holds the repository and all the version and maintain a complete record of all change and issues, whereas the developers can check out locally the project from the central server. To make changes developers can check out the needed project from the server and make their modifications which will be shared automatically with all developers. Developers in centralized type can only check out the latest version of a project from the repository to make their modifications on. This type has some drawbacks:

1. Inaccessible sever would prevent developers from pushing the latest change to the server.
2. Everything will be lost if the centralized repository got corrupted.

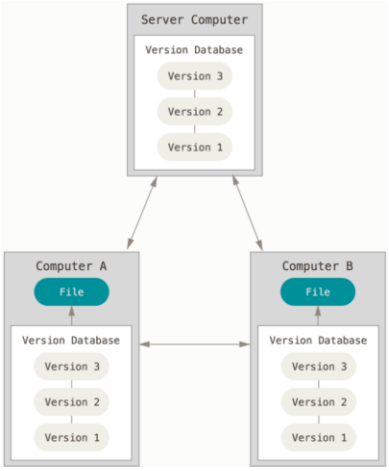
3. Because there is a single copy, there should be a restriction on making the change on it. This restriction will reduce the number of participations.




### 6.4.4. Distributed Version Control

Distributed type was developed to overcome the drawbacks of centralized by providing the ability to branch and merge, reducing the restriction of the local repository. This type works on keeping the entire history records of change on each local computer and syncing them with a remote server if needed. Because of that, developers can collaborate on a project and made their modifications (with all versions) without impacting others. At some point, they will push their change to the remote server to be accessible by other developers. There is three main advantage of using distributed type:

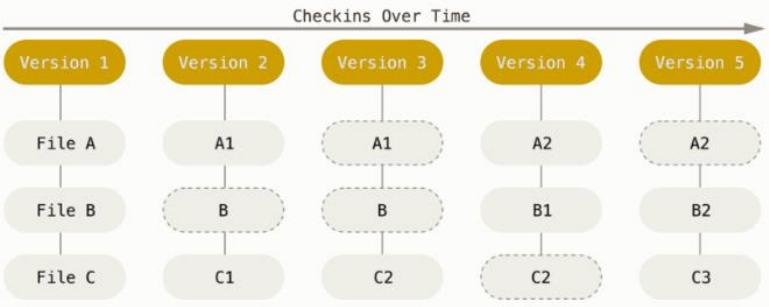
- Flexibility comes from easy to work independently but in a collaborative environment,
- Online hosting services, like GitHub, which remote access to teams across the world,
- Availability, the repository will always be available online even if the local was corrupted,
- it doesn't require access to remote servers to make the change,
- branching and merging can be done very easily.




### 6.4.5. Git

Git is a distributed version control, and it is one of the most popular version controls used nowadays. It was developed in 2005 as a free open source which made it a desirable tool for many companies to implement an extended version based on it for their profit. Every time developer

makes a change and committed, Git either stores a reference to the old file if not has changed or takes a new snapshot of the file that changed. This versioning by Git is done locally on the developers' computer (compatible with Linux, Windows, macOS), but it can be linked to a remote repository (GitHub, BitBucket,…). To keep track of changes made on files, Git used SHA-1 to checksum the content of files/directories. This helps Git in detecting if there is any change made to files without the need to compare them comprehensively.

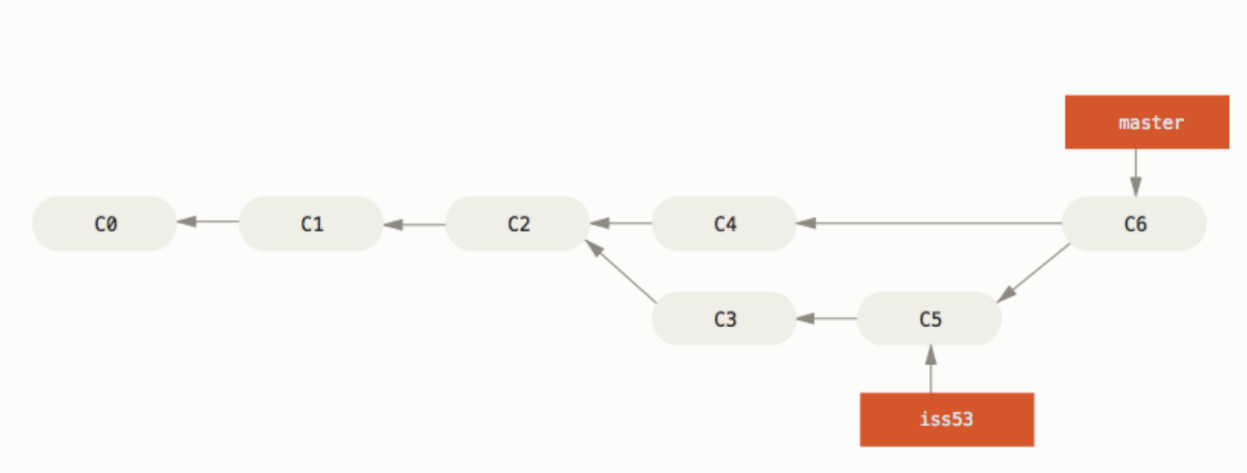



Losing data is not easy when using Git. After committing the change, Git stores the new version of file(s) without modifying the old one and stores the information in a database. For that reason, the most senior developer recommends committing frequent after finishing a functional code. To check your local repository state, git provides three states that describe the current situation:

- Modified: This means the current content of a file does not match the one in the repository due to some changes which have been made.
- Staged: file that has been modified and checked to be ready for commit. It kind of lists of files to which they will be committed. If a file is modified but not staged, then it will not be committed.
- Committed: It means the change has been added to the repository and the information was stored in the database.

### 6.4.6. Concept of Branching and Merging

Suppose you are working on software for a company that automates their work. One day, one of the divisions asked you to make some tweaks to one of the processes to match this division's needs. In this case, you as a developer can't modify the main repository; however, you need to create a secondary one. Version control provides branching to help address this situation. The developer can create another branch from the main one and modify it as needed without impacting the main branch. This concept can be used to create different applications from the main application, add/test new features, and restrict direct change on the main branching. After finishing development, the new branch can be kept as new software or it can be merged into the main branch.




In the above figure, you can see a new branch was created based on C2. Modifications were made to create C3 and C5 on the new branch. However, the new change was not added to the main branch. After C5 the new branch was merged into the main branch to create C6. After merging, modifications made to the new branch are added to the main one.

### 6.4.7. Starting with Git

Git has many commands line that is very useful in different situations. This article will list the most common one that developer needs to know to start using Git.

Setup identity: after installing the Git, you need to set the username and email using the following command:

1- *git config --global user.name "John Doe"*
   This command set the user name of the git locally.
2- *git config --global user.email johndoe@example.com*
   *This command will set the email of git*

Working with Git repository:

3- *git --vesrion*
   This will show you the Git version that you have installed.
4- *git init*
   Running this command inside the project directory will create a subdirectory called "*.git"* which will contain all the necessary files, but at this point, nothing will be tracked by Git. If the project already has files that need to be tracked, then used add command.
5- *git add .*
   This command will be added everything in the current path to versioning control. If you need to add a specific file, then you can replace the dot "." with the file name or you can use an expression to match all files of the same type.
6- *git commit -m "message"*

This command will go through all the files that have been added and store them safely. The *-m* is require adding a description of what this commit is about or what has been changed.

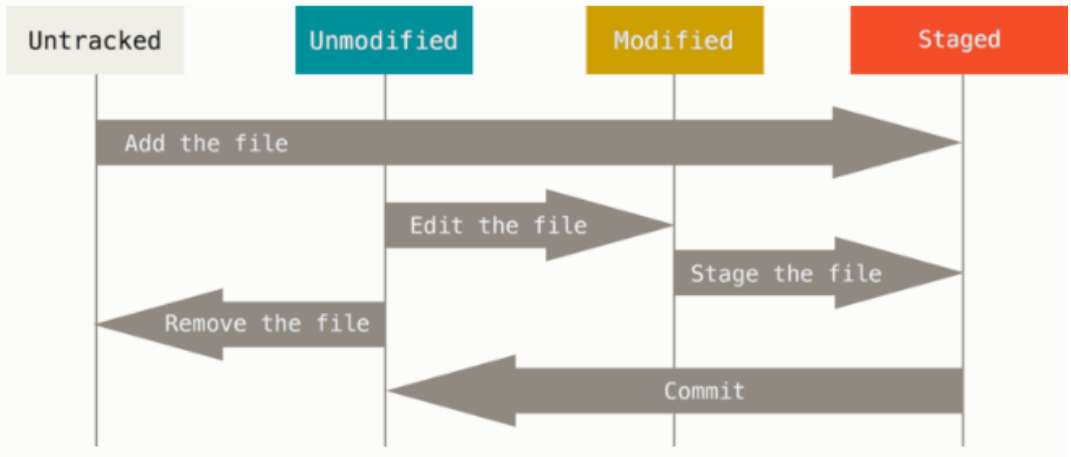
7- *git status*
Used to display the state of the directory and lets you know which changes have been staged (added) and which haven't.

8- *git clone repository_path*
Is used to create a new full copy of an existed repository.

9- *git push*
After committing, change ware store on the local machine but the remote repository branch has not been updated with the new change. This command will push the change to the remote repository with all information.

10- *git pull*
If someone else made a change on the remote repository that you have a clone, these changes will not be on your local machine. To get the recent version of a remote repository, this command is used.

11- *git checkout*
This command has many ways to be used, but basically, it asks git to get a specific version of the project from the repository. Specifying the version can be based on the *branch* name or *commit* id.




As figure show, all files are started as untracked until they been added using the “git add“ command. Which will be tracked. Any file you change, it will be flagged as modified so you can add it latter (stage it). You you stage the modified file it will be ready to commit, which will make the change permanent and store it in the repository.

12- *git log*

this command list all commits that been made along with name of the author and date of the commit. If you include -p  with command. The results will show the difference (add and deleted parts).

13- *git commit  --amend*

This command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit. This is helpful if you forget to commit some files, or you are doing some minor changes. The commit messages will be combined with precious commit message. You can use *--amend --no-edit* to prevent changing previous commit message.

14- *git stash*

This command used to save uncommitted changes (staged and unstaged) away for later. This is helpful if you are switching branch and you are not ready to commit changes that you are currently working one. To reapply the stash change, use command "*git stash pop"* which will remove the change from the stash and adding it to you working copy, or you can use "*git stash apply*" to apply the change without removing from stash.

15- *git checkout branch_name*

The git checkout command lets you navigate between the branches created by git branch. Checking out a branch updates the files in the working directory to match the version stored in that branch, and it tells Git to record all new commits on that branch. This command can be used to create new branch "*git checkout -b branch_name"*.

# 7. Software Testing

The aim of program testing is to help in identifying all defects in a program and show that a program does what it is intended to do. When you test software, you execute a program using artificial data. Then checking the results for any errors, anomalies, or information about the program's non-functional attributes that help correct or improve the program under test. However, in practice, even after satisfactory completion of the testing phase, it is not possible to guarantee that a program is error free. This is because the input data domain of most programs is very large, and it is not practical to test the program exhaustively with respect to each value that the input can assume.

## 7.1. What is Testing?

Testing is part of a broader process of software verification and validation (V & V). The difference between them are:

- **Validation:** *Are we building the right product?*
- **Verification:** *Are we building the product right?*

Software verification is the process of checking that the software meets its stated functional and non-functional requirements. The aim of software validation is to ensure that the software meets the customer's expectations. It goes beyond checking the specification to demonstrating that the software does what the customer expects it to do.

## 7.2. Important Terminologies

There are some terms that need to be cleared for better understanding.

1) **Mistake** is essentially any programmer action that later shows up as an incorrect result during program execution. Example, not initializing a certain variable.
2) **Error** is the result of a mistake committed by a developer. Terms *error*, *fault*, *bug*, and *defect* are used interchangeably.
3) **Failure** of a program means an incorrect behaviour exhibited by the program during its execution. Every failure is caused by one or more bugs present in the program.
4) **Test case** consist of [ *I, S, R*]:
    a. *I* is the data input to the program under test
    b. *S* is the state of the program at which the data is to be input, and
    c. *R* is the result expected to be produced by the program.
5) **Test scenario** is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output.
6) **Test script** is an encoding of a test case as a short program.
7) **Test suite** is the set of all test cases that have been designed by a tester to test a given program.

## 7.3. How to test a program?

Testing a program involves executing the program with a set of test inputs and observing if the program behaves as expected. If the program fails to behave as expected, then the input data and the conditions under which it fails are noted for later debugging and error correction.

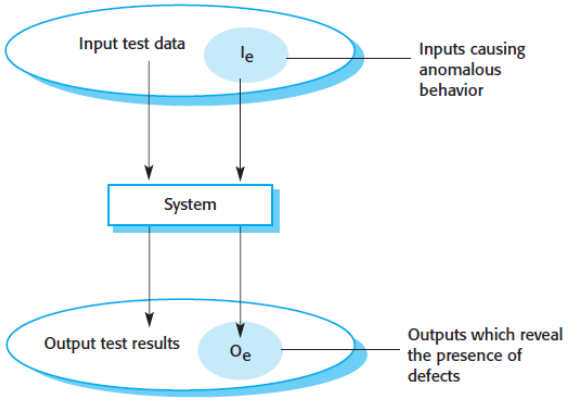



Figure 7.1: Illustrate testing model.

## 7.4. Types Of Software Testing

There are two types of testing white and black box,

- Black Box (Functional) Testing: it refers to the concept that a module is to be tested as to how well it meets its specifications (not the structural). This does not require the source code or any knowledge about the program structure. This test based on Designed without knowledge of the program's internal structure and design. Based on functional and non-functional requirement specifications.
- White Box (Structural) Testing: the structure of the module's source code is used together with the module's specifications to guide the test cases used to test the module. In this type of testing, the source code needs to be available to testing. This type tests typically based on coverage of the source code (all statements/conditions/branches have been executed).

## 7.5. Testing Levels

To make sure the program work correctly different level of testing is performed. In general, there are different levels:

1. **Unit Testing**: where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods
2. **Component Testing**: where several individual units are integrated to create composite components. Component testing should focus on testing the component interfaces that provide access to the component functions.

3. **System Testing**: where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.
4. **Acceptance Testing**: It is performed by given the product to a set of users to check whether it meets their needs which can expose more faults. Also called alpha/beta testing.

### 7.5.1. Unit Testing

Unit testing is usually automated to reduce the effort and speed up the process of testing. When you are testing object classes, you should design your tests to provide coverage of all the features of the object. This means that you should test all operations associated with the object. An automated test has three parts:

- *A setup part*, where you initialize the system with the test case, namely, the inputs and expected outputs.
- *A call part*, where you call the object or method to be tested.
- *An assertion part*, where you compare the result of the call with the expected result. If the assertion evaluates to be true, the test has been successful; if false, then it has failed.

Sometimes, the object that you are testing has dependencies on other objects that may not have been implemented or whose use slows down the testing process. In such cases, you may decide to use mock objects. Mock objects are objects with the same interface as the external objects being used that simulate its functionality. For example, a mock object simulating a database may have only a few data items that are organized in an array.

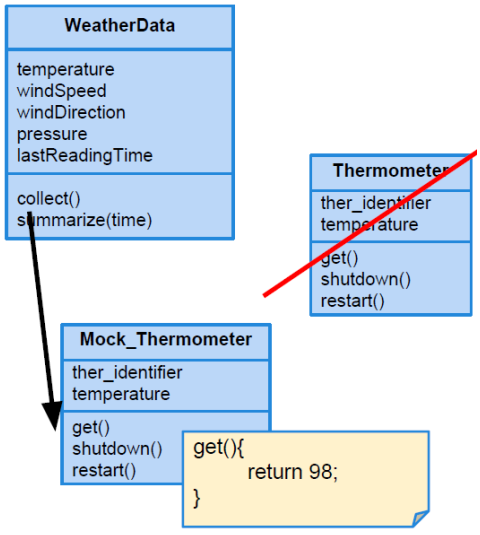



Figure 7.2: Example of Mock object

### 7.5.2. Unit Test Cases

Testing is expensive and time consuming, so it is important that you choose effective unit test cases. When choose test cases (input), you need to consider input that reflect normal operation

which create pass testing, and test case that could cause system to fail based on previous experiences. There are strategies that can be effective in choosing test cases:

- Equivalence Partitions Testing: is identifying groups of inputs that have common characteristics and should be processed in the same way. You should choose tests from within each of these groups.
- Boundary Testing: Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.

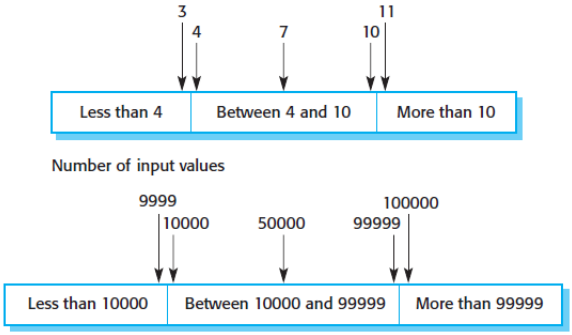



Figure 7.3: Example of partition and boundary testing

## 7.6. Structural Testing

It is type of white-box testing in which source code need to be available. This type is used to test the structure of the code. So, the test cases are driven from structure to test different aspects and path. In order to measure the effectiveness of the testing some metrics need to be included. These metrics are called coverage metrics.

### 7.6.1. Coverage Metrics

Coverage uses one or more criteria to determine how the code was examined during the execution of test suites. This can help give indication if we need to write more testing. If the coverage is low, that means, we need to perform/write more test case to cover more code based on the criteria/metrics. The most popular criteria are:

1) **Statement coverage** is a metric to measure the percentage of statements that are executed by a test suite in a program at least once.

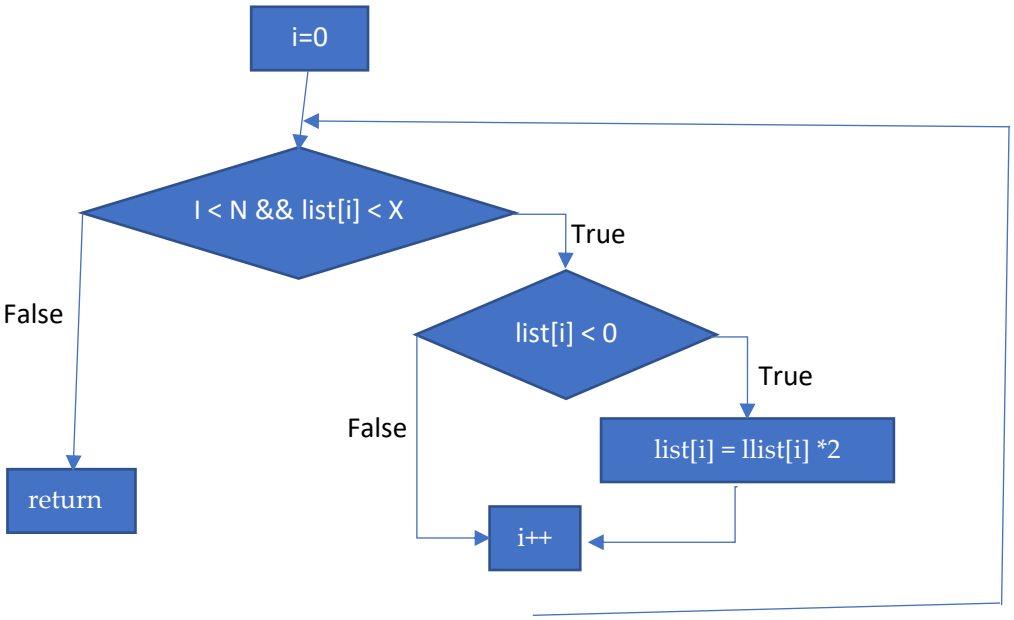
```
int computeGCD(int x,int y){
1 while (x != y){
2  if (x>y) then
3       x=x-y;
4  else
5       y=y-x;
6 }
7 count << x;
8 return x;
}
```

To get statement coverage, we need to have as many as possible statements executed using our inputs. By choosing the test set {($x$ =3, $y$ = 3), ($x$ = 4, $y$ = 3), ($x$ = 3, $y$ = 4)}, all statements of the program would be executed at least once. The coverage is the percentage of number of statements executed over the total number of statements.

2) **Branch coverage**: A test suite achieves branch coverage, if it makes the decision expression in each branch in the program to assume both true and false values. In other words, for branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed.
For same example above, to achieve branch coverage we need to have each branch to be resulted of at least one time true and at least one time false. So the good test suite could be {($x$ = 3, $y$ = 3), ($x$ = 3, $y$ = 2), ($x$ = 3, $y$ = 4)}.

Another example:

```
int twic(int list[], int N, int X)
{
      int i=0;
      while (i<N and list[i] <X)
      {
            if (list[i]<0)
                list[i] = llist[i] *2;
            i++;
     }
     return(1);
}
```



In the above example, to get the branch coverage we need to get each brach to be true once and false once.

3) **Condition coverage**: if each basic condition in every conditional expression assumes both true and false values during testing. That means every condition (example x>3) need to be once false and once true. If the branch has more than one condition, compound condition, (example (x>3 && y >5 ) then each one of the two condition need to be once false and once true. For the example (x>3 && y >5) we can write test cases as follow:

| Test case | x>3 | y >5 |
|---|---|---|
| x=5, y=7 | True | True |
| X=1, x=2 | False | False |

# References


1- Ian Sommerville, Software Engineering Tenth Edition, 2016
2- Elvis C. Foster, Software Engineering a Methodical Approach Second Edition, 2022
3- Rod Stephens, beginning Software Engineering Second Edition, 2023

4- Ronald J. Leach, development Introduction to Software Engineering Second Edition, 2016