Unit 1 - Software Engineering

1 Introduction to Software Engineering

After learning the several fundamental topics of a programming language and data structures, we can move on to learning how software systems are designed, build and managed. In this unit, you will learn:

- Definitions and Concepts of Software Engineering
- Requirements Engineering (partially done in Jupyter)
- Data Design & Modeling (mostly done in Jupyter)
- Development
    - Development Tools
    - Test-driven Development
- Testing, Deployment and Maintenance
- Agile Software Development (another document)

1.1 Definitions and Concepts of Software Engineering

Software engineering is a term that is made up of 2 words: Software and Engineering.

Software by definition is a combination of program/s, database/s and documentation in a suite that solves a specific problem, this software is called a software product.

Engineering on the other hand, is the use of scientific principles to design and develop machines or products.

Software Engineering is a branch of engineering where scientific principles, methods and procedures are used in the development of software. It's main goal is to produce efficient and reliable software to solve or enhance the productivity in a related problem domain in the most cost effective way.

> Quote from Fritz Bauer, a German computer scientist:
> Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machines.

Software has evolved from a single instruction provided to the machine to the present stage of decision-making software. Initially, computers where operated by inputting single real-time instructions then when programming languages were created, single instructions were framed into structures called programs and programs in turn grew into systems. From then, software started to fit into 2 classifications, namely application software and system software. Refer to figure 1 on the next page.

- **System software** comprises programs that control the operations of the hardware components, commonly called an operating system. They include Windows, Linux/Unix and DOS.
- **Application software** consists of programs purely for the tasks of the users, obtaining the inputs and processing them in the instructions already defined. They generally have no direct access to the hardware or any prerecorded system software of the device (aka BIOS). They are mostly built with the users productivity in mind.

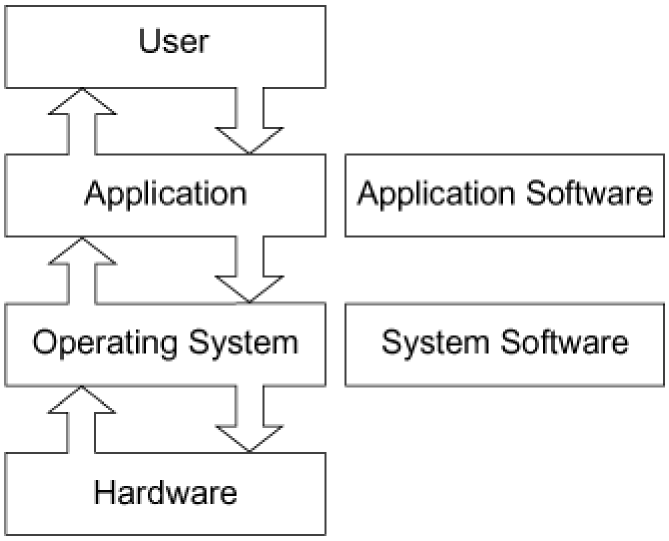


Figure 1: Interrelationship of software.

1.1.1 Stakeholders

From the beginning of every development, the most important group of people to identify are the **stakeholders**. These are the people or groups of people that are directly affected by the software development process and they can exist both within and outside the organization. Input from these people defines how the final software product will look and function. They are also the people whom defines the scope and compromises of the software as it is being developed therefore involving them throughout the development process is vital. Example of stakeholders are the end users who have to use the software for their work or the users who are working with products that are the output of the software, etc.

Overall, a software project team would consist of the following group of people:

- **Managers or company liaisons or customers** - these are the people who make the final decisions about the timeline, budget and scope. They are also the ones who are able to authorized more time for the project or reduce/remove project features.
- **Project Managers** - they keep track of every stage of the project. They also serve as the contact between the stakeholders and the development teams. They're main goal is to ensure that the software product created meets all the client's requirements.
- **Developers** - they build the software based on the requirements of the clients but they are also stakeholders as they possess the expertise to advise the executives/clients on which features are feasible and how long each would take to build.
- **Partners** - these are the outside groups of people that have an involvement in the development process because they either own 3rd party tools that the project needs or they are the client's business partners who needs to ensure that the software is compatibility with their systems.
- **Authorities** - depending on the type of project, these group of people may or may not be involved. The people included in this group are are legal, regulatory bodies, shareholders and company owners. They have the power to shut down the project if any parts of the project infringes on any legal or contractual bindings.

1.1.2 Software Development Life Cycle (SDLC)

In general the phases of software development can be largely mapped out as such,

1. **Defining and understanding the problem** - this is where the customer and the developer will interact with each other to document the features and requirements of the software.
2. **Designing** - where the software requirements are converted into design models. This is where teams will use languages such as **Unified Modeling Language (UML)** to model the software and non-software systems. This phase acts as a bridge between the first phase of defining and understanding the problem and the next phase, prototyping and coding therefore the designs produced in this stage should be **precise & specific**, **conform to budget & requirements** and **provide a direct solution to the problem**.
3. **Coding** - this is where the previously made design is transformed into lines of code using some programming language. Organizations (in this phase), will typically use a software versioning tool to help track the code changes and a bug tracker tool to help keep track of bugs.
4. **Testing** - depending on the software development methodology adopted by the company, this phase can be carried out at different points of the coding cycles (which will be elaborated as we go along). Testing is done both manually by many different groups of people ranging from the developer themselves to internal staff from different departments to professional teams of software testers and automatically by automated testing frameworks (eg: `PyTest`).
5. **Implementation** - in this phase (also called the **deployment phase**), the software product is released to the end users or into the production environment. The success of the implemented product depends on how much the end users like the product and prefer to use it regularly.
6. **Maintenance** - this is the process of modifying the production system after the delivery to correct the faults, improve the performance and adapt to the changing environment. This is seen in software products where companies provide support and training to customers and/or cooperate clients on their products. Therefore, the responsibility of the team of developers does not end after the software product is implemented.

These phases (as a whole) are called **Software Development Life Cycle**. One the traditional method (devised in the 1970s) is the tried and tested methodology is called the **Waterfall Model**. This model has processes similar to a step waterfall, that means each phase is carried one after the other in a linear manner. Refer to figure 2 below.

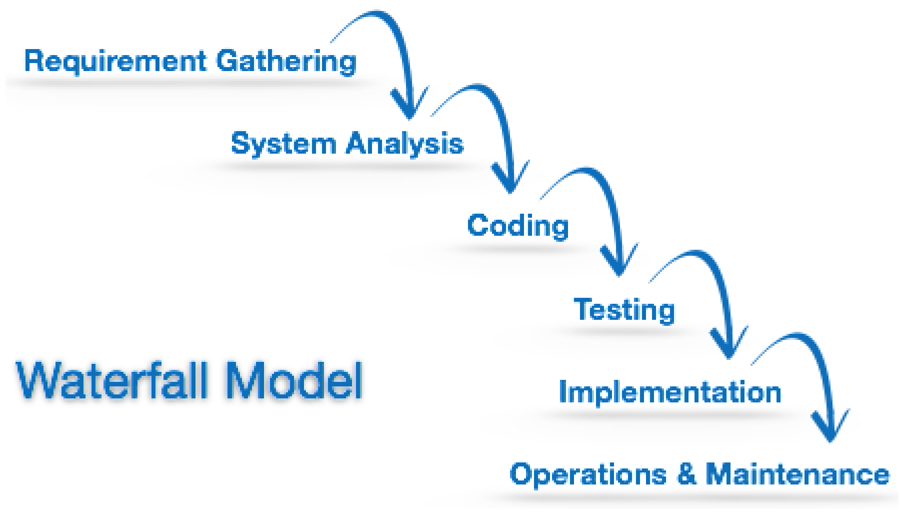


Figure 2: Waterfall Model.

This methodology assumes several points:

- total knowledge of the requirements of the system obtained before its development
- the output of each phase is correct and complete before advancing to the next phase
- each process is irreversible that means retracting is not allowed until the system is completed

The advantages of this model are:

- easy to understand
- easy for implementation
- ensures a comprehensive, functional, well-integrated system.
- ensures user participation (since the user must signoff on each phase)

The disadvantages of this model are:

- does not match well with reality
- unrealistic to freeze the requirement upfront
- software product is delivered only at the last phase
- difficult and expensive to make changes

Although waterfall model is one of the earliest methodologies created, there have also been changes and adaptations to it to produce other methodologies. One popular methodology used in most organizations is the Agile Software Development Model where the software product is constructed and tested in small increments, focusing on the essential requirements early. We will be further detailing this method in the section of the same name.

1.1.3 Software Architecture

Earlier we have learnt about stakeholders, they are the group of people that have a vested interest in the development of the software. Another part of software development that is also decided in the early phases is the **software architecture**. The software architecture is the foundation of how the software is to be built as a whole. Do take note that software architecture is also referred to as software architecture pattern and it is different from the term software design pattern which will be explained later.

Software architecture pattern are terms such as **Microservices, Client/Server, Serverless, Event-Driven**, etc. Each of these have specific blueprints for the system and the developing project. For example, a microservice architecture involves developing small, independent modular services where each service solves a specific problem or performs a unique task and these modules communicate with each other through well-defined API to serve the business goal. Refer to figure 3 on the next page.

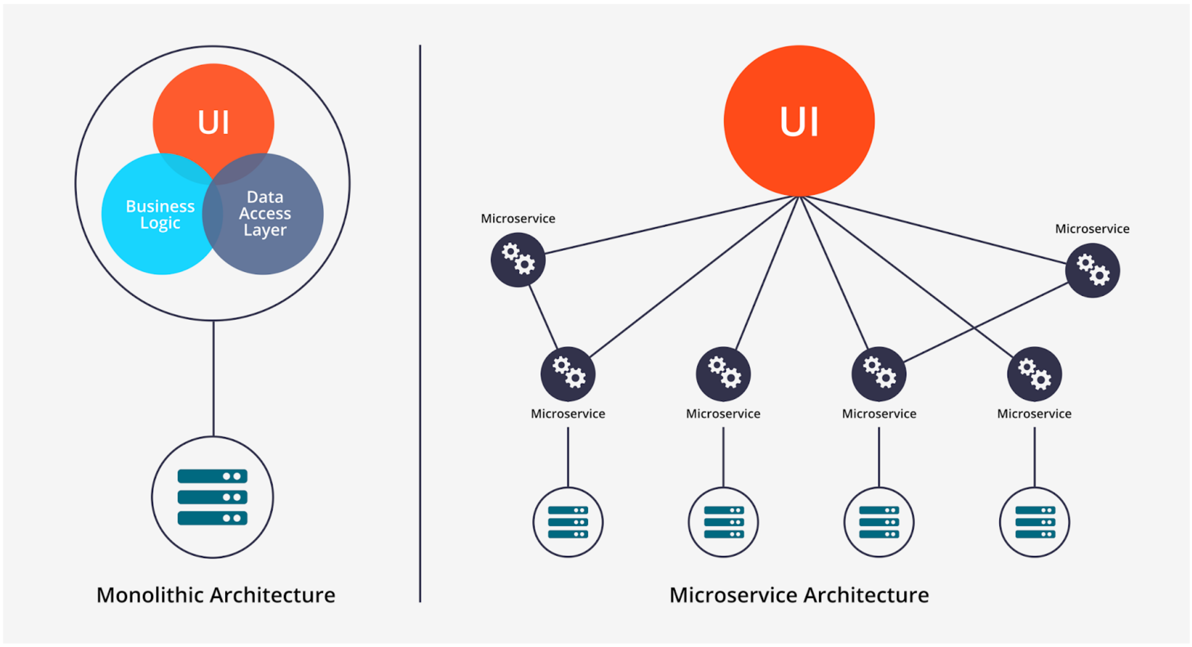


Figure 3: Monolithic vs Microservice Architectures.

We can say that the software architecture is responsible for **the skeleton** and **the high-level infrastructure** of a software, whereas the software design pattern is responsible for the **code level design** such as, what each module is doing, the classes scope, and the functions' purposes, etc. For example, the adapter pattern is a structural design pattern where it allows incompatible interfaces to work together. This is very commonly used in systems that has to interface with legacy code. The adapter will "translate" the inputs into an appropriate format for the use by the legacy code.

1.1.4 Characteristics of Quality Software

On the whole, the characteristics of a good quality software are:

1. **Completeness** - The software product being developed should meet all the requirements of the customers no matter how complex the product is, and the requirements agreed upon should be present in the product. The final product should not miss out any of the required functions.
2. **Consistency** - Operations of the product should be stable and capable of handling the problems in implementation.
3. **Durable** - Customers may vary in technology requirement or geographical aspects and the product should respond to a diversity of environments.
4. **Efficiency** - The product should not waste the resources, execution and waiting time of operations and the memory spaces allocated for the processes.
5. **Security** - The stored contents and the input statements of the software should be confidential and conserved if they possess a high significance.
6. **Interoperability** - Communication between other processes and environments should be enabled to apply a product universally.

1.2 Requirements Engineering

In the previous section, we learnt about the phases of the Software Development Life Cycle. We also learnt that developing a software product is nothing but taking the requirements of the clients to the next level. To have a successful product, gathering, documenting, disseminating and managing the requirements is vital. Although it sounds easy, gathering the requirements effectively and managing them efficiently is a complex task and needs to be handled in a systematic manner.

In this section, we will learn about the steps and guidelines of requirements engineering so that our software projects do not fail due to faulty or incomplete requirements capturing as according to industry average, more than 90% of projects fail due to this reason.

1.2.1 Importance of Requirement Engineering

Requirements are the stepping stones to the success of any project. If projects get started without properly understanding the user’s needs or without exploring the multiple dimensions of the requirements, there will be misalignment at the end between the final result delivered and the user’s expectations of the project resulting in a lot of rework. Projects must also take into consideration the set time frame and budget considerations. This is no different from software development projects.

For example, if a defect is found in the requirements review stage, it will cost a lot less to fix the issue than if it was found in the later stages or worst after it has been released to the clients. Refer to figure 4 below.

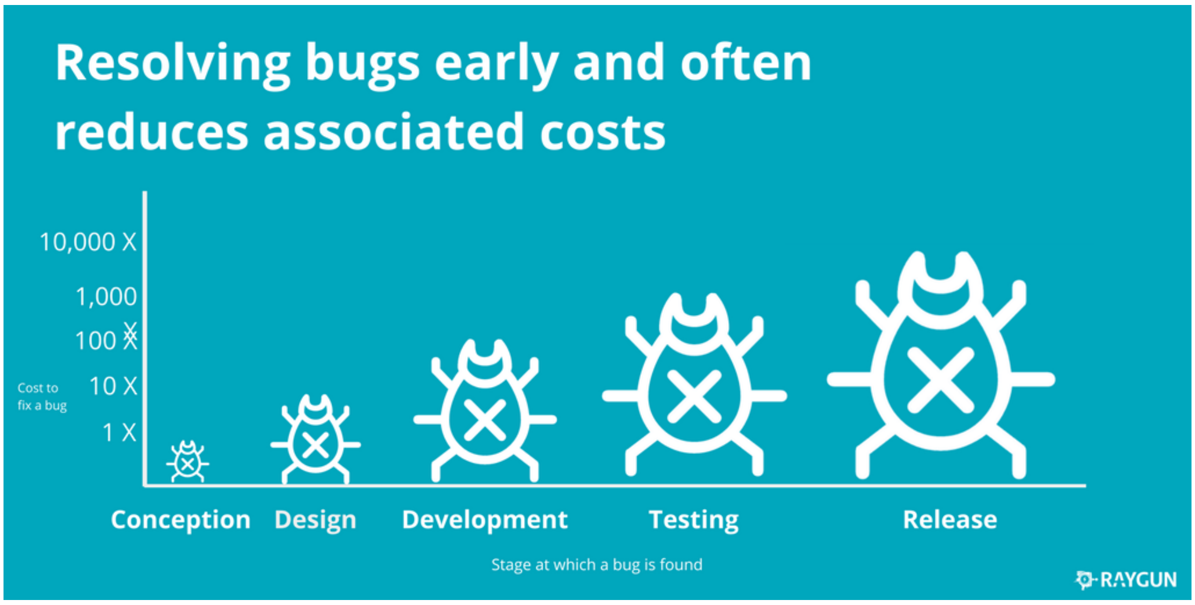


Figure 4: Cost of fixing defects in the various stages.

1.2.2 Types of Requirements

There are 3 main types of requirements:

- **Functional** - this set of requirements defines how the product will behave to meet the user's needs. Examples of such requirements are data manipulation, processing, calculations, etc that forms the basis of business rules. Use Cases are used to capture these requirements. They are also the main drivers of which software architecture the product is to use.

- **Non Functional** - these requirements support the functional requirements. From the end user's point of view, this would be the quality attributes of the product which includes performance, availability, usability and security and from a developer's perspective include reusability, testability, maintainability and portability. Non functional requirements (NFR) cater to the architectural needs of the overall system, whereas functional requirements cater to the design needs of the overall system.

  Non Functional Requirements can be measured but a measurement criteria must first be defined then a metrics can be applied. For example, the measurement of availability (or uptime) of a system can be quantified as a percentage of the uptime per week therefore if the system was running continuously without any downtime, the availability would be 100% for that week.

  Non Functional Requirements can either be drawn from product-oriented or process-oriented approaches. As the name implies, product-oriented approach concentrates on the product & its associated NFR aspects whereas the process-oriented approach concentrates on the process & its associated NFR aspects.
- **Interface Specification** - this set of specifications defines how the software product is to interact with other systems in order to work. For example, a retail shop billing system may need to interact with an inventory system to ensure that the warehouse keeps track of the number of stock at hand. The protocol of how these two systems interact with each other are captured in the interface specification document.

1.2.3 Steps of Requirements Engineering

Although the process may vary based on the application domain, a few generic steps are common across all types of software projects. The basic aims of these processes are to provide a mechanism to understand the user's needs, analyze the need, assess the feasibility, specify & validate the requirements & proposed solution and manage the requirements over the whole life cycle of the project.

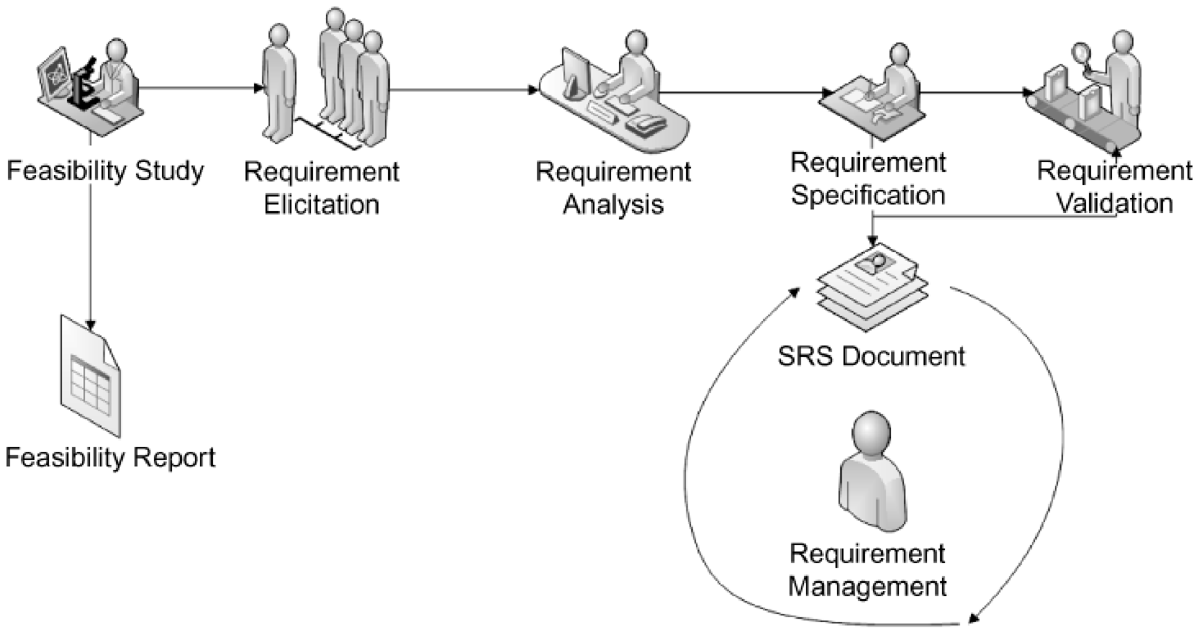


Figure 5: Requirements engineering processes.

From figure 5 above, we can see the 5 steps of requirements engineering:

1. **Feasibility Study** - this step determines the worthiness of the proposed system. It aims to answer to following questions:
   - Does the system contributes to organizational objectives?

   - Can the system be developed using the current available technology and within the specified time and budget?
   - Can the system be easily integrated with the other surrounding systems as required by the overall architecture?

   Therefore, feasibility can be classified into three broad categories:

   - **Operational Feasibility** - checks the usability of the proposed software within the organization.
   - **Technical Feasibility** - checks whether the level of technology required for the development of the system is available with the software firm, including hardware resources, software development platforms and other software tools.
   - **Economic Feasibility** - checks whether there is high Return on Investment (ROI) when investing in the software system. All the costs involved in developing the system, including software licenses, hardware procurement and manpower cost, needs to be considered while doing this analysis. Based on the ROI, stakeholders will make a decision on whether or not the proposed project should be undertaken.
2. **Requirements Elicitation** - the source (be it stakeholders or another system) for all the requirements are identified and then by using these sources, the user's needs and all the possible problem statements are identified. This process is normally iterative such that at the end of each iteration the client's needs may become clearer and new needs may emerge. The stakeholders involved during this phase include end users, managers, development and test engineers, domain experts and business analysts. The techniques used to get these requirements include interviews, brainstorming, task analysis, prototyping, etc.

3. **Requirements Analysis** - after we gathered most of the requirements in the elicitation phase, this phase is used to understand the requirements in detail. This phase refines the data and functional & behavioral constraints of the software, which acts as the input to the design of the software. Analysts will create models of the system to capture the data requirement & flow, functional processing, operational behavior and information content. The models are created using UML which will be covered in the Jupyter section during the lecture.

4. **Requirements Specification** - after the requirements have been gathered and analyzed, the next step is to put all these knowledge into a clear, concise and unambiguous manner so that the design and development teams can use it going forward. This phase will produce a **Software Requirement Specification (SRS)** document that is validated with the clients and/or users to confirm that all the requirements have been captured at this stage. This document may also be use to draft the basis of the contractual agreement between the client and the company. If you would like to read in detail about the IEEE standard for SRS documents, it can be found here.

5. **Requirements Validation** - this phase is carried out with the clients to ensure that their needs are captured completely, clearly and consistently. The validation step should provide enough confidence to all the stakeholders that the proposed system will provide all the features required and will be completed within the time and budget allocated for the project.

6. **Requirements Management** - this phase deals with how the existing requirements can be stored and tracked and how the changes to these requirements and also the introduction of new requirements can be handled during the requirements phase as well as throughout the life cycle of the project.

1.2.4 Introduction to Unified Modeling Language (UML)

In step 3 of the requirements engineering above, we stated that analysts would use models to represent the system's operational characteristics. In this section, we shall learn the different approaches used to model these characteristics. **Note** that this section (1.2.4) and next section (1.3 Data Design & Modeling) are mostly taught using Jupyter except for the theory sections.

The reason that we are learning these modeling approaches is because systems have been gradually getting bigger and more complex. Thus, the aims of these models are to fulfill the following objectives (step by step):

1. Clearly illustrate the user scenarios
2. Explain the functional activities in detail
3. Classify the design problems and their relationships
4. Define system behavior and class structure
5. Define data flow as it is transformed

The industry standard for visualizing, specifying, constructing, and documenting the artifacts of a software intensive system is called the Unified Modeling Language (UML). It is a general purpose modelling language used by analysts to create diagrams to portray the behaviour and structure of a system. UML have been an approved standard for modeling software based systems by the International Organization for Standardization (ISO) since 2005 and it was last reviewed in 2017. The version of UML is currently at 2.5. Figure 6 below shows an example of a UML class diagram.

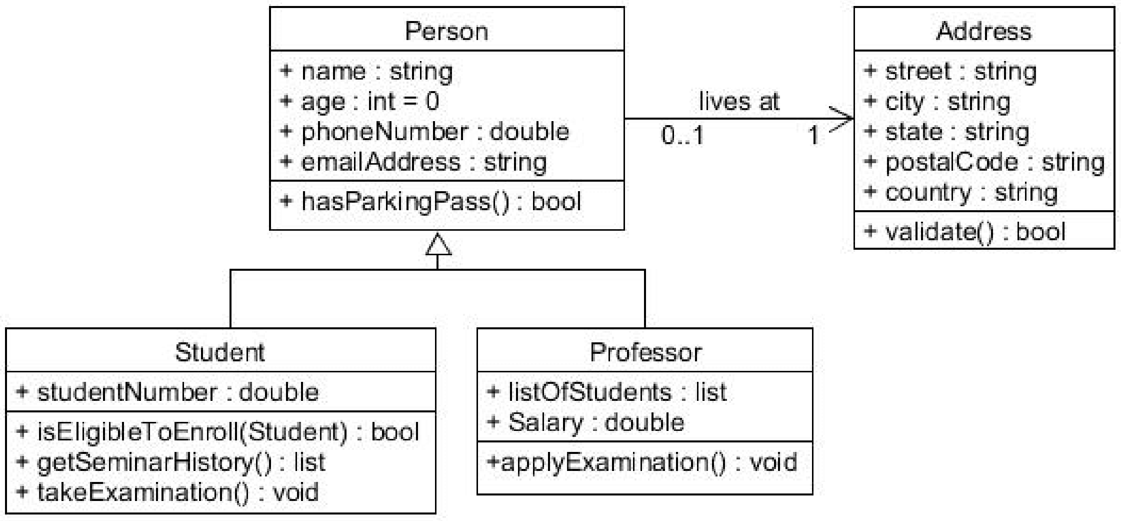


Figure 6: UML Class Diagram.

As we can see from figure 6 above, UML is closely linked to Object-Oriented Analysis and Design as objects are used to represent real world entities therefore diagrams created using UML can be broadly classified into 2 types:

- **Structural** - captures the static aspects or structure of the system. Static aspects are like how the classes are linked to each other.
- **Behaviour** - captures the dynamic aspects or the behaviour of the system. Dynamic aspects like how the users use the system, the different ways of interactions.

In addition, being closely linked to OOP, UML also share the same basic concepts:

- **Classes** - templates to create objects.
- **Objects** - in programing terms, an object has attributes and behaviours but in UML terms, an object can refer to a smaller part of a system such as a module.
- **Inheritance** - a child class inheriting the non private properties of the parent class.

- **Abstraction** - implementation details are hidden from the user.
- **Encapsulation** - protecting internal data from outside influence.
- **Polymorphism** - a way for functions to exist in different forms.

There are many types of UML diagrams but we will only be focusing on some of them as UML is a huge topic on its own. As stated by Grady Booch (one of the most important developer of UML), "For 80% of all software only 20% of UML is needed".

1.2.5 Use Case Diagrams

Please refer to Jupyter Notebook Lecture.

1.3 Data Design & Modeling

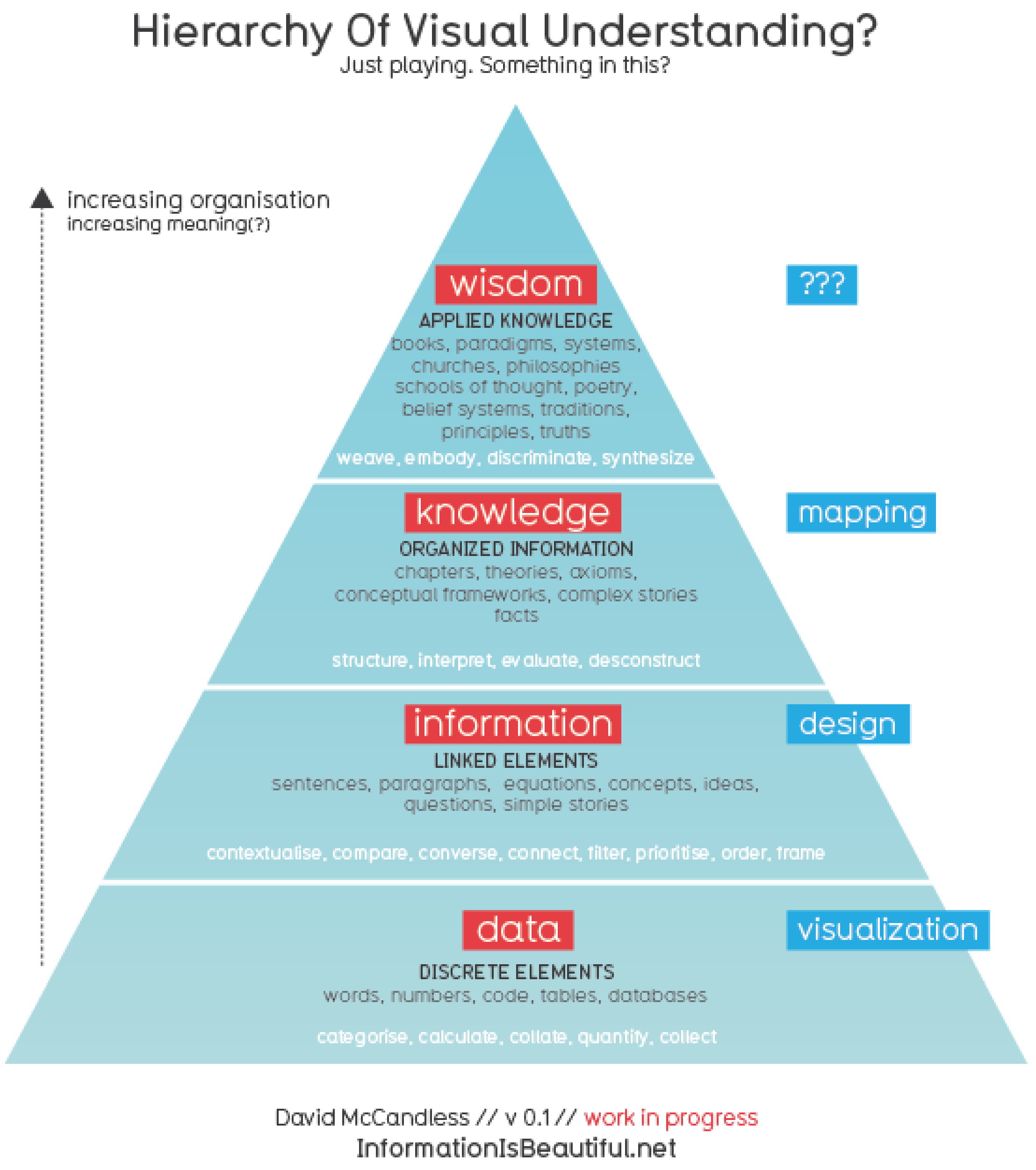
1.3.1 Data, Information and Knowledge

In everyday language we use knowledge all the time. Sometimes we mean "know-how", while other times we are talking about "wisdom". On many occasions we even use it to refer to "information". Part of the difficulty of defining knowledge arises from its relationship to two other concepts, namely data and information. These two terms are often regarded as lower denominations of knowledge, but the exact relationship varies greatly from one example to another.

Within more technologically oriented disciplines, particularly involving information systems, knowledge is often treated very similarly to information. It is seen as something one can codify and transmit and where IT plays a pivotal role in knowledge sharing. So how exactly do we differentiate these terms:

- **Data** - Facts and figures which relay something specific, but which are not organized in any way and which provide no further information regarding patterns, context, etc. Robert J. Thierauf (1999) defined data as "unstructured facts and figures that have the least impact on the typical manager."
- **Information** - For data to become information, it must be contextualized, categorized, calculated and condensed (Davenport & Prusak 2000). Information thus paints a bigger picture; it is data with relevance and purpose (Bali et al 2009). It may convey a trend in the environment, or perhaps indicate a pattern of sales for a given period of time. Essentially information is found "in answers to questions that begin with such words as who, what, where, when, and how many" (Ackoff 1999).
- **Knowledge** - Knowledge is closely linked to doing and implies know-how and understanding. The knowledge possessed by each individual is a product of his experience, and encompasses the norms by which he evaluates new inputs from his surroundings (Davenport & Prusak 2000).

These 3 terms are along the "wisdom" are often grouped together to describe a knowledge management systems but for the context of this course, "wisdom" is omitted as it is still currently debatable whether or not computing systems can be "wise". The terms can also be grouped into the Data, Information, Knowledge, Wisdom (DIKW) Pyramid which represents the enrichment of data at each stage. Refer to figure 7 on the next page.



Hierarchy Of Visual Understanding?
Just playing. Something in this?
increasing organisation
increasing meaning(?)
wisdom
APPLIED KNOWLEDGE
books, paradigms, systems, churches, philosophies schools of thought, poetry, belief systems, traditions, principles, truths
weave, embody, discriminate, synthesize
knowledge
ORGANIZED INFORMATION
chapters, theories, axioms, conceptual frameworks, complex stories facts
structure, interpret, evaluate, desconstruct
information
LINKED ELEMENTS
sentences, paragraphs, equations, concepts, ideas, questions, simple stories
contextualise, compare, converse, connect, filter, prioritise, order, frame
data
DISCRETE ELEMENTS
words, numbers, code, tables, databases
categorise, calculate, collate, quantify, collect
???
mapping
design
visualization
David McCandless // v 0.1 // work in progress
InformationIsBeautiful.net

Figure 7: DIKW Pyramid by David McCandless.

1.3.3 Data Flow Diagrams

Please refer to Jupyter Lecture Notebook for this section.

1.3.4 Entity Relationship (E-R) Diagrams

Entity Relationship (E-R) diagrams are used in the conceptual phase of database design. Its purpose is to specify the entities that are to be represented in the database, the attributes of the entities, the relationships among the entities, and constraints on the entities and relationships. E-R diagrams will be covered in greater detail in the Database module.

E-R modeling uses 3 main concepts:

- **Entity** - is a "thing" or an "object" in the real world that is distinguishable from all other objects. An entity always has a set of properties with values to uniquely identify it.
- **Attributes** - descriptive properties possessed by each member of an entity set. Each attribute stores similar information concerning each entity in the entity set but each entity may also have its own value for each attribute. Attribute can also be classified into *simple*, *composite*, *multivalued* and *derived*.

- **Relationship** - captures how entities are related to one another. Relationships are usually verbs such as assign, associate, or track and provide useful information that could not be easily found out by just the entity types.

Cardinality is mapped onto entities associated with via relationship sets. There are 3 main types of cardinalities: *One-to-One*, *One-to-Many* or *Many-to-One* and *Many-to-Many*.

E-R diagrams uses many different notations and the 2 most common ones are Chen notation and Crow's Feet notation.

1.3.5 UML Class Diagrams

Please refer to Jupyter Lecture Notebook for this section.

1.3.6 Other UML Diagrams

After learning some UML diagrams and other forms of system/work flow illustrations (data flow diagram, E-R diagram, etc), let's complete this section off with the full list of UML diagrams:

Structural UML Diagrams

1. **Class Diagram** - The most widely used UML diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing system's classes, their methods and attributes. Class diagrams also help us identify relationship between different classes or objects.
2. **Composite Structure Diagram** - A composite structure diagram is a UML structural diagram that contains classes, interfaces, packages, and their relationships, and that provides a logical view of all, or part of a software system. It shows the internal structure (including parts and connectors) of a structured classifier or collaboration (refer to figure 8 on the next page). It performs a similar role to a class diagram, but allows you to go into further detail in describing the internal structure of multiple classes and showing the interactions between them. You can graphically represent inner classes and parts and show associations both between and within classes.

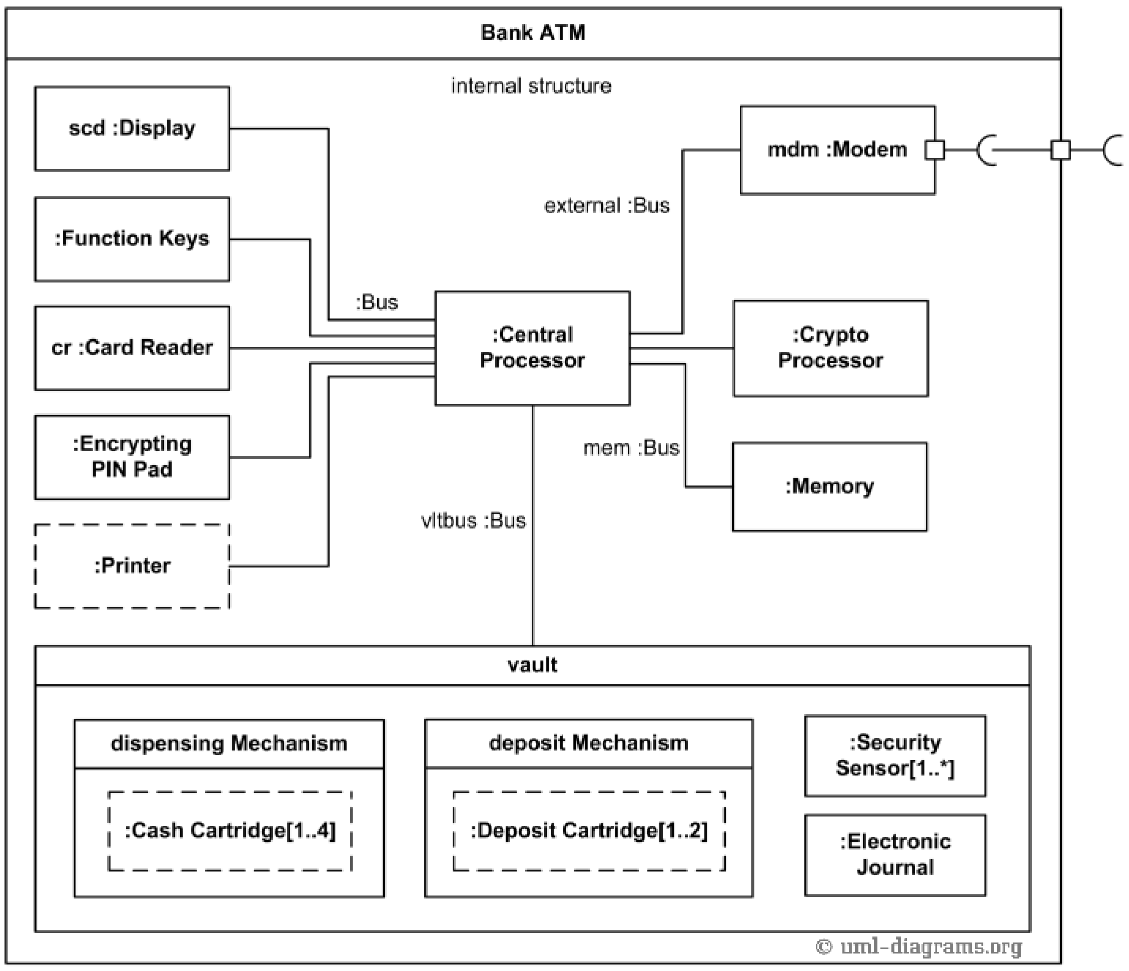


Figure 8: Composite Structure diagram of a Bank ATM.

3. **Object Diagram** - An Object Diagram can be referred to as a screenshot of the detailed state of the instances in a system and the relationship that exists between them. Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behaviour of the system at a particular instant in time (refer to figure 9 below). Thus, an object diagram encompasses objects and their relationships which may be considered a special case of a class diagram.

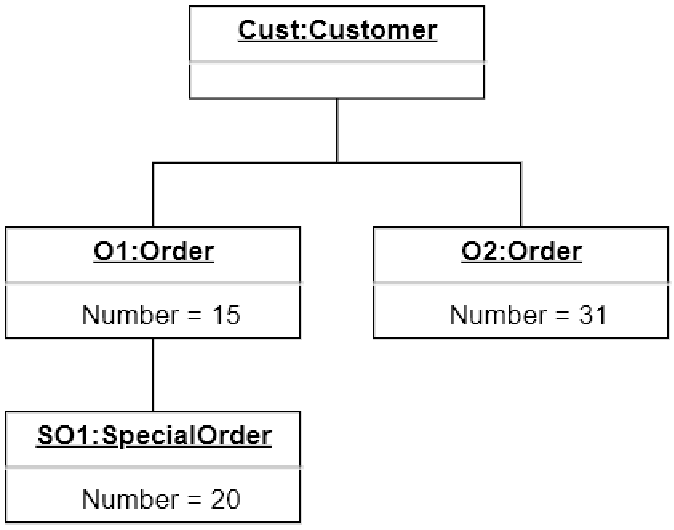


Figure 9: Order diagram of a Order Management System.

4. **Component Diagram** - Component diagrams are essentially class diagrams that focus on a system's components that often used to model the static implementation view of a system. They are used for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development (refer to figure 10 below). Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.

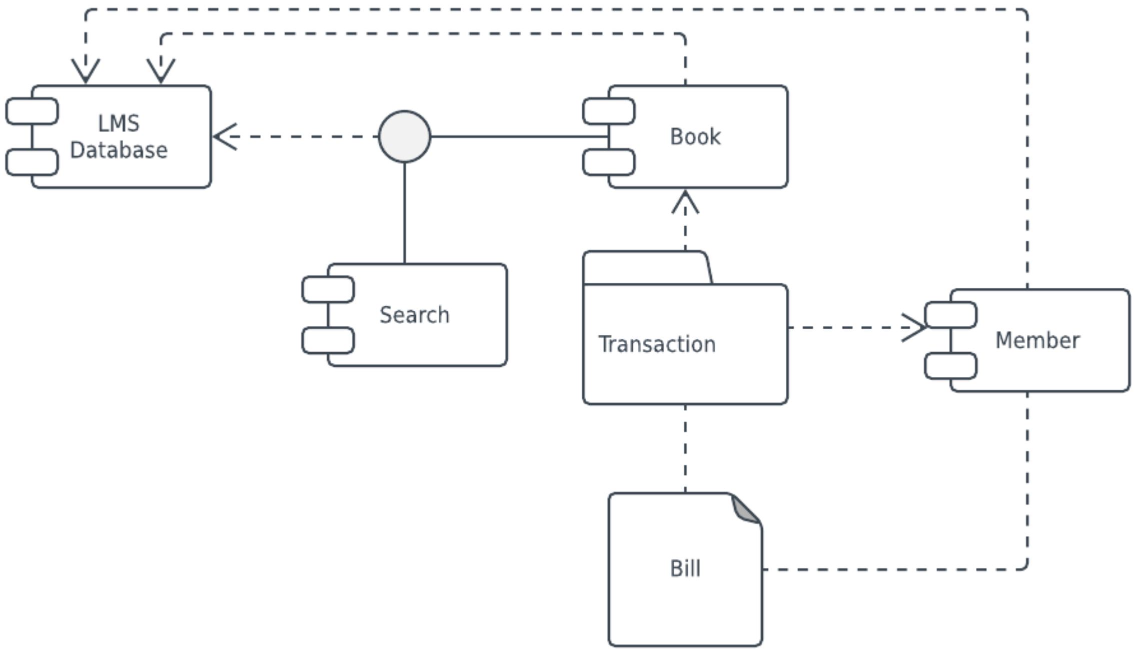


Figure 10: Component diagram of a Library Management System.

5. **Deployment Diagram** - Deployment Diagrams are used to represent system hardware and its software. It tells us what hardware components exist and what software components run on them. We illustrate system architecture as distribution of software artifacts over distributed targets. An artifact is the information that is generated by system software. They are primarily used when a software is being used, distributed or deployed over multiple machines with different configurations. Refer to figure 11 on the next page.

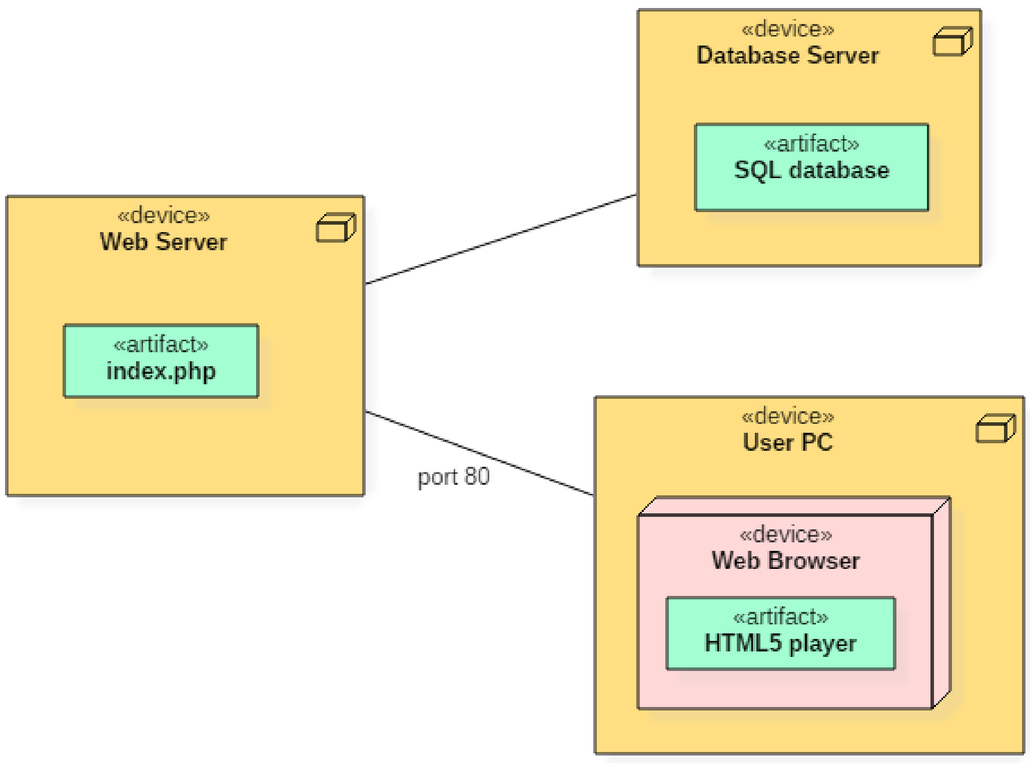


Figure 11: Deployment diagram of a HTML5 Video Player in a browser.

6. **Package Diagram** - We use Package Diagrams to show both structure and dependencies between sub-systems or modules. It simply shows us the different views of the system and the internal composition of its packages (refer to figure 12 below). Packages also help us to organize UML diagrams into meaningful groups and make the diagram easy to understand. They are primarily used to organize class and use case diagrams.

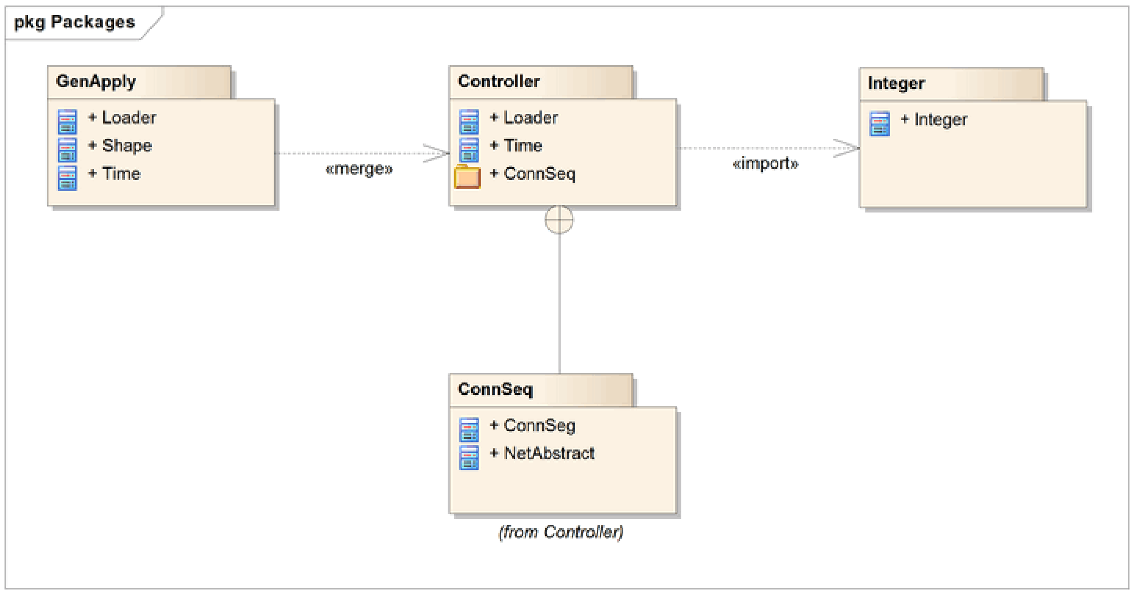


Figure 12: Package diagram example.

7. **Profile Diagram** - Is basically an extensibility mechanism that allows you to extend and customize UML by adding new building blocks, creating new properties and specifying new semantics in order to make the language suitable to your specific problem domain. Refer to figure 13 below.

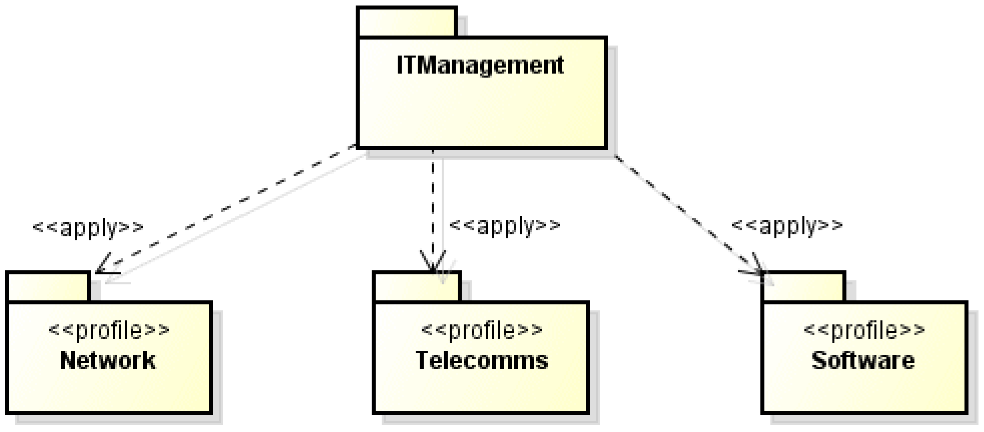


Figure 13: Profile diagram example.

Behaviour Diagrams

1. **State Machine Diagrams** - A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as **State machines** and **State-chart Diagrams** . These terms are often used interchangeably. So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli. Refer to figure 14 below.

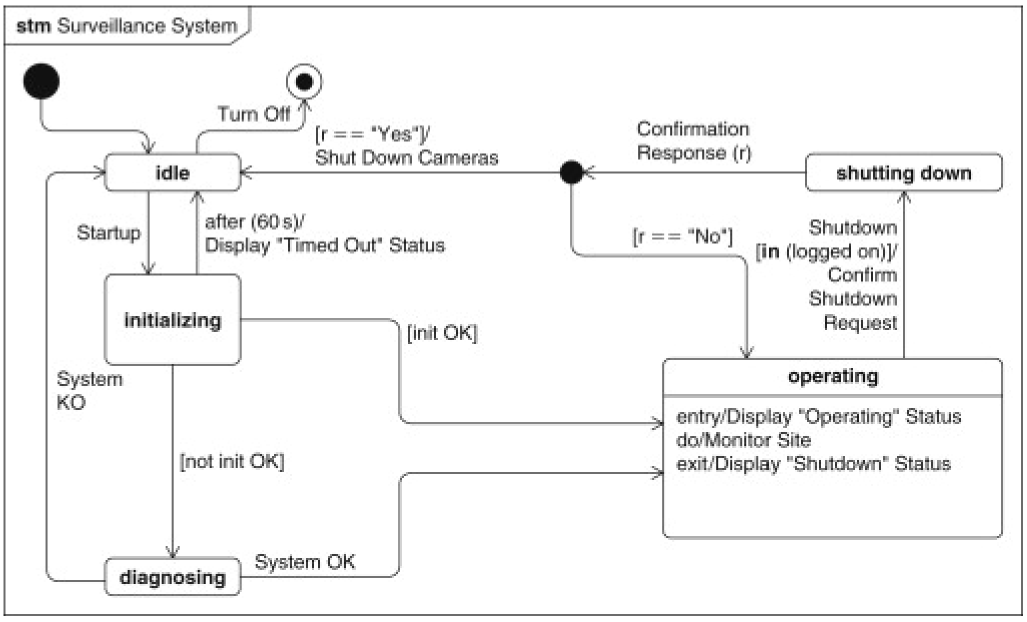

stm Surveillance System
Turn Off
[r == "Yes"]/ Shut Down Cameras
Confirmation Response (r)
idle
shutting down
Startup
after (60 s)/ Display "Timed Out" Status
[r == "No"]
Shutdown [in (logged on)]/ Confirm Shutdown Request
initializing
[init OK]
System KO
operating
entry/Display "Operating" Status
do/Monitor Site
exit/Display "Shutdown" Status
[not init OK]
diagnosing
System OK

Figure 14: State Machine Diagram of a Surveillance System.

2. **Activity Diagrams** - We use Activity Diagrams to illustrate the flow of control in a system. Think of it as an advance version of a flow chart thus it can be used to show the steps involved in an execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram. An activity diagram focuses on condition of flow and the sequence in which it happens.
3. **Use Case Diagrams** - Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the expected behavior of the system and its interaction with external agents (actors) not the exact method of making it happen (how). A key concept of use case modeling is that it helps us design a system from the end user's perspective without going into implementation details.
4. **Sequence Diagram** - A sequence diagram simply depicts interaction between objects in a sequential order that is the order in which these interactions take place. We can also use the terms "event diagrams" or "event scenarios" to refer to a sequence diagram. Sequence Diagrams are time focus and they show the order of the interaction visually by using the vertical axis of the diagram to represent time what messages are sent and when. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.
5. **Communication Diagram** - A Communication Diagram (known as Collaboration Diagram in UML 1.x) is used to show sequenced messages exchanged between objects. A communication diagram is an extension of object diagram that shows the objects along with the messages that travel from one to another. In addition to the associations among objects, communication diagram shows the messages the objects send each other. Refer to figure 15 below.

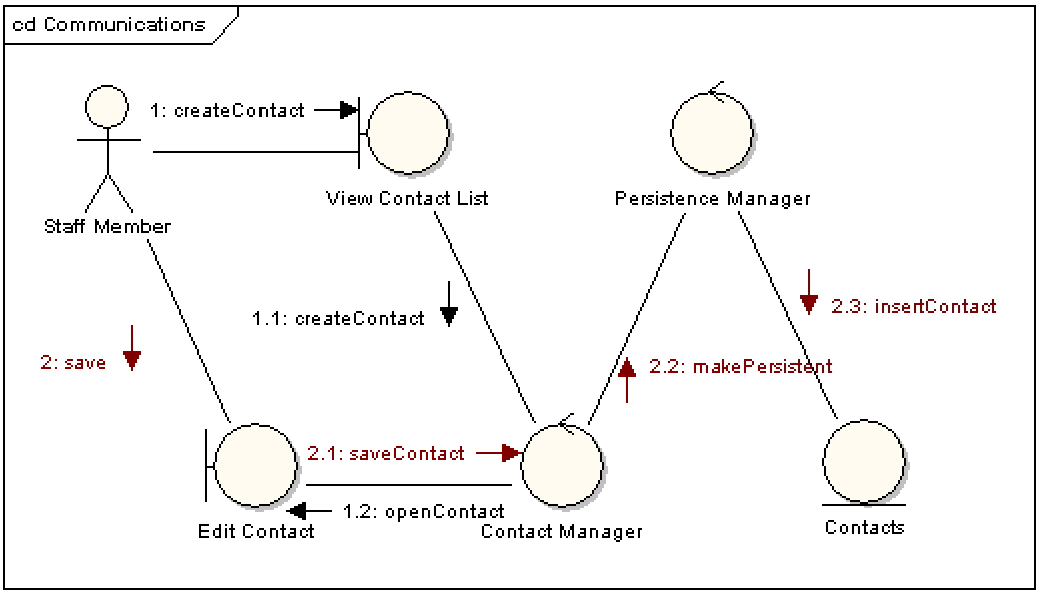


Figure 15: Communication Diagram.

6. **Timing Diagram** - Timing Diagram are a special form of Sequence diagrams which are used to depict the behavior of objects over a time frame. We use them to show time and duration constraints which govern changes in states and behavior of objects. Refer to figure 16 on the next page.

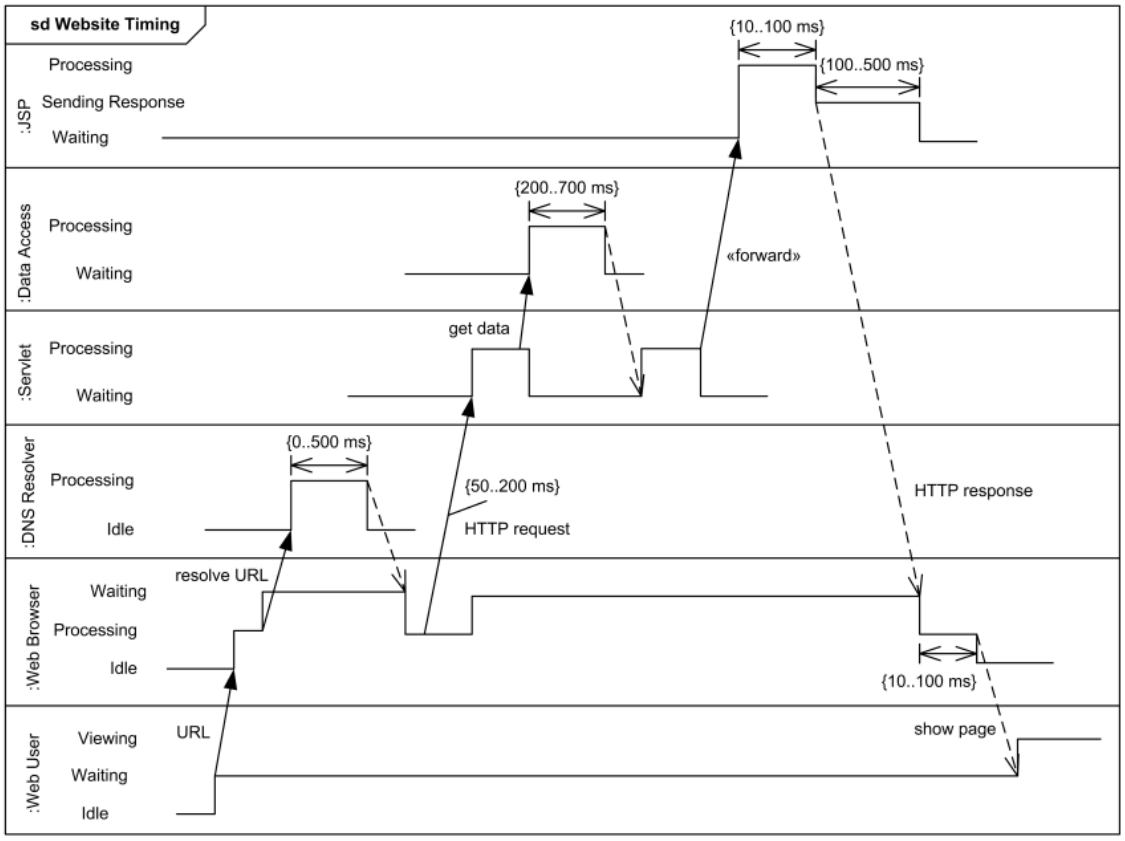


Figure 16: Timing Diagram of a Website.

7. **Interaction Overview Diagram** - An Interaction Overview Diagram provides a high level of abstraction an interaction model. It is a variant of the Activity Diagram where the nodes are the interactions or interaction occurrences. The Interaction Overview Diagram focuses on the overview of the flow of control of the interactions which can also show the flow of activity between diagrams (refer to figure 17 below). In other words, you can link up the "real" diagrams and achieve high degree navigability between diagrams inside an Interaction Overview Diagram.

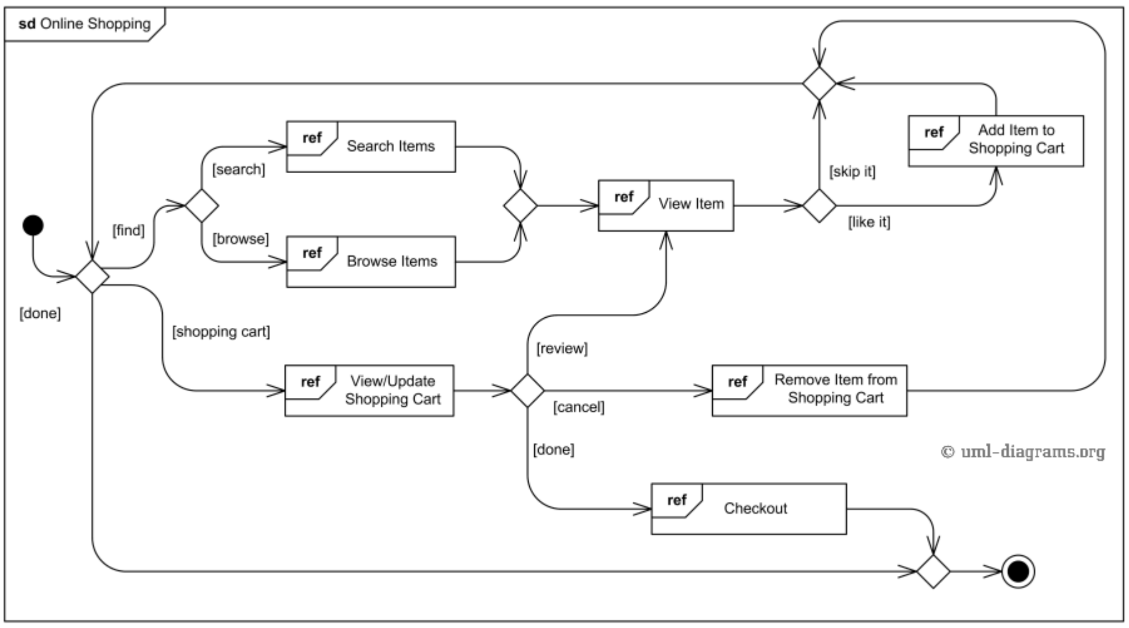


Figure 17: Interaction Overview Diagram for Online Shopping.

From the list, we can see that there are many types of UML diagrams but not all of them are widely used. Figure 18 below, shows the results of a UML usage survey done by the Universit'a di Genova in Italy. Interpret the results as follows:

- widely used, if it is ≥ 60% of the sources
- scarcely used if it is ≤ 40% of the sources

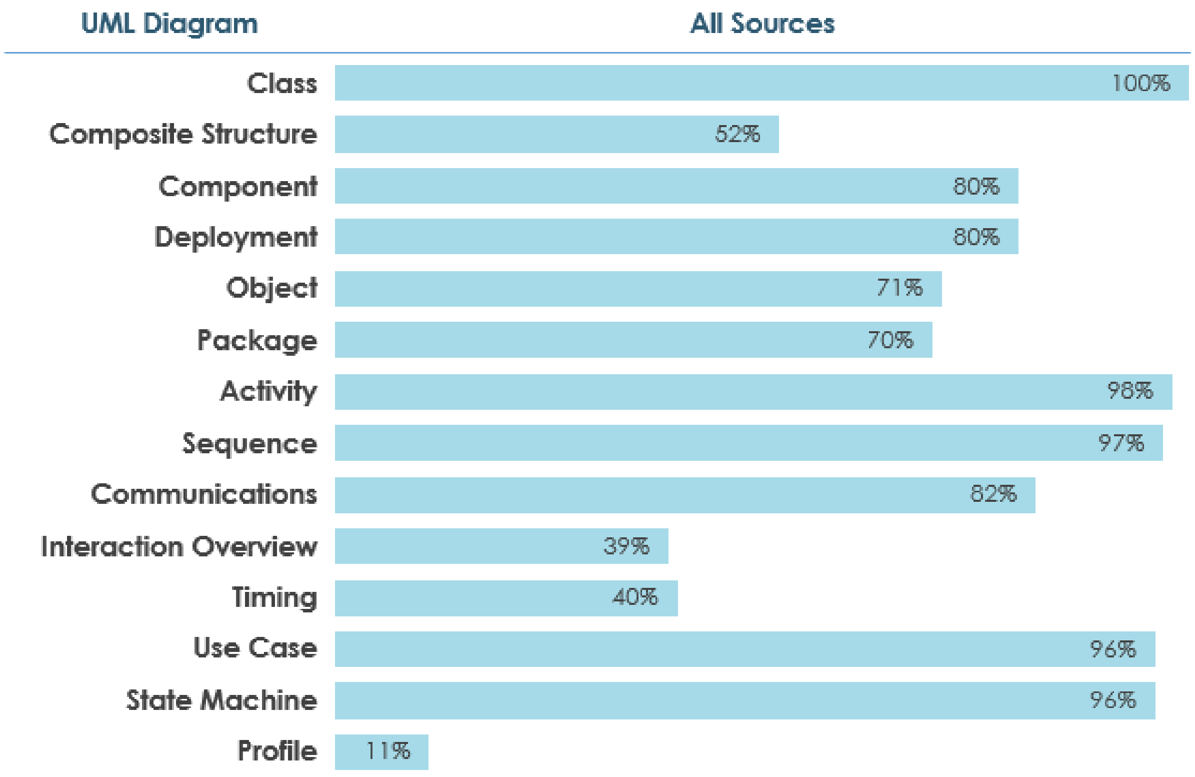


Figure 18: Popularity of the different types of UML diagrams usages.

1.4 Development

The development phase is about writing code and converting design documentation into the actual software within the software development process. This stage is generally the longest in the whole SDLC as it is the backbone of the whole process.

1.4.1 Development Tools

During development, there will be a need to setup tools to facilitate the development process, some of these tools are:

- Version control systems
- Build management tools
- Continuous integration, continuous delivery and continuous deployment

1.4.1.1 Version control systems

Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems are a class of software tools that is used for managing changes to program codes, documents or other collection of information. They are normally installed on a server in organizations. Changes are usually identified by a number or letter code (depending on organization). Each revision is associated with a timestamp, the person who made the changes and an optional description of the changes.

In general, these version control systems work by creating a **repository** (a stored area) where the system under development is uploaded to. Normally each software project has its own repository. Developers of the same team could then **check out** (aka download) a copy of the system under development then work on their assigned parts. This modified data is not immediately reflected in the repository, instead the changes would be needed to be **committed** (aka uploaded) to the repository for the changes to be made visible to everyone in the team. The local copy that developers work on is called the **working copy**.

The process of *checking out* from a repository is sometimes done via **branching**. Branching means that you diverge from the main line of development and continue to develop without messing with the main line. Once the work is done, the codes are committed and merged back into the main line. Often in large projects, merging with create **conflicts**. Conflicts happens when there are changes made to the same document and the system is unable to reconcile the changes. In this case, the developer must resolve the conflict manually either by combining the changes or by selecting one change in favour of the other.

There are several version control systems in the market and terminology between them can vary from system to system but below is a list of some of the most common terminologies that are common throughout.

- **Branch** - A set of files under version control may be *branched* or *forked* at a point in time so that, from that time forward, two copies of those files may develop at different speeds or in different ways independently of each other.
- **Checkout** - To *check out* is to create a local working copy from the repository. A user may specify a specific revision or obtain the latest.
- **Clone** - *Cloning* means creating a repository containing the revisions from another repository. This is equivalent to *push*ing or *pull*ing into an empty (newly initialized) repository.
- **Commit** - To *commit* (*check in*) is to write or merge the changes made in the working copy back to the repository. A commit contains metadata, typically the author information and a commit message that describes the change.

- **Conflict** - A conflict occurs when different parties make changes to the same document, and the system is unable to reconcile the changes. A user must *resolve* the conflict by combining the changes, or by selecting one change in favour of the other.
- **Head** - This refers to the most recent commit, either to the trunk or to a branch. The trunk and each branch have their own head, though HEAD is sometimes loosely used to refer to the trunk.
- **Merge** - A *merge* or *integration* is an operation in which two sets of changes are applied to a file or set of files.
- **Fetch, Push, Pull** - Copy revisions from one repository into another. *Pull* is initiated by the receiving repository, while *push* is initiated by the source. *Fetch* is sometimes used as a synonym for *pull*, or to mean a *pull* followed by an *update*.
- **Repository** - The *repository* (or "repo") is where files' current and historical data are stored, often on a server.
- **Resolve** - The act of user intervention to address a conflict between different changes to the same document.
- **Tag** or **Label** - A *tag* or *label* refers to an important snapshot in time, consistent across many files. These files at that point may all be tagged with a user-friendly, meaningful name or revision number.
- **Working Copy** - The *working copy* is the local copy of files from a repository, at a specific time or revision. All work done to the files in a repository is initially done on a working copy.

In the market, there are many types of version control systems but the 2 main ones are **Git** and **SVN**. Most of you may have heard of GitHub and GitLab, both are web-based Git repositories. While SVN, it is the Apache Subversion with TortoiseSVN as the client. Both are used for workflow and project management but the approach for version control used by each system is different (refer to figure 19 below). It is the role of the development team leader to decide with is the best system to use for the project but it is always good that the developers themselves know the difference between the 2 systems.

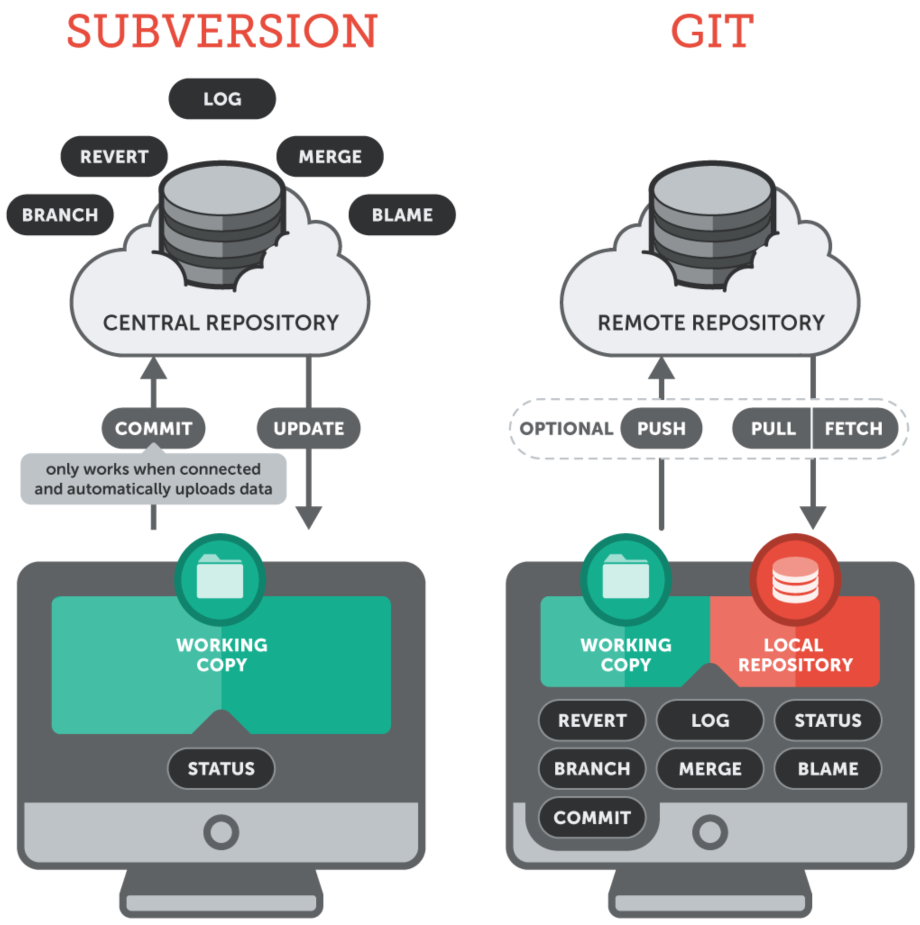


Figure 19: Main difference between Git and SVN.

	Git	SVN
Server architecture	Installed locally and acts as both server and client.	Requires a centralized server and user client.
Revisioning	Git is an **SCM (source code management)** tool. As such, it does not have a global revision number feature.	SVN is a **revision control system**. As such, it features a global revision number.
Repository cloning	Yes	No
Storage format	Metadata	Files
Storage requirements	Limited capacity to handle large binary files.	Can handle large binary files in addition to code.
Branching	Branches are references to a certain commit. They can be created, deleted, and changed at any time without affecting other commits.	Branches are created as directories inside a repository and when the branch is ready, it is committed back to the trunk.
Access control & permissions	Assumes all contributors have the same permissions to the entire codebase	Allows for granular permissions with read and write access controls per file level and per directory level.
Ease of use	Harder to learn	Easier to learn
Cryptographic hashing	To protect from repository corruption (due to network issues or disk failures) Git supports cryptographically hashed contents.	N.A.
License	GNU (General public license)	Open source under the Apache License
Change tracking	Repository level	File level

1.4.1.2 Build Automation

In the previous section, we have learnt about versioning control systems, those are used to manage changes during software development. Also within software development is call build automation. This is the process of creating the software build automatically. Steps to build a software includes:

- downloading the dependencies
- compiling the source code into binary code
- packaging the binary code
- running automated tests
- Deploying to production systems

This is where customer requirements like target platforms are important because development platform can different from customer's target platforms. As we know, in general there are mainly 3 types of operating systems, namely Windows, Linux/Unix and MacOS. Although MacOS is very similar to Linux, it is not 100% the same.

Historically, building programs were done using `makefiles` which originate from Unix like systems. These `makefiles` contains a set of directives which tells `make` how to compile and link a program. The principle is that files only requires recreating if their dependencies are newer than the file being created/recreated. Beware that if the project is large, *make* times can vary anywhere from 30 minutes to more than 8 hours. These `makefiles` are commonly used with complied languages like C/C++ or Java. Although Python interpreter does produce a `.pyc` file (compiled bytecode) which is then used by the Python virtual machine for execution.

These repetitive tasks were then brought together using build automation utilities such as Ant (Java), Cabal (Haskell) or Rake (Ruby). These utilities are broadly classified into 2 types: task-oriented or product-oriented. Task-oriented tools describe the dependency of networks in terms of a specific set task and product-oriented tools describe things in terms of the products they generate. These build automation utilities can setup separately on an automation server where required.

Build automation is considered the first step into moving towards implementing a continuous delivery system.

1.4.1.3 Continuous Integration (CI), Continuous Delivery (CD) and Continuous Deployment (CD)

At this point, we have read about and experienced most of the manual processes of building a program individual. However in organizations, programs are rarely developed with a single person, they are developed in teams. Because of this, systems such as *Continuous Integration*, *Continuous Delivery* and *Continuous Deployment* were developed to streamline the build to deployment process.

Continuous Integration and Continuous Delivery/Deployment is the practice of producing workable software in short cycles. Continuous Integration means to merge all developers' working copies to a shared mainline several times a day then building it. With continuous delivery, the delivery process is manual but with continuous deployment, the delivery process is automated.

These systems applies the following set of principles for it to work:

1. **Code is maintained in a repository** - the project's source code and dependencies are all placed in a repository. The convention is that the system should be buildable from a fresh checkout and not require additional dependencies. The mainline should be the location for the working version of the software.
2. **Automate the build** - using a single command, the system should be able to compile the source code. The build script not only compiles the the source codes, it also generates the documentation, web pages, statistics (build and tests) and distribution files (eg: `.exe` or `MSI` files for windows).
3. **Built self-testing** - after the code is built, tests (like regression tests) should be executed to confirm that the new build behaves as it should.
4. **Everyone commits to the mainline everyday** - committing code regularly can reduce the number of conflicts in the codes. For example, resolving conflicts in a week-old code is far more difficult than resolving conflicts in a day-old code especially for sections of codes that are frequently modified.
5. **Every commit to the mainline should be built** - The systems should built commits to the current working version to verify that they integrate correctly. This can either be done

manually or through a daemon that monitors the versioning control system for changes and then triggering the build process.
6. **Every bugfix commit should have its own test case** - whenever a bug is fixed, a test case should accompany it so that every subsequent future build would be checked for the reoccurrence of that same bug, this is called *regression*.
7. **Build should be fast** - builds needs to be completed fast so that any integration problems can be quickly identified.
8. **Test the system on a clone of the production environment** - more often than not, production environment differs from both development and test environments therefore is it important to test the system under development in a simulated production environment. In terms of software systems, this is normally done using containerization software like Kubernetes and/or Docker where full software environments can be simulated with Docker without the need of simulating the accompanying hardware and then managed with/without Kubernetes.
9. **Easily accessible latest deliverables** - this allow the stakeholders and testers to get the latest builds early for testing. This reduces the development cost as fixing errors early in the build is better than at before production launch. Developers are also encouraged to regularly update their local working copy from the repository every start of the day.
10. **Everyone can see the results of the latest build** - build dashboards can be used to show everyone the status of the builds therefore any broken builds can be spotted immediately.
11. **Automate deployment** - most CI systems allows execution of scripts after a build finishes thus it is possible to write deployment scripts to deploy the application to the live testing server for testing.

Both CI and CD systems are uses the notion of a pipeline. Jenkins, Microsoft Azure, Amazon Web Services and Google Cloud Platforms are some well known CI/CD systems. Each of these systems allows the user (also called DevOps) to create pipelines where *nodes* can be added to define the various stages of the build, test and deploy stage (refer to figure 20 below).

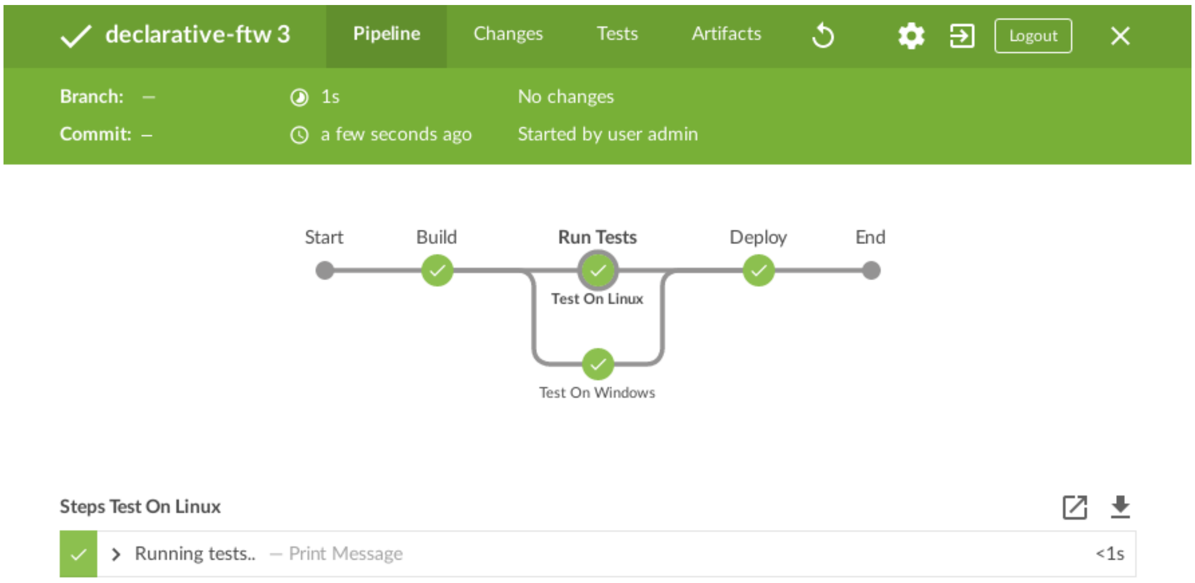


Figure 20: Jenkins pipeline with Declarative Pipeline 1.2.

All of these systems will have a propriety pipeline syntax that uses their propriety language or YAML. This pipeline syntax is used for each node to tell the pipeline how to run the node, what to do after executing the node and if there are reports (produced from executing the nodes), where it is stored so that the pipeline can retrieve it and display it.

The benefits of CI/CD systems are:

- integration bugs are caught and fixed early resulting in savings on both time and money over the lifespan of the project.
- avoids last minute chaos at release dates especially during integration.
- not a large developmental time lost when reversal of code base due to bugs as regular updates have been done.
- constant availability of a "current" build for testing, demo, or release purposes.
- immediate feedback when build fails a testing node.
- frequent releases enables the development team to get faster feedback from the clients on whether the development is going in the right track.
- higher level of customer satisfaction due to their frequent involvement in the development process.

The downsides of CI/CD systems are:

- constructing project specific automated tests takes a considerable amount of work.
- can be cumbersome or not required when the project is small or using untestable legacy codes.
- value added depends on the quality of tests and how testable the code really is.
- larger teams means that new code is constantly added to the integration queue, so tracking deliveries (while preserving quality) is difficult and builds queueing up can slow down everyone.
- with multiple commits and merges a day, partial code for a feature could easily be pushed and therefore integration tests will fail until the feature is complete.
- differences in the environments used for development, testing and production can result in undetected issues slipping to the production environment.
- documentation and in-process reviews can be difficult to achieve because of the continuous integration nature.
- not all tests can be done with automation, some tests still need humans to verify thus slowing down the delivery pipeline.

1.4.2 Test-driven Development (TDD)

In the previous section, we have read about 1 type of software development approach, in this section, we will learn about another type. Test-driven development uses an approach where the unit tests (are tests that test 1 component at a time) are developed **before** the coding phase. These tests are used specify and validate what the code will do.

1.4.2.1 Process

The process is as follows (based off the book *Test-Driven Development by Example*). Refer to figure 21 and the process list below:

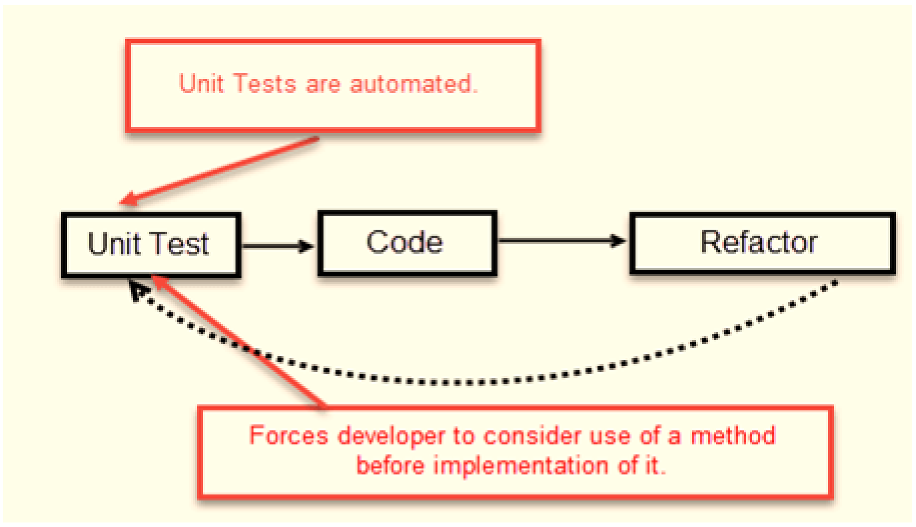

Unit Tests are automated.
Unit Test
Code
Refactor
Forces developer to consider use of a method before implementation of it.

Figure 21: Test Driven Development representation.

1. **Add a test** - every new feature begins with writing a test, The test defines a function or an improvement of a function (should be detailed). This can be done via Use Cases to detail the requirements and exception conditions or even a modified version of an existing test. **Important difference** is that the developer focus on the requirements **before** writing the code.
2. **Run all tests and see if any new test fails** - validates that the test framework is working properly and the new codes are fine because it passes all the new tests.
3. **Write the code** - code written in this step can be messy because it is written to pass the tests. Codes however, must not be written beyond the functionality of the tests.
4. **Run tests** - if all test cases pass, the programmer can be confident that the new code meets the test requirements and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.
5. **Refactoring** - because in step 3 the codes are written imperfectly, in this step, the codes are cleaned up. Codes are moved to where it is more logically belongs, duplication codes must be removed and structures like objects, classes, models, variables and functions should clearly represent their current purpose and use, as extra functionality is added. Design patterns are implemented as well. Variables that are used to trigger a pass in step 3 must definitely be removed.
6. **Repeat** - each time a test or test set is created, the cycle is repeated. Continuous Integration helps by providing revertible checkpoints.

1.4.2.2 Good practices

Test Structure
Having an effective layout of a test case ensures that all required actions are completed, this improves the readability of the test case and smooth the flow of execution. Consistent structure also helps in building a self-documenting test case. Test Case generally have the following layout:

1. **Setup** - Make sure that the system under test is set up to the correct state to start the test.
2. **Execution** - Trigger the system under test to perform the target behaviour and capture all the output.
3. **Validation** - Ensure that the results of the tests are correct. Results may include explicit outputs captured during execution or state changes in the system under test.

4. **Cleanup** - Restore the system under testing to the state before the tests. This would allow other tests to run immediately after this test.

Test case should also be small and simple enough such that it runs fast, it is independent (must not depend on result from previous tests), repeatable, self-validating and thorough.

Spell Out The Rules
After the test structure has been settled, organize the tests in a more logical pattern and make sure the testing tools align with these rules or conventions. Bring everyone (including new hires) in the team, up to speed with the testing conventions and existing knowledge as well.

Treat Implementation Separately
Generally there would be 2 source directories: implementation and testing. Some projects will have more than these 2 but separation must be maintained between implementation and test codes. This is to reduce the chances of accidentally packaging tests with production binaries.

Naming of Test Classes
If test automation is used, test classes should be named similar to the implementation class, for example, if the implementation class is `magicmix`, the test class should be `magicmix_test`. This is to help identify the classes or methods that are being tested especially when the project is big.

1.4.2.3 Refactoring

This is a process of editing and cleaning up previously written code without changing its external/intended behaviour at all. This is done to improved code readability and reduced complexity. These can improve the code's maintainability and create a simpler, cleaner, or more expressive internal architecture or object model to improve extensibility. It is also used for optimization of programs.

In TDD, code refactoring is always done after the unit tests because the unit tests are used to ensure that the components still behave as expected. Refactoring is then carried out in an iterative cycle of making a small code transformation, testing for correctness then making another small code transformation. Any time a test fails, the change is undone and repeated in a different way, that is why the tests must execute very quickly.

Some best practices of refactoring are as follows:

- **Refactor first before adding or fixing anything** - before any new features, fixing a bug or code reviews, refactor the existing code. This will reduce the amount of rework required in the future.
- **Plan your refactoring project and timeline carefully** - think about you overall goal. What changes are to be made? It is just a change of variable names, code optimization or full clean-up? Because code refactoring is also done within the development time frame, make sure that enough time is allocated.
- **Test Often** - refactoring can mess things up by creating bugs or sudden unforeseen changes in functionalities of the software. This is why testing throughout the refactoring process is imperative.
- **Focus on progress, not perfection** - all code will one day become legacy code thus refactor with the mindset of an ongoing maintenance project.

1.4.2.4 Limitations

Unit Tests
Since unit tests are mainly used for TDD, there is still not enough tests being conducted. This can be seen software that incorporates user interfaces that works with databases that requires a specific network configuration. Unit tests would test the individual components but it would have to fakes and mocks the represent the outside world.

Unit tests are also mostly created by the developers and thus can still possess blind spots. Requirements can also be misinterpreted leading to wrong unit tests being written.

False Sense of Security
Constantly passing unit tests can lead to a false sense of security resulting in fewer other tests like integration, compliance and exploratory testing.

Fragile Tests
These tests are tests that are badly written because they require many predefined inputs. These test are prone to failure and expensive to maintain. The goal is to write tests such that the inputs are separated from the test cases that they run.

1.5 Testing, Deployment and Maintenance

1.5.1 Testing

The process of testing is to determine the correctness of software under the following:

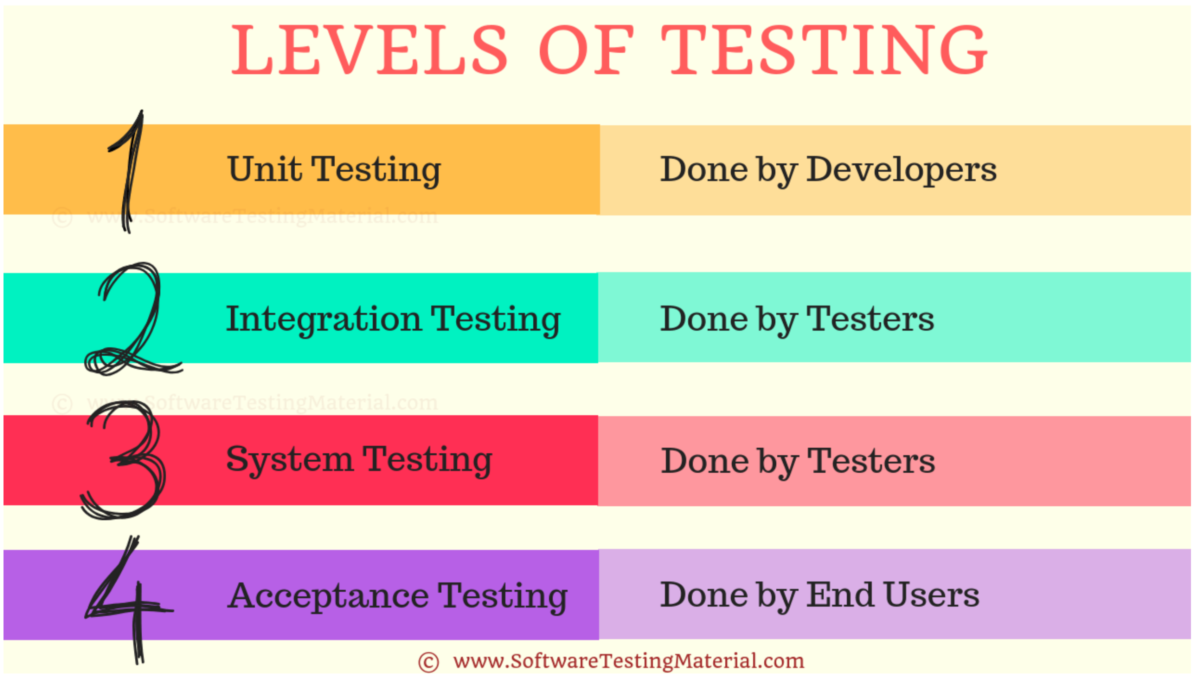
- meets the requirements that guided its design and development
- responds correctly to all kinds of inputs
- performs its functions within an acceptable time
- is sufficiently usable
- can be installed and run in its intended environments
- achieves the general result its stakeholders desire

Since there are infinite number of tests that can be executed on any component, software testing generally uses a strategy to select feasible tests for the available time and resources given. As seen in the previous section, testing is an iterative process where a bug is fixed then retested to find other deeper bugs or any new bug created from the fix.

There are different approaches to testing:

- **static, dynamic and passive testing** - static testing involves the syntax testing, it is normally done by interpreters or compliers. Dynamic tests are done when the program is running, for example, using the debugger to tests the inputs of a program at runtime. Passive testing is to make sure that the system behaves as expected when there is no interaction with it (typical of programs that run over a network).
- **Exploratory testing** - done manually and requires a human to test the software product. This tests normally run in parallel throughout the project.
- **"Box" testing** - traditional consisting of *white* and *black* box testing where *white* box testing means that the testers knows the inner workings of the program (the actual codes) and *black* box testing, where the testers do not know the internal implementation of the program, they are only aware of what the software is supposed to do. *Grey* box testing has been added to the mix where the testers have the knowledge of the internal workings of the program but the tests are designed from the black box point of view.

There are also 4 levels of testing (as can be seen on figure 22 below).



LEVELS OF TESTING

1	Unit Testing	Done by Developers
2	Integration Testing	Done by Testers
3	System Testing	Done by Testers
4	Acceptance Testing	Done by End Users

Figure 22: 4 Levels of Testing.

1. **Unit tests** - are used to verify the functionality of individual sections of the source code. Normally written by developers using the the *white-box* approach.
2. **Integration testing** - the process of testing the connectivity or data transfer between a couple of unit tested modules. Normally done in batches as the different components are being integrated to work as a system.
3. **System testing** - after integration testing, this test checks the system as a whole to ensure that it is performing up to its requirements.
4. **Acceptance testing** - largely involves testing from the end users so that it can be delivered and as the final product soon. Common tests include Alpha and Beta testing.

There are many types of tests, the common ones are listed in the table below

Type of Test	**Description**
Installation testing	Tests whether or not the software can be installed on the client's target systems.
Compatibility testing	Tests whether or not the software can be used with other application software or operating systems or target environments that is vastly different from the original target platform. For example, an application developed originally for a the desktop is now required to be transformed into a web application.
Smoke tests	Tests to see if the dependent systems and/or system itself is ready for testing.
Regressing testing	Focuses on finding defects after a major code changes has occurred. Especially used to test systems for lost of features or resurfaced old bugs.
Alpha testing	Can be simulated or actual operational testing done by potential users/customers or an independent test team at the developers' site.
Beta testing	Done after Alpha testing where tests are done by a limited audience outside of the development team. Can also include real users of the system.
Destructive testing	Attempts to cause the software or a sub-system to fail. Tests for the robustness of the system in the event of invalid or unexpected inputs.
Software Performance testing	Used to determine how a system or sub-system performs in terms of responsiveness and stability under a particular workload. **Load/Stability testing** is part of this whereby the system is tested to see how well it can perform under a certain load. Load refers to large quantities of data or large amount of users. **Volume testing** tests how the system handles sudden increases of certain components (esp files). **Stress testing** tests how reliability the system is under unexpected or rare workloads.
Usability testing	Tests whether or not the user interface of the systems is easy to use and understand.
Accessibility testing	Tests whether or not the software is accessible to people with disabilities.
Security testing	Used for systems that handles confidential data to prevent hacking.
Internationalization and localization	Tests whether or not the software can be used with different languages and geographic regions.
Concurrent testing	Tests whether or not the software behaves and performs under a concurrent computing environment, generally used for systems that are required to perform concurrent processing.

1.5.2 Deployment

Defines the steps or procedures required to be carried out by either the developer, customer or both sides to ensure that the software is ready for use. These processes are general customized for the specific requirements of the customer's target platform therefore there are no predefined steps that can be listed down.

However, deployment consist of these activities:

- **Release** - prepares the system for use in an production environment. This involves determining the resources required for the system to operate within tolerable performance and planning and/or documenting subsequent activities of the deployment process.
- **Installation and Activation** - depending on the complexity of the system, this process can be just to setup an automatic start up option to requiring the users to make selections on which components to install. Activation is the activity of starting up the executable component of software for the first time, it is not to be confused with the activation of the software license).
- **Deactivation** - the inverse of activation, it means to shutdown any already-executing components of a system. This is normally done when the system requires an update.
- **Uninstallation** - the inverse of installation where the system is removed from the customer's machine because it is no longer required. Internal registers may also be reconfigured to allow a full uninstallation of the system's dependencies.
- **Update** - replaces an earlier version fully or part of the software system with a newer release. Processes involved are deactivation then installation.
- **Built-in update** - procedures for installing updates are sometimes built into some software systems to facilitate automatic updates. For example, Windows updates.
- **Version Tracking** - helps tracks the software version on the user's machine and to prompt an update installation where necessary.

1.5.3 Maintenance

The modification of the software product after delivery to rectify faults, improve performance or other customer related modifications. Because change is inevitable, mechanisms must be developed for evaluation, controlling and making modifications. The purpose is to preserve the value of software over the time.

Since maintenance is an integral part of any software product, maintenance plans have to be carefully crafted during the development phase. It should specify how users will request modifications or report problems, how much resources & cost should go into software maintenance, scope of maintenance and how long the support should last. On average, software maintenance can cost more then 50% of the whole SDLC (refer to figure 23 on the next page).

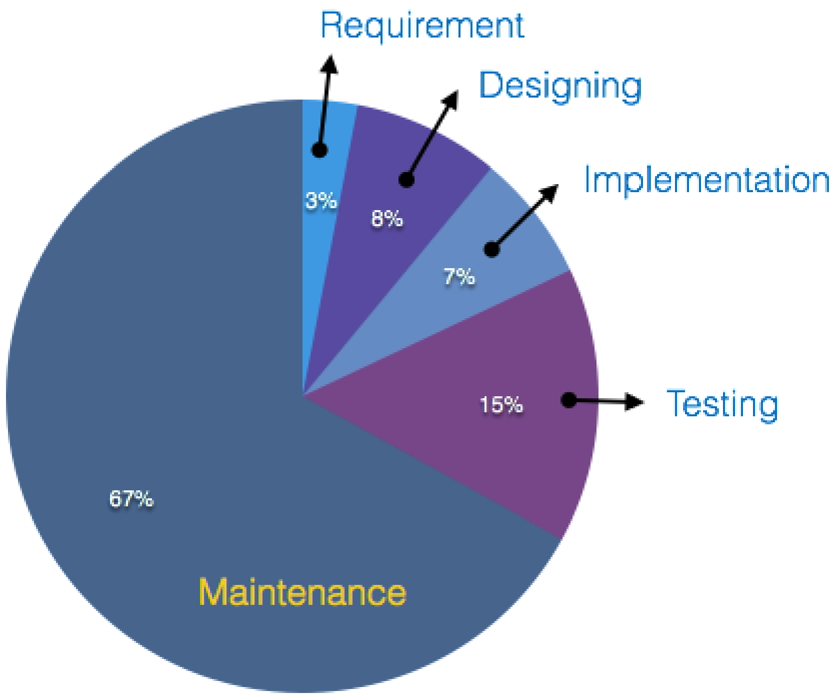


Figure 23: Maintenance cost chart.

Factors that contribute to the maintenance cost are as follows (from TutorialsPoint):

- The standard age of any software is considered up to 10 to 15 years.
- Older software, which were meant to work on slow machines with less memory and storage capacity cannot keep themselves challenging against newly coming enhanced software on modern hardware.
- As technology advances, it becomes costly to maintain old software.
- Most maintenance engineers are newbie and use trial and error method to rectify problem.
- Often, changes made can easily hurt the original structure of the software, making it hard for any subsequent changes.
- Changes are often left undocumented which may cause more conflicts in future.
- Structure of Software Program
- Programming Language
- Dependence on external environment
- Staff reliability and availability

There are 4 types of software maintenance:

- **Corrective Maintenance** - This includes modifications and updates done, in order to correct or fix problems, which are either discovered by users or concluded by user error reports.
- **Adaptive Maintenance** - This includes modifications and updates applied to keep the software product up-to date and tuned to the ever changing world of technology and business environment.
- **Perfective Maintenance** - This includes modifications and updates, done in order to keep the software usable over long period of time. It includes new features, new user requirements for refining the software and improve its reliability and performance.
- **Preventive Maintenance** - This includes modifications and updates to prevent future problems of the software. It aims to rectify problems, which are not significant at this moment but may cause serious issues in future.

1.5.4 Symptoms of poor design

With software, the word "design" can mean 2 different things:

- the specifications of modules and their interrelationships are known as design.
- a design pattern is a general, reusable solution to a commonly occurring problem within a given context

Regardless of which design is used, recognizing the good from the bad is not always straightforward. The following table highlights the key points.

Characteristics	Good Design	Bad Design
Change	Changing one part of the system does not always require a change in another part of the system.	One conceptual change requires changes to many parts of the system.
Logic	Every piece of logic has one and one home.	Logic has to be duplicated.
Nature	Simple	Complex
Cost	Small	Very high
Link	The logic link can easily be found.	The logic link cannot be remembered.
Extension	System can be extended with changes in only one place.	System cannot be extended so easily.

1.5.5 Documentation

Documentation is the document that accompanies computer software or is embedded in the source code. The main purpose is of documentation is to explains how the software operates or how to use it. There are different types of documentation depending on the audience:

- **Requirements** - Typically created in the beginning of a software development project. Has the goal to clearly and precisely specify the expectations in regards to the software being created. May include functional requirements, limitations, hardware or software requirements, compatibility requirements, and so on. For everyone involved in the production of the software.
- **Architecture/Design** - Defines the high-level architecture of the software system being created. May describe the main components of the system, their roles and functions, as well as the data and control flow among those components. Database design documents belongs in this type of documentation. For everyone involved in the production of the software.
- **Technical** - Documentation of the software code, algorithms, application programming interfaces (APIs). Written for the technical audience like software developers, testers and end users. Comments written in source codes using a certain format can be converted to technical documentation using the tools such as Doxygen. These tools will define the commenting format so that it will be able to generate reference manuals in either text or HTML forms.
- **End User** - Manuals for the end-user, system administrators and support staff. These documentation simply describe how a program is used. These documentation also includes a "Troubleshooting" section to provide some troubleshooting assistance to common problems. It is very important for user documentation to not be confusing and up to date as

much as possible. Typical formats include tutorials, thematic (grouped by chapters) and an alphabetically list or reference.

1.5.6 Retrospective Process Improvement

These sessions are held during the production process (normally after sprints) where teams come together to reflect on their performance for constant improvement. Sprints are short periods of time (about a week or 2) where a team works to complete a set number of tasks. These sessions are normally used for teams who adopt the Agile methodology of software development. This methodology will be describe in further detail in the document *Agile Software Development*.

The purpose of this process is to find out if things can be done differently from the previous to help improve performance. A retrospective session should be a safe space for people involved to share their honest feedback on what’s going well, what could be improved, and generate a discussion around things that should change next time around. The person leading these sessions acts as a moderator and should be careful of varying levels of emotional baggage that themselves or the people can bring to the session.

A well run retrospective session can yield:

- self-improvement of each team member's work process and/or role.
- identify areas for team wide improvement and to provide a platform to talk about values or the results that the whole team can work on, moving forward.
- insight and collaborative feedback into what is going well within the team, what is not and what can be improved.

References

1. "How to Define Stakeholders for Your Software Development Project", Concepta, https://www.conceptatech.com/blog/how-to-define-stakeholders-for-your-software-development-project
2. C Geetha, C Subramanian, S Dutt, 2015, Software Engineering, Pearson Education
3. E C Foster, 2014 Software Engineering: A Methodical Approach, Apress
4. "Unified Modeling Language (UML) | An Introduction", 2019, Geeks for Geeks, https://www.geeksforgeeks.org/unified-modeling-language-uml-introduction/
5. "Defining Knowledge, Information, Data", 2018, Knowledge Management Tools, http://www.knowledge-management-tools.net/knowledge-information-data.html
6. "Data, Information, Knowledge, Wisdom?", Information is Beautiful, https://informationisbeautiful.net/2010/data-information-knowledge-wisdom/#comment-35878
7. "Learning Guides", Visual Paradigm, https://www.visual-paradigm.com/guide/
8. I Brudo, 2020, SVN vs. Git: Which is right for you in 2020?, Codata, https://blog.codota.com/svn-vs-git/
9. D, Adam, 2014, Assessing challenges of continuous integration in the context of software requirements breakdown: a case study, Chalmers University of Technology and University of Gothenburg, http://publications.lib.chalmers.se/records/fulltext/220573/220573.pdf
10. Continuous delivery, Wikipedia, https://en.wikipedia.org/wiki/Continuous_delivery
11. Continuous integration, Wikipedia, https://en.wikipedia.org/wiki/Continuous_integration
12. B Kent, 2002, *Test-Driven Development by Example*. Vaseem: Addison Wesley
13. Software testing, Wikipedia, https://en.wikipedia.org/wiki/Software_testing
14. Software deployment, Wikipedia, https://en.wikipedia.org/wiki/Software_deployment
15. Software maintenance, Wikipedia, https://en.wikipedia.org/wiki/Software_maintenance
16. Software Maintenance Overview, TutorialsPoint, https://www.tutorialspoint.com/software_engineering/software_maintenance_overview.htm
17. Software documentation, Wikipedia, https://en.wikipedia.org/wiki/Software_documentation
18. "Difference between Good Design and Bad Design in Software Engineering", 2020, Geeks for Geeks, https://www.geeksforgeeks.org/difference-between-good-design-and-bad-design-in-software-engineering/