# Software Engineering for High Performance Computing

**Rao**
Politecnico di Milano

Originally written in: **2025-02-17**

Last updated at: **2025-05-15**

# Contents

# 1 Introduction

## 1.1 Why software engineering is important

Software is everywhere and our society is now totally dependent on **software-intensive systems**. Obviously, our society could not function without software. For example, transportation systems, energy systems and manufacturing systems.

## 1.2 Definition

There are some fields of computer science dealing with software systems:

- Large and complex
- Built by teams
- It exists in may versions
- Last many years
- Undergo changes

## 1.3 Required skills

We should know that software engineering is not just about programming, as a **programmer**, we should have these following skills:

- develops a complete program
- works on known specifications
- works individually

For a **software engineer**, we should have these following skills:

- identifies requirements and develops specifications
- designs a component to be combined with other components, developed, maintained,used by others; component can become part of several systems
- works in a team

We can summarize the skills of a software engineer as follows:

- Technical
- Project management
- Cognitive
- Enterprise organization
- Interaction with different cultures
- Domain knowledge

The quality of human resources is of primary importance. The main goal of a software engineer is to develop software products. Not only is the product significant, but the process is also fundamental. The quality of the process affects the quality of the product.

## 1.4 The software product and the process

The product developed by a software engineer differs from traditional product types. It isn't easy to describe and evaluate because it is intangible. Some **aspects affecting the product quality**:

- Development technology
- Process quality
- People quality
- Cost, time and schedule

### 1.4.1 ISO/IEC 25010

An ISO (International Organization for Standardization) called **ISO/IEC 25010** comprises the **nine quality characteristics**:

| SOFTWARE PRODUCT QUALITY | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| FUNCTIONAL SUITABILITY | PERFORMANCE EFFICIENCY | COMPATIBILITY | INTERACTION CAPABILITY | RELIABILITY | SECURITY | MAINTAINABILITY | FLEXIBILITY | SAFETY |
| FUNCTIONAL COMPLETENESS | TIME BEHAVIOUR | CO-EXISTENCE | APPROPRIATENESS RECOGNIZABILITY | FAULTLESSNESS | CONFIDENTIALITY | MODULARITY | ADAPTABILITY | OPERATIONAL CONSTRAINT |
| FUNCTIONAL CORRECTNESS | RESOURCE UTILIZATION | INTEROPERABILITY | LEARNABILITY | AVAILABILITY | INTEGRITY | REUSABILITY | SCALABILITY | RISK IDENTIFICATION |
| FUNCTIONAL APPROPRIATENESS | CAPACITY | | OPERABILITY | FAULT TOLERANCE | NON-REPUDIATION | ANALYSABILITY | INSTALLABILITY | FAIL SAFE |
| | | | USER ERROR PROTECTION | RECOVERABILITY | ACCOUNTABILITY | MODIFIABILITY | REPLACEABILITY | HAZARD WARNING |
| | | | USER ENGAGEMENT | | AUTHENTICITY | TESTABILITY | | SAFE INTEGRATION |
| | | | INCLUSIVITY | | RESISTANCE | | | |
| | | | USER ASSISTANCE | | | | | |
| iso25000.com | | | SELF-DESCRIPTIVENESS | | | | | |

Figure 1.1: ISO/IEC 25010

### 1.4.2 Productivity

A process quality to consider is **productivity** (the process of **producing a product**). The definition can be: "ability to produce a good amount of product". To measure it, we can use **delivered items by unit of effort**, where:

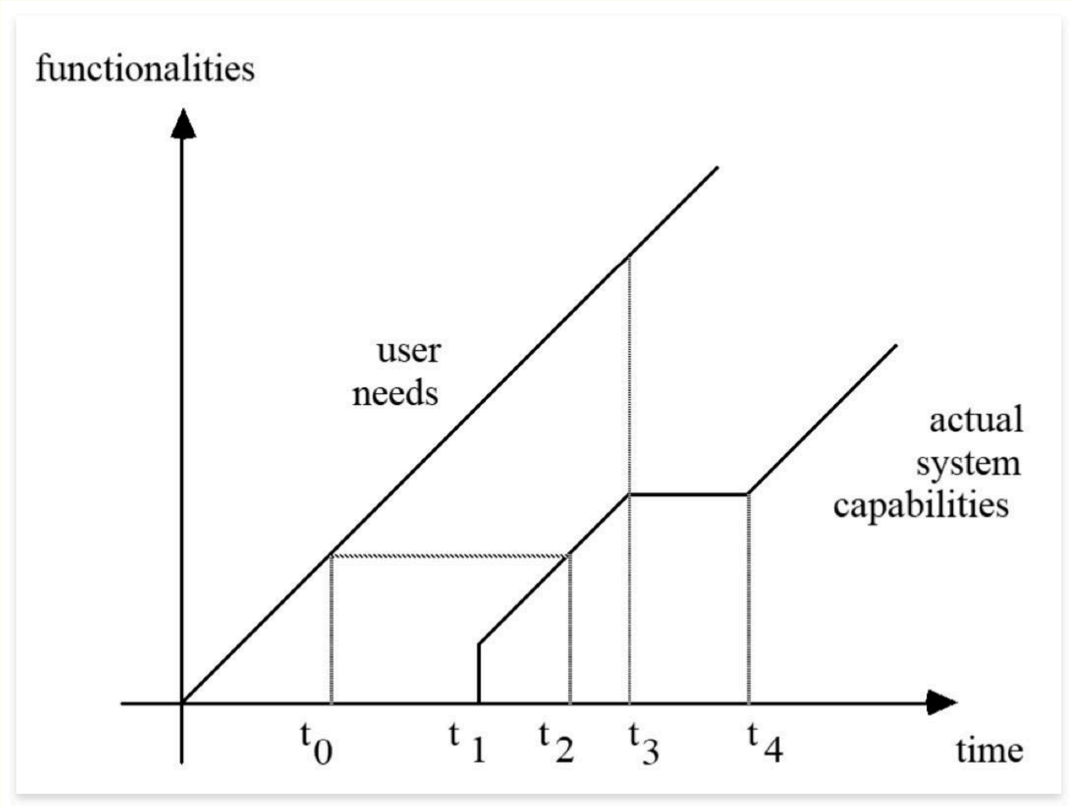
- **Delivered items**: lines of code (and variations) function points
- **Unit of effort**: person month (note: persons and months cannot be interchanged).

### 1.4.3 Timeliness

Another process quality to consider is timeliness. The definition is: "the ability to respond to change requests in a timely fashion".

As you can see by the graph, the "user needs" is a linear function. A software engineer should be able to respond to the client's requests as soon as possible. As the graph shows, a request made on time $t_0$ is completed on time $t_2$; but another request can be made at that time, and so on.

The actual system capabilities can't grow up always because sometimes there are "brainstorming times" to increase product quality.




## 1.5 Software Lifecycle

Initially, no reference model is inside a software lifecycle: code and fix(or refactoring). However, a traditional waterfall model is chosen to react to the many problems that a software engineer faces.

### 1.5.1 Waterfall model

The **waterfall model** is a breakdown of development activities into linear sequential phases, meaning they are passed down onto each other, where each phase depends on the deliverables of the previous one and corresponds to a specialization of tasks. Its organization is the following:
**High phases**:

- **Feasibility Study**: this is a cost-benefit analysis. The main goal is determining whether the project should be started, possible alternatives, and needed resources. The outcome is a feasibility study document. This paper provides
  - A preliminary problem description
  - Some scenarios describing possible solutions
  - Costs and schedules for the different alternatives
- **Requirement analysis and specification**: this is an analysis of the domain in which the application takes place. The main goal is to identify requirements and derive specifications

for the software. Note these specifics require a interaction with the user and an understanding of the properties of the domain. The **outcome** is a particular document called **Requirements Analysis and Specification Document** (RASD).

- **Design**: this is the **definition of the software architecture**. There, the definition of components (modules) and the relations/interactions among these components. The **main goal** is to support the concurrent development of separate responsibilities. The outcome is a summary of this info in a **design document**.

**Low phases**:

- **Coding and Unit Test**: each module is **implemented using the chosen programming language**. Furthermore, each module is tested in isolation by the module developer. Also, the programs should include their documentation.
- **Integration and System Test**: the modules are **integrated into** (sub)systems. The integrated (sub)systems are **tested**. Follows an incremental implementation scheme. A complete system test is needed to verify the overall properties. Note that sometimes we have alpha test and beta test.
- **Deployment**: is the process used to conceive, specify, design, program, document, test, and bug fix to create and maintain applications, frameworks, or other software components.
- **Maintenance**: the maintenance is divided into **two types**:
  - Corrective deals with the repair of faults or defects found.
  - Evolution is also divided into three types:
    - **Adaptive**: the software is adapted to new environments or requirements.
    - **Perfective**: the software is improved to meet new requirements.
    - **Preventive**: the software is improved to prevent future problems.

# 2 HPC software, relevant qualities and SE methods

High Performance Computing the practice of aggregating computing power in a way that delivers much high performance than one could get out of a typical desktop computer or workstation to solve large problems in science, engineering, or business.

Thousands of processors working in parallel to analyze billions of pieces of data in real time, performing calculations thousands of times faster than a normal computer.

The use of parallel processing for running advanced, large-scale application programs efficiently, reliably and very quickly on supercomputer systems.

The platform technology concerned with programming for performance, where performance takes the broad meaning of:

- Speed (reducing time to solution)
- Energy efficiency (doing more with less power)
- Upscaling (handling larger problems such as simulating a wing and then a full plane, or a cell and then an organ)
- High throughput (the ability to handle large volumes of data in near real-time, as required in the financial services industry, telecoms or satellite imagery).

## 2.1 HPC software

There are two categories of HPC software:

- **Compute-intensive applications**: These are complex calculations that require a large number of computing resources. They also often require parallel computing.
- **Data-intensive applications**. They focus on processing, storing and retrieving large amounts of data. Typically built as distributed systems to ensure reliability and scalability.

## 2.2 Relevant Qualities

For the two categories explained, there are some important characteristics:

For Compute-intensive applications:

- **Correctness**
- **Performance**
- **Portability**
- **Maintainability**
- **Evolvability**

For Data-intensive applications:

- **Reliability**

- **Scalability**
- **Maintainability**

In the software, there can be some errors, but a software engineer should be able to recognize the type of failure, faults or defects. The fault can be of two types:

- **Hardware Faults**
- **Software Faults**

## 2.3 Systems Engineering Methods

There are several systems engineering methodologies required in High Performance Computing.

- Modelling the software structure and checking its properties.
- Performance analysis and improvement.
- Documentation, standards, support to maintainability.
- Support to scalability.
- Attention to operability and automation.

# 3 Requirement Engineering

## 3.1 Definition

**Requirement engineering** is the process of defining, documenting and maintaining requirements in the engineering design process. It is a critical part of software development and systems engineering. The questions derived from requirement engineering are:

- **Identify stakeholders**
- **Identify their needs**
- **Produce documentation**
- **Analyze, communicate, implement requirements**

## 3.2 Interplant between the world and the machine

For an ambulance dispatching system:

- For every urgent call reporting an incident, an ambulance should arrive at the incident location within 14 mins
- For every urgent call, details about the incident are correctly encoded
- When an ambulance is dispatched, it will reach the incident location in the shortest possible time
- Accurate ambulance locations are known by GPS
- Ambulance crews correctly notify ambulance availability through a mobile data terminal

The **machine** is the part of the system to be developed (typically a software-to-be and a hardware). The **world** (or environment) is the part of the real world that is affected by the machine.

Requirements engineering is **concerned with the phenomena that occur in the world**. In the previous example, RE is concerned with the following phenomena:

- Occurrence of incidents
- Reports of incidents by public calls
- Encoding of call details into dispatching software
- Assignment of an ambulance
- Arrival of an ambulance at the scene of an incident

But RE is also interested in the phenomena that occur inside the machine. In the previous example

- The creation of a new object of the class `Incident`
- The updating of a database entry

Using the example on the previous page, we can show the phenomena we are interested in the world or in the machine set.

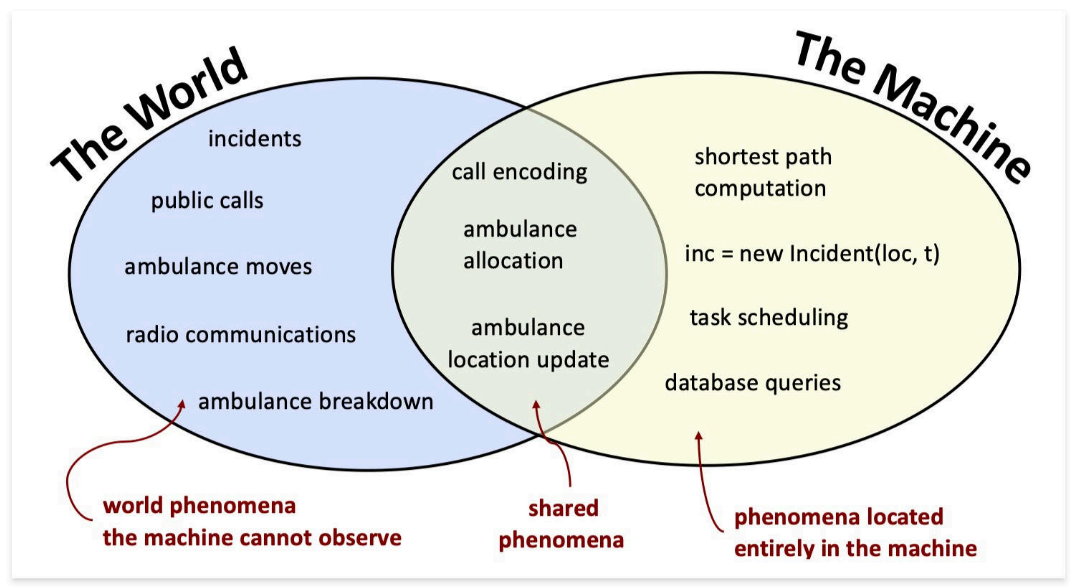



Figure 3.3: World and machine set

More generally, we can divide the machine and the world sets as:

- The world which have goals and domain properties
- The machine which have computers and programs
- The requirements which is the bridge between the world and the machine.

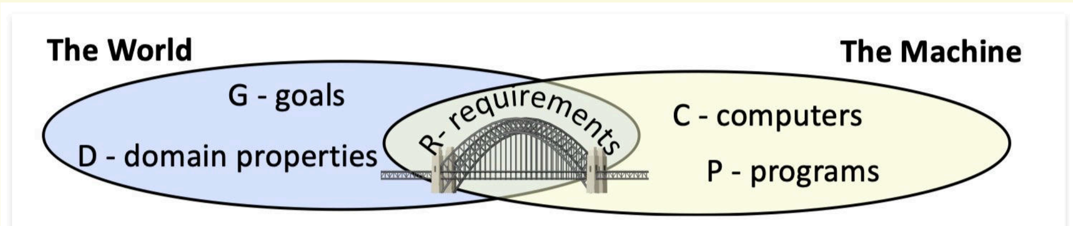



Figure 4: Goals, domain properties, requirements, computers and programs.

We explain more detailed these value inside the two sets:

- **Goals** are prescriptive assertions formulated in terms of world phenomena (not necessarily shared)
- **Domain properties** are descriptive assertions assumed to hold in the world
- **Requirements** are prescriptive assertions formulated in terms of shared phenomena

Using the example, we can identify the goal, the domain assumptions and the requirement as follows:

- **Goal**: For every urgent call reporting an incident, an ambulance should arrive at the incident scene within 14 minutes.
- **Domain assumptions**
  - For every urgent call, details about the incident are correctly encoded.

  - When an ambulance is mobilized, it will reach the incident location in the shortest possible time.
  - Accurate ambulances' location are known by GPS.
  - Ambulance crews correctly signal ambulance availability through mobile data terminals on board of ambulances.
- **Requirement**: When a call reporting a new incident is encoded, the Automated Dispatching Software should mobilize the nearest available ambulance according to information available from the ambulances' GPS and mobile data terminals.

**Def Completeness of Requirements** **definition 3.2.1**

We say that $R$ is **complete** if and only if:

- **R** ensures satisfaction of **G** in the context of domain assumptions **D**
$$R \text{ and } D \mid = G \tag{3.1}$$
- **G** captures all the stakeholders' needs.
- **D** represents **valid properties/assumptions about the world**.

## 3.3 Formulating and classifying requirements

The requirements can be of three types:

- **Functional requirements**: describe the interactions between the system and its environment (independent from implementation). In other words, are the main (functional) goals the software has to realize.
- **Non-functional requirements (NFRs)**: further characterization of user-visible aspects of the system not directly related to functions.
- **Constraints requirements** (or **technical requirements**): imposed by the customer or the environment in which the system operates.

NFRs have some characteristics:

- Constraints on **how functionality must be provided to the end user**
- The application domain determines their **relevance** and their **prioritization**
- Have a **strong influence on the structure of the system to be built**. For example, a system may require 24/7 availability. As a result, it is likely to be designed as a replicated system (with redundant components).

## 3.4 Eliciting requirements

The **Requirements Elicitation** is the practice of researching and discovering the requirements of a system from users, customers, and other stakeholders. The goal of **requirements elicitation** is to ensure that the software development process is based on a clear and comprehensive understanding of the customer's needs and requirements. To do that, exist a simple and effective tool called ***scenarios***.

**Def Scenario** **definition 3.4.1**

A scenario is a concrete, focused, informal description of a single feature of the system to be.

## 3.5 Use cases

Scenarios provide a nice summary of what the requirements analysis team can derive from observation, interviews, analysis of documentation. By generalizing the scenarios, we can get **Use Cases**.

To specify a use case, it's very important to follow the following scheme.

**Def Use Case Schema** **definition 3.5.1**

- **Name of Use Case**
- **Actors**: Description of Actors involved in the use case
- **Entry condition**: When this use case starts the following condition is true
- **Flow of Events**: Free form, informal natural language
- **Exit condition**: When this use case ends the following condition is true
- **Exceptions**: Describe what happens if things go wrong
- **Special Requirements**: Nonfunctional Requirements, Constraints.

The following suggestions are useful in defining an appropriate use cases:

- Use cases named with **verbs** that indicate hat the user is trying to accomplish
- Actors named with **nouns**
- Use cases steps in **active voice**
- The causal relationship between steps should be clear
- A use case per user transaction
- Separate description of exceptions
- Keep use cases small (no more than two/three pages)
- The steps accomplished by actors and those accomplished by the system should be clearly distinguished (this helps us in identifying the boundaries of the system)

**e.g. Bad Use case** **example 3.5.1**

Example of a bad use case referring to the ambulance dispatching:

- **Use case name**: Accident
- **Actors**: Field Officer
- **Flow of Events**:
  1. The Field Officer reports the accident
  2. An ambulance is dispatched
  3. The dispatcher is notified when the ambulance arrives on the site.

The errors are as follows:

- In the use case name field, **the word is a noun**. It's better to use a verb that indicates what the user is trying to achieve.
- The Dispatcher actor is not declared in the Participating Actors field, but is mentioned in the Flow of Events field.

Now we present an example of a *well composed* use case.

**e.g. Good Use case** **example 3.5.2**

There are two **actors** involved:

- **Field Officer**: the person who reports the accident
- **Dispatcher**: the person who dispatches the ambulance

The **Entry Condition** is always true because an emergency can be reported at any time. The **sequence of events** is as follows:

- The **FieldOfficer** activates the Report Emergency function of her terminal
- Friend (the system to be developed) responds by presenting a form to the officer
- The FieldOfficer fills the form, by selecting the emergency level, type, location, and brief description of the situation
- At which point, the Dispatcher is notified
- The Dispatcher reviews the submitted information and allocates resources by invoking the AllocateResources use case. The Dispatcher selects a response and acknowledges the emergency report

The **Exit Condition** is the following: the FieldOfficer has received the acknowledgment and the selected response.

There are two **exceptions**:

- The FieldOfficer is notified immediately if the connection between her terminal and the control room is lost
- The Dispatcher is notified immediately if the connection between any logged in Field-Officer and the control room is lost

And the **Special Requirements** are:

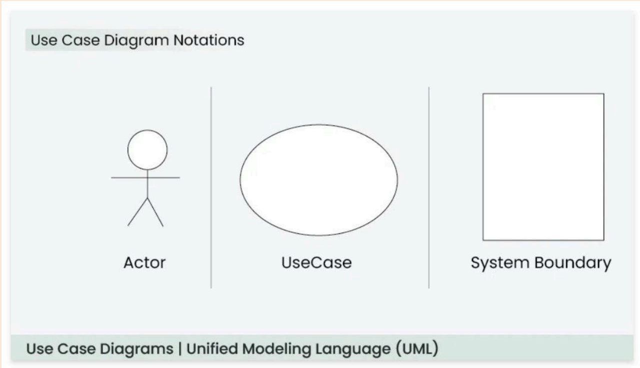
- The FieldOfficer's report is acknowledged within 30 seconds
- The selected response arrives no later than 30 seconds after it is sent by the Dispatcher

## 3.6 UML Use Case Diagrams

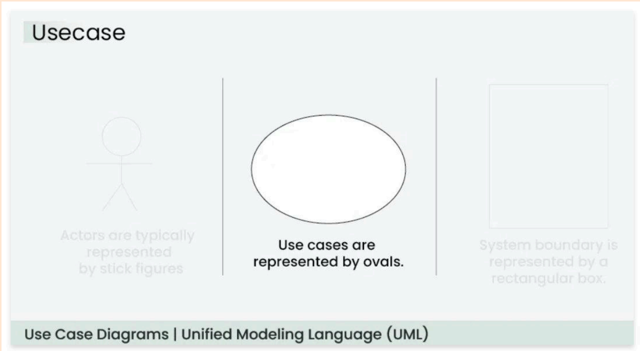
A Use Case Diagram is a type of **Unified Modeling Language (UML)** diagram that represents the interaction between actors (users or external systems) and a system under consideration to accomplish specific goals. It provides a high-level view of the system's functionality by illustrating the various ways users can interact with it.

**Def** **The shapes of a Use Case Diagram** **definition 3.6.1**

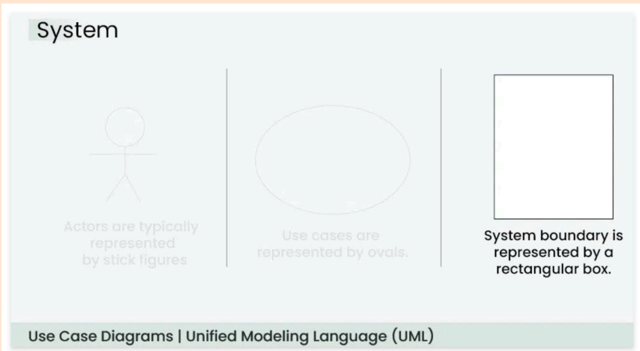
- **Actors**:




- **Use Cases**:



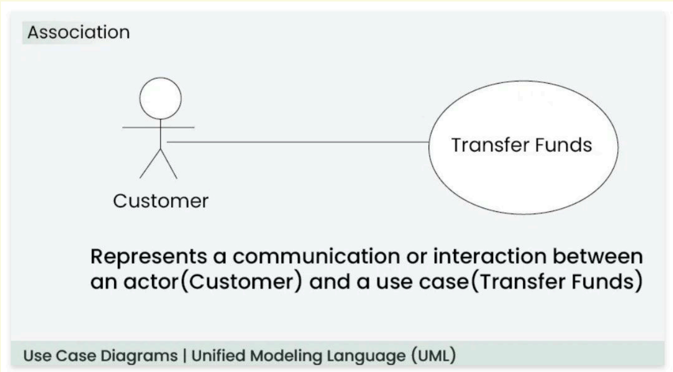

- **System Boundary**: The system boundary is a visual representation of the scope or limits of the system you are modeling. It defines what is inside the system and what is outside. The boundary helps to establish a clear distinction between the elements that are part of the system and those that are external to it. The system boundary is typically represented by a rectangular box that surrounds all the use cases of the system.




### 3.6.1 Association Relationship

In a Use Case Diagram, **relationships** play a crucial role in depicting the interactions between actors and use cases. These relationships provide a comprehensive view of the system's functionality and its various scenarios. Take *Online Banking System* as an example:

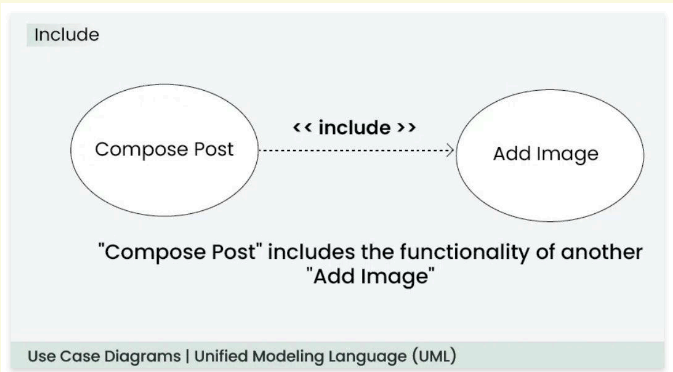
- **Actor**: Customer
- **Use Case**: Transfer Funds
- **Association**: A line connecting the "Customer" actor to the "Transfer Funds" use case, indicating the customer's involvement in the funds transfer process.




### 3.6.2 Include Relationship

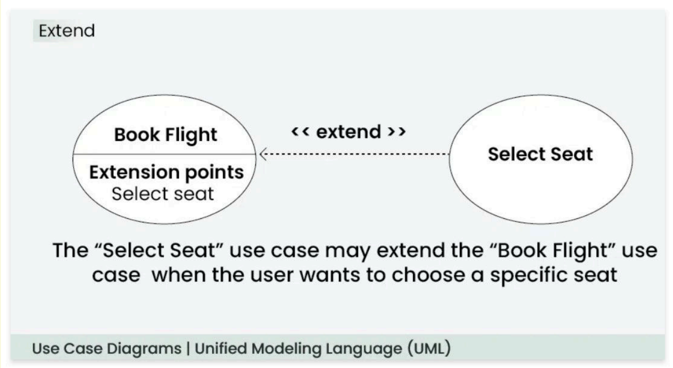
The ***Include Relationship*** indicates that a use case includes the functionality of another use case. It is denoted by a dashed arrow pointing from the including use case to the included use case. This relationship promotes modular and reusable design.

- **Use Cases**: Compose Post, Add Image
- **Include Relationship**: The "Compose Post" use case includes the functionality of "Add Image." Therefore, composing a post includes the action of adding an image.



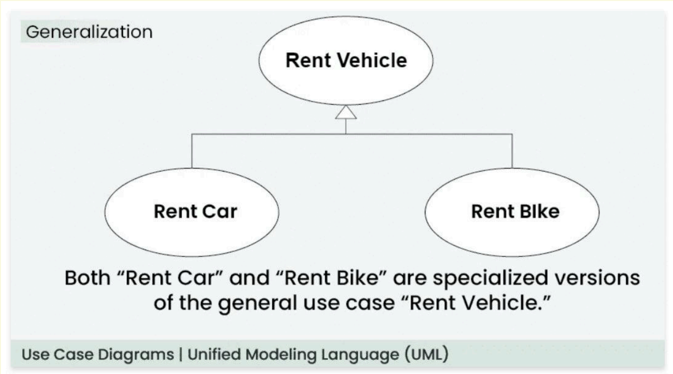

### 3.6.3 Extend Relationship

The ***Extend Relationship*** illustrates that a use case can be extended by another use case under specific conditions. It is represented by a dashed arrow with the keyword "extend." This relationship is useful for handling optional or exceptional behavior.




### 3.6.4 Generalization Relationship

The ***Generalization Relationship*** establishes an "is-a" connection between two use cases, indicating that one use case is a specialized version of another. It is represented by an arrow pointing from the specialized use case to the general use case.




Below are the main steps to draw use case diagram in UML:

1. **Identify Actors**: Determine who or what interacts with the system. These are your actors. They can be users, other systems, or external entities.
2. **Identify Use Cases**: Identify the main functionalities or actions the system must perform. These are your use cases. Each use case should represent a specific piece of functionality.
3. **Connect Actors and Use Cases**: Draw lines (associations) between actors and the use cases they are involved in. This represents the interactions between actors and the system.
4. **Add System Boundary**: Draw a box around the actors and use cases to represent the system boundary. This defines the scope of your system.
5. **Define Relationships**: If certain use cases are related or if one use case is an extension of another, you can indicate these relationships with appropriate notations.
6. **Review and Refine**: Step back and review your diagram. Ensure that it accurately represents the interactions and relationships in your system. Refine as needed.
7. **Validate**: Share your use case diagram with stakeholders and gather feedback. Ensure that it aligns with their understanding of the system's functionality.

**e.g. Online Shopping System** **example 3.6.1**

Let's understand how to draw a Use Case diagram with the help of an Online Shopping System:

**Actors**:

- Customer
- Admin

**Use Case**:

- Browse Products
- Add to Cart
- Checkout
- Manage Inventory (Admin)

**Relations**:

- The Customer can browse products, add to the cart, and complete the checkout.
- The Admin can manage the inventory.

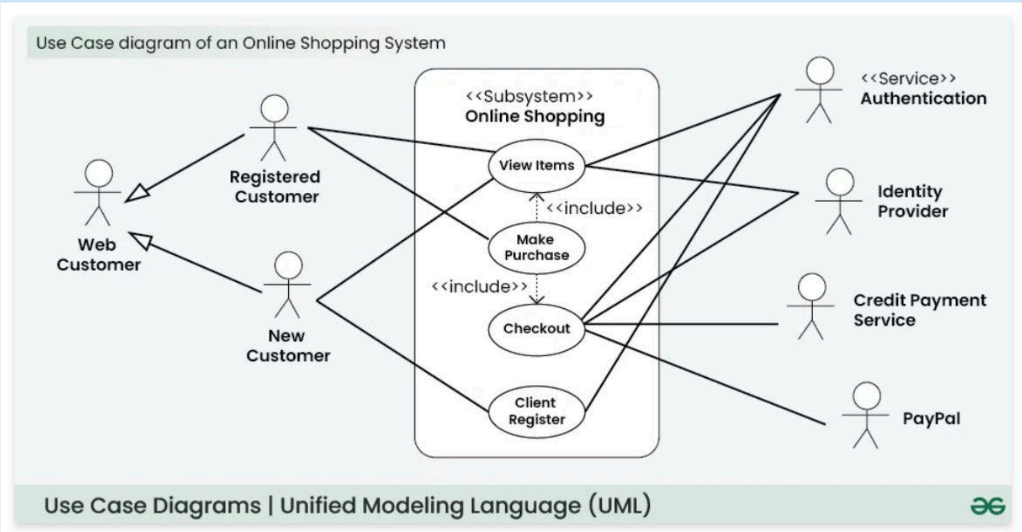



Figure 3.12: Use Case Diagram for Online Shopping System

# 4 Software Design

## 4.1 Software Architecture

The Architecture is so **important** because it is the vehicle for communication: internal (different teams) and external (teams and stakeholders). The Architecture manifests the first set of design decisions and is a portable abstraction of a system.

> **Def Software Architecture** **definition 4.1.1**
>
> The ***Software Architecture (SA)*** of a system is the **set of structures** needed to reason about the system. These structures comprise software **elements**, **relations** among them, and **properties** of both.

## 4.2 Architecture and multiple structures

### 4.2.1 Component-and-connector (C&C) structures

Describe how the system is structured as a **set of elements** that have **runtime behavior** (components) and **interactions** (connectors).

- The **components** are the principal units of computation (for example the clients, servers, services, etc.)
- The **connectors** represent communication (for example request-response mechanisms, pipes, asynchronous messages, etc.)

The purpose of these structures is to enable us to answer questions such as:

- What are the major executing components and how do they interact at runtime?
- What are the major shared data stores?
- Which parts of the system are replicated?
- How does data progress through the system?
- Which parts of the system can run in parallel?
- How does the system's structure evolve during execution?

### 4.2.2 Module structures

Show how a system is structured as **a set of code or data units** that have to be procured or constructed, **together with their relations**. An example of modules: packages, classes, functions, libraries, layers, database tables, etc.

Modules constitute **implementation units** that can be used as the basis for work splitting (identifying functional areas of responsibility). Typical relations among modules are: uses, is-a (generalization), is-part-of.

### 4.2.3 Allocation structures

Define **how the elements** from component-and-connector or module structures **map** onto things that are not software. For example hardware (possibly virtualized), file systems, teams. Some typical allocation structures include deployment structure, implementation structure, work assignment structure.

## 4.3 Software design descriptions and UML

### 4.3.1 Component Diagram (C&C structure)

A **Component Diagram** breaks down the actual system under development into **various high levels of functionality**. Each component is responsible for one clear aim within the entire system and only interacts with other essential elements on a need-to-know basis.

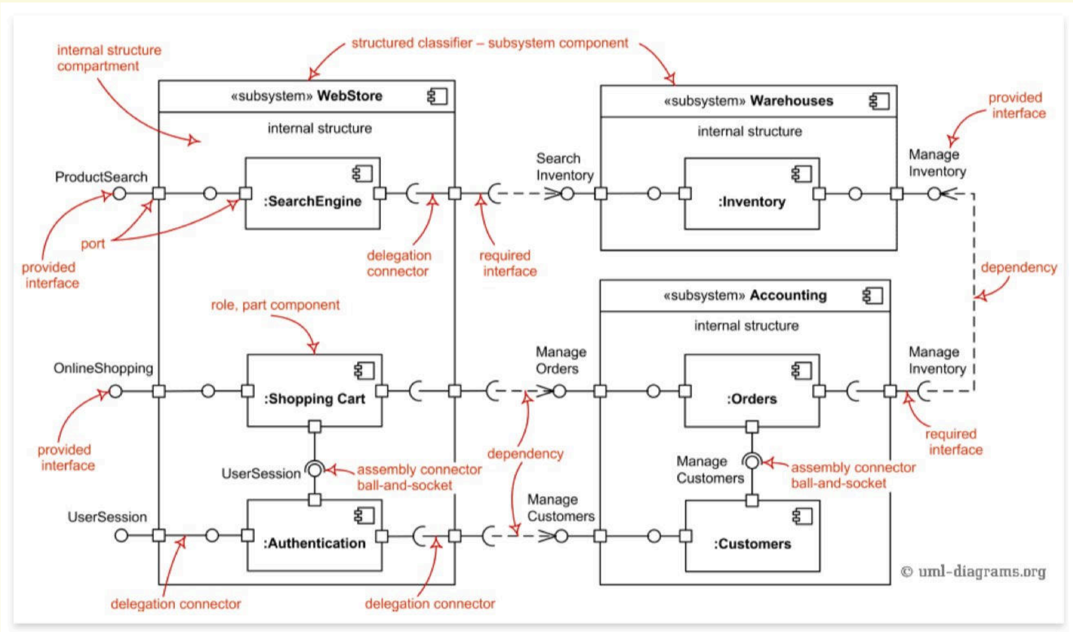


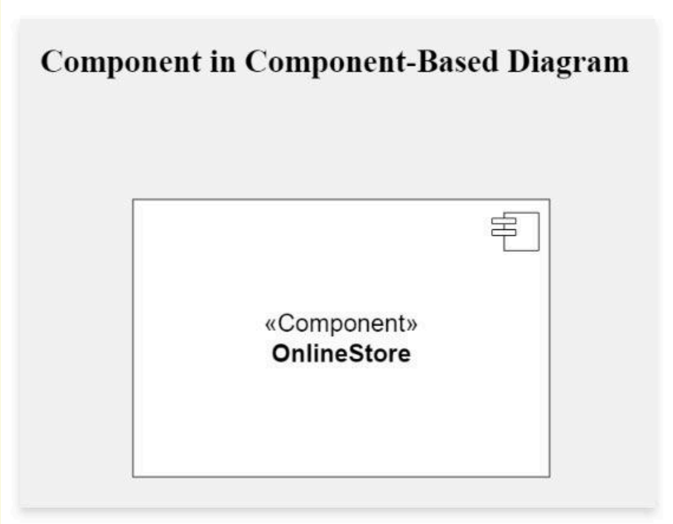

Figure 4.13: Component Diagram

Component-Based Diagrams in UML comprise several key elements, each serving a distinct role in illustrating the system's architecture. Here are the main components and their roles:

#### Component

Represent modular parts of the system that encapsulate functionalities. Components can be **software classes, collections of classes, or subsystems**.

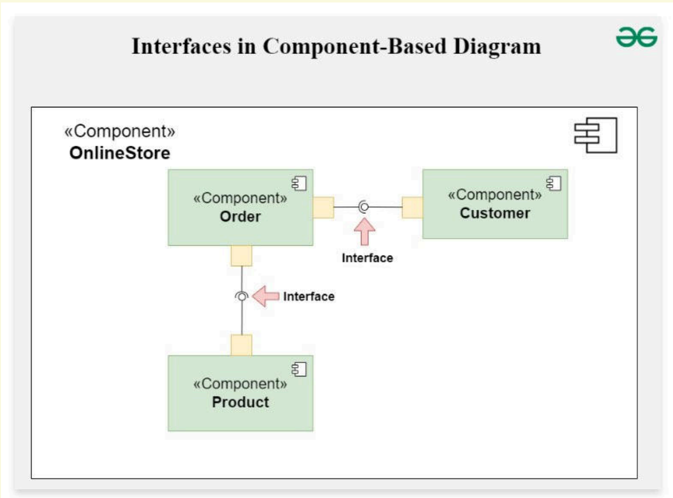
- **Symbol**: **Rectangles** with the component stereotype («component»).
- **Function**: Define and encapsulate functionality, ensuring **modularity and reusability**.




### Interfaces

Specify a set of operations that a component offers or requires, serving as a contract between the component and its environment.

- **Symbol**: **Circles** (lollipops) for provided interfaces and half-circles (sockets) for required interfaces.
- **Function**: Define how components communicate with each other, ensuring that components can be developed and maintained independently.

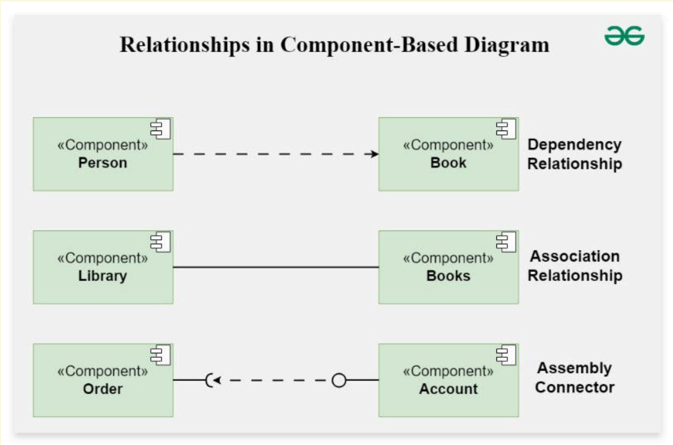



### Relationships

Depict the connections and dependencies between components and interfaces.

- **Symbol**: Lines and arrows.
  - **Dependency (dashed arrow)**: Indicates that one component relies on another.
  - **Association (solid line)**: Shows a more permanent relationship between components.
  - **Assembly connector**: Connects a required interface of one component to a provided interface of another.

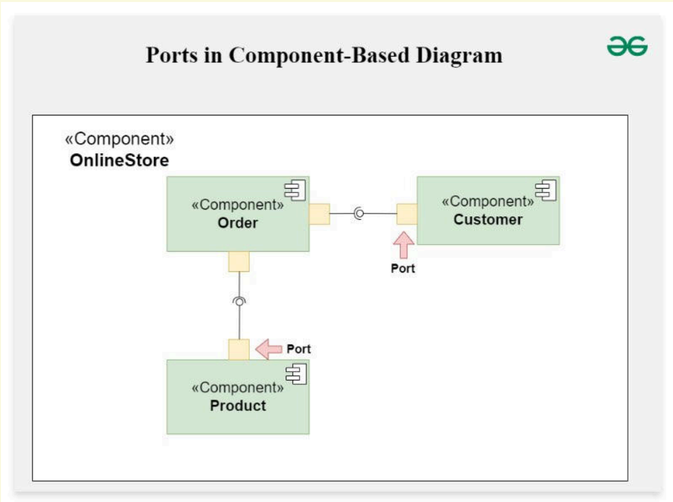
- **Function**: Illustrate how components interact and depend on each other, helping to visualize the overall architecture of the system.




**Ports**

Role: Represent specific interaction points on the boundary of a component where interfaces are provided or required.

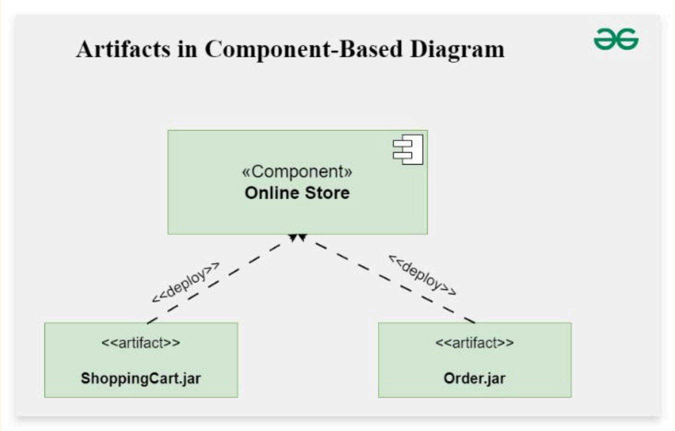
- **Symbol**: Small squares on the component boundary.
- **Function**: Allow for more precise specification of interaction points, facilitating detailed design and implementation.




**Artifacts**

Represent physical files or data that are deployed on nodes.

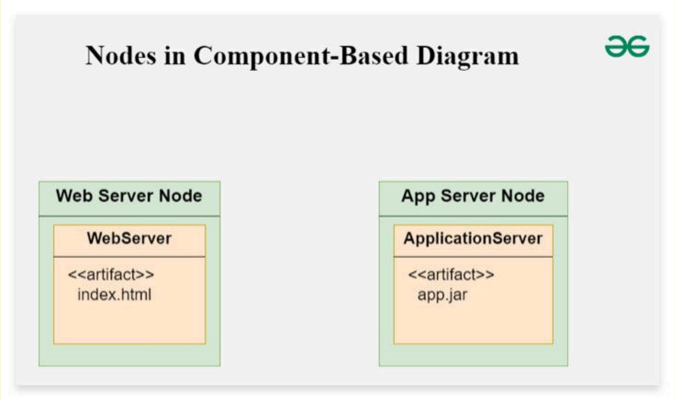
- **Symbol**: Rectangles with the artifact stereotype («artifact»).
- **Function**: Show how software artifacts, like executables or data files, relate to the components.




### Nodes

Represent physical or virtual execution environments where components are deployed.

- **Symbol**: 3D boxes.
- **Function**: Provide context for deployment, showing where components reside and execute within the system's infrastructure.




### Steps to Create Component-Based Diagrams

From understanding the system requirements to creating the final design, there are multiple processes involved in creating a component-based diagram. These steps will assist you in creating the ideal component-based diagram:

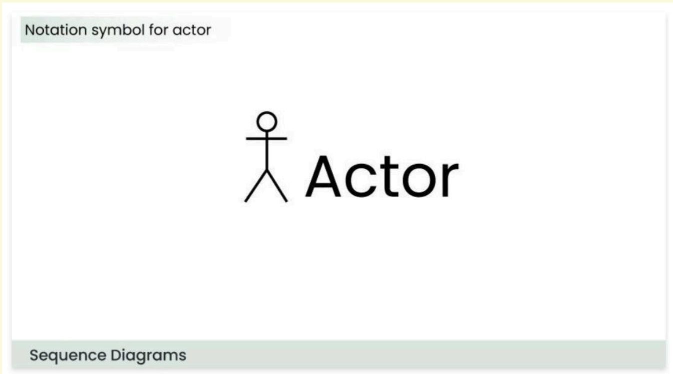
1. **Identify the System Scope and Requirements**
2. **Identify and Define Components**
3. **Identify Provided and Required Interfaces**
4. **Identify Relationships and Dependencies**
5. **Identify Artifacts**
6. **Identify Nodes**
7. **Draw the Diagram**
8. **Review and Refine the Diagram**

### 4.3.2 Sequence Diagram (C&C structure)

A **Sequence Diagram** is a key component of Unified Modeling Language (UML) used to visualize the interaction between objects in a sequential order. It focuses on **how objects communicate with each other over time**, making it an essential tool for modeling dynamic behavior in a system. Sequence diagrams illustrate object interactions, message flows, and the sequence of operations, making them valuable for understanding use cases, designing system architecture, and documenting complex processes.

**Actors**

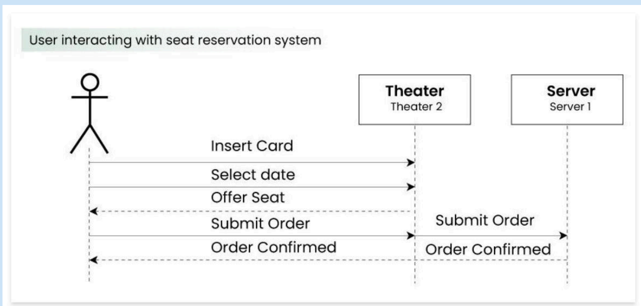
An actor in a UML diagram represents a type of role where it interacts with the system and its objects. It is important to note here that an actor is always outside the scope of the system we aim to model using the UML diagram.




We represent an actor in a UML diagram using a stick person notation. We can have multiple actors in a sequence diagram.

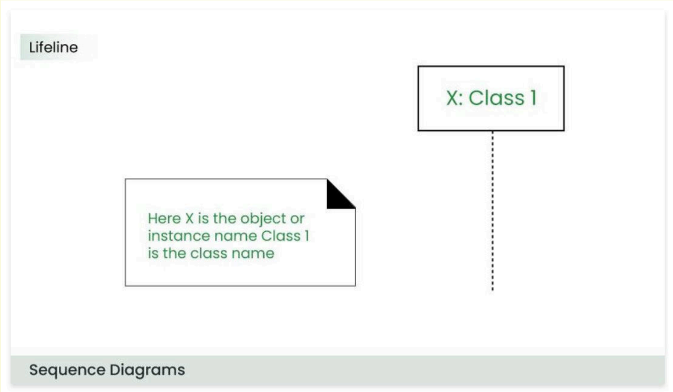
**e.g. Actors** **example 4.3.1**

Here the user in seat reservation system is shown as an actor where it exists outside the system and is not a part of the system.




**Lifelines**

A **lifeline** is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a sequence diagram is represented by a lifeline. Lifeline elements are located at the top in a sequence diagram.

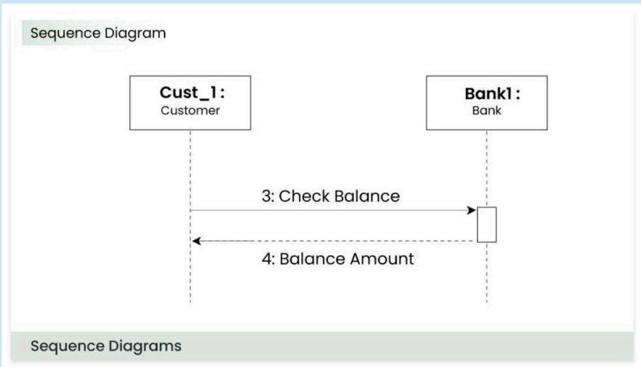



We display a lifeline in a rectangle called head with its name and type. The head is located on top of a vertical dashed line (referred to as the stem) as shown above.

- If we want to model an unnamed instance, we follow the same pattern except now the portion of lifeline's name is left blank.
- **Difference between a lifeline and an actor**: A lifeline always portrays an object internal to the system whereas actors are used to depict objects external to the system.

**e.g. Lifelines** **example 4.3.2**

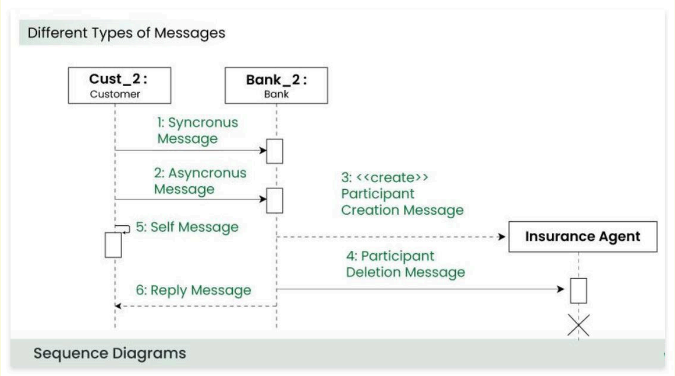
The following is an example of a sequence diagram:




**Messages**

Communication between objects is depicted using messages. The messages appear in a sequential order on the lifeline.

- We represent messages using arrows.
- Lifelines and messages form the core of a sequence diagram.

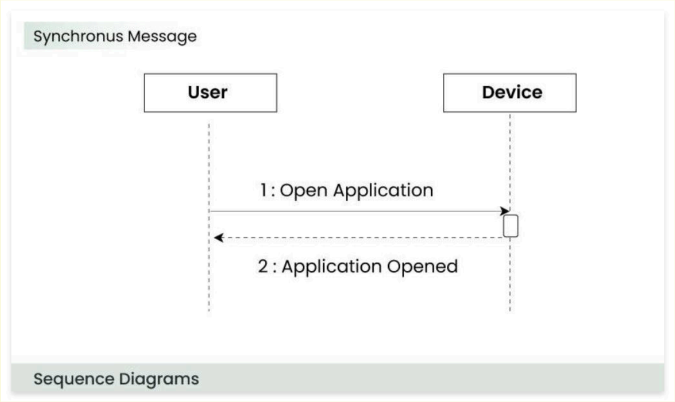



Messages can be broadly classified into the following categories:

### Synchronous Messages

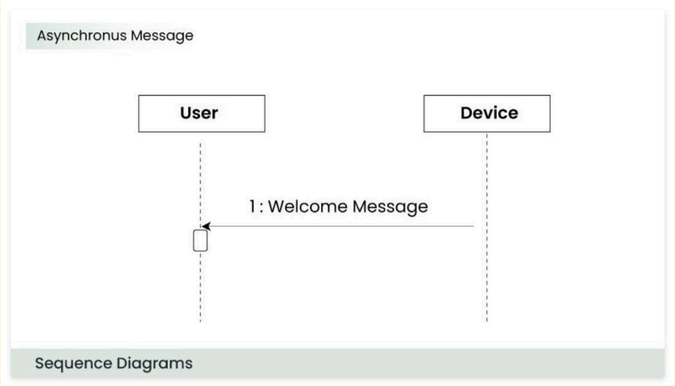
A **synchronous message** waits for a reply before the interaction can move forward. The sender waits until the receiver has completed the processing of the message. The caller continues only when it knows that the receiver has processed the previous message i.e. it receives a reply message.

- A large number of calls in **object oriented programming** are synchronous.
- We use a **solid arrow** head to represent a synchronous message.



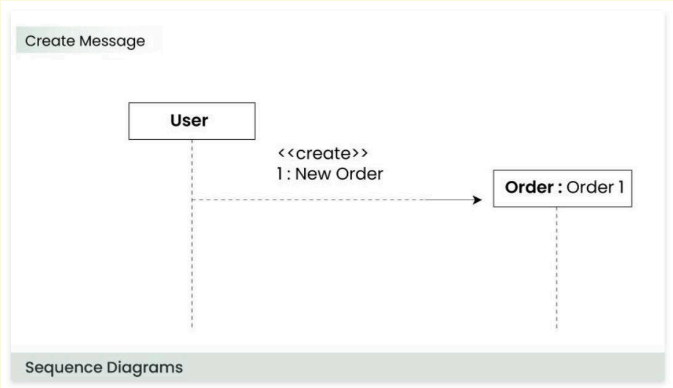

### Asynchronous Messages

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



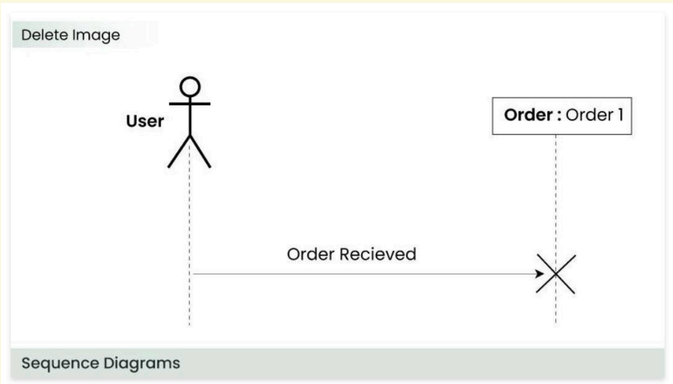
### Create Messages

We use a **Create message** to instantiate a new object in the sequence diagram. There are situations when a particular message call requires the creation of an object. It is represented with a dotted arrow and create word labelled on it to specify that it is the create Message symbol.



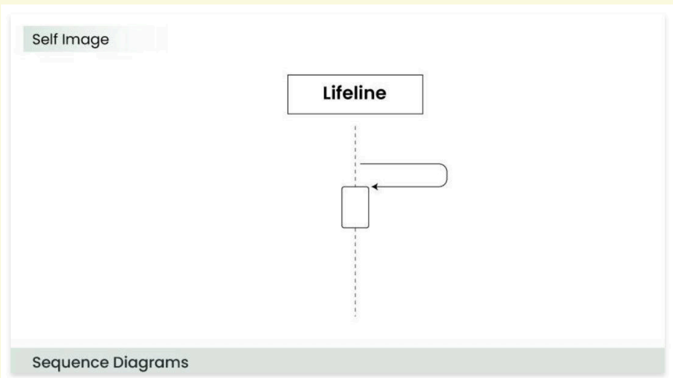

### Delete Messages

We use a **Delete Message** to delete an object. When an object is deallocated memory or is destroyed within the system we use the Delete Message symbol. It destroys the occurrence of the object in the system.It is represented by an arrow terminating with a `x`.



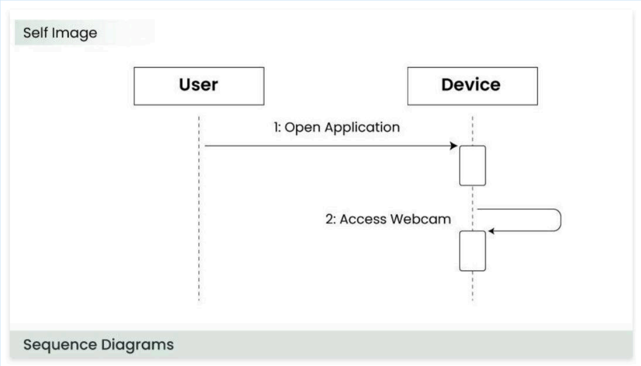

### Self Message

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



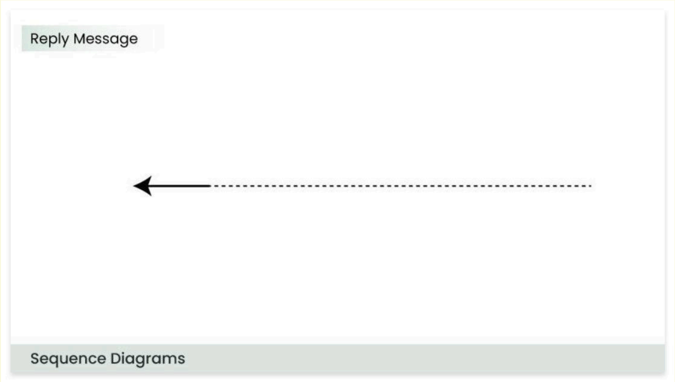
**e.g. self-messages** **example 4.3.3**

Consider a scenario where the device wants to access its webcam. Such a scenario is represented using a self message.



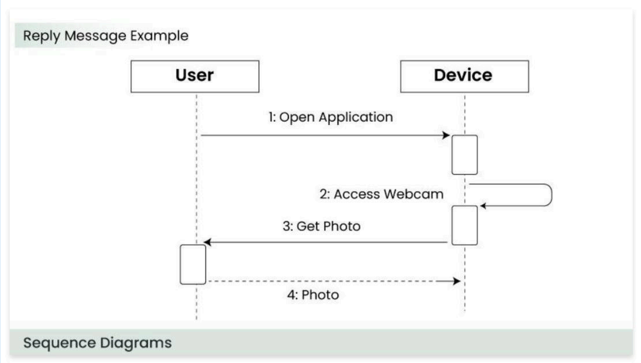

**Reply Messages**

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




**e.g. reply-messages** **example 4.3.4**

Consider the scenario where the device requests a photo from the user. Here the message which shows the photo being sent is a reply message.

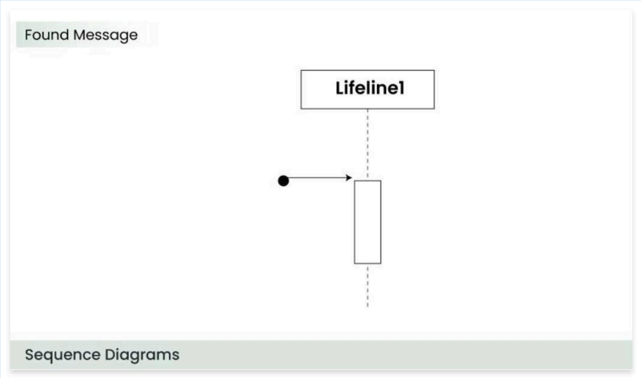


### Found Messages

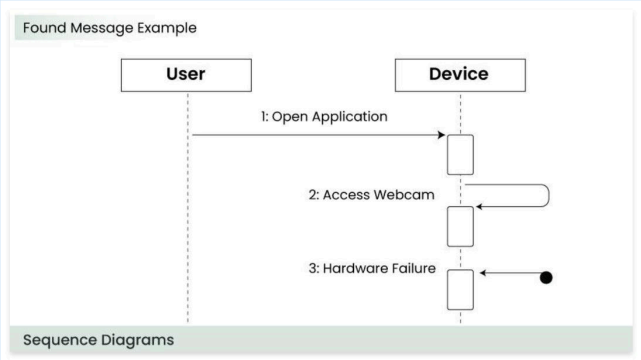
A Found message is used to represent a scenario where an unknown source sends the message. It is represented using an **arrow directed towards a lifeline** from an end point.

**Found message** **example 4.3.5**

Consider the scenario of a hardware failure.




It can be due to multiple reasons and we are not certain as to what caused the hardware failure.

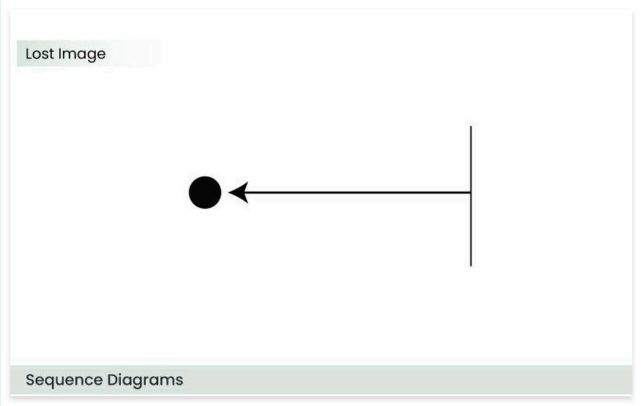



### Lost Messages

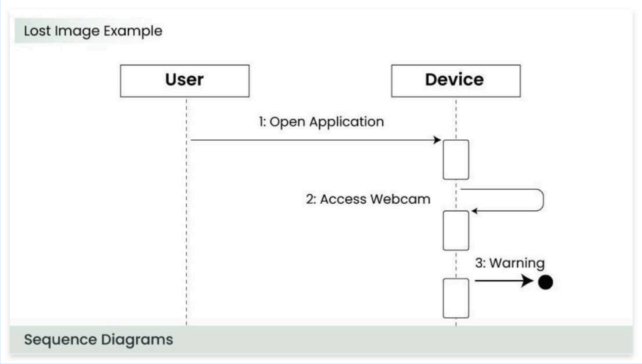
A Lost message is used to represent a scenario where the recipient is not known to the system. It is represented using an arrow directed towards an end point from a lifeline.

**e.g. Lost message** **example 4.3.6**

Consider a scenario where a warning is generated.




The warning might be generated for the user or other software/object that the lifeline is interacting with. Since the destination is not known before hand, we use the Lost Message symbol.

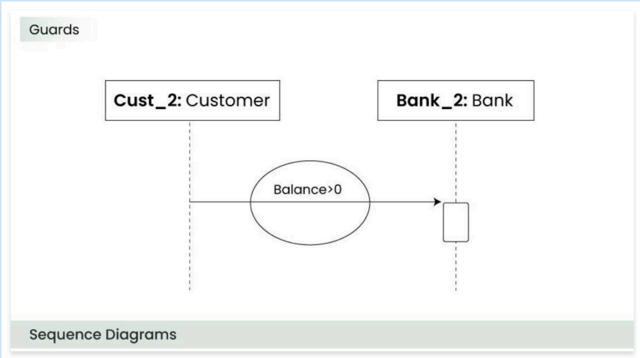



**Guard Messages**

To model conditions we use **guards** in UML. They are used when we need to restrict the flow of messages on the pretext of a condition being met. Guards play an important role in letting software developers know the constraints attached to a system or a particular process.

**e.g. Guard message** **example 4.3.7**

In order to be able to withdraw cash, having a balance greater than zero is a condition that must be met as shown below.




**How to create Sequence Diagrams**

Creating a sequence diagram involves several steps, and it's typically done during the design phase of software development to illustrate how **different components** or **objects interact** over time. Here's a step-by-step guide on how to create sequence diagrams:

1. **Identify the Scenario**: Understand the specific scenario or use case that you want to represent in the sequence diagram. This could be a specific interaction between objects or the flow of messages in a particular process.
2. **List the Participants**: Identify the participants (objects or actors) involved in the scenario. Participants can be users, systems, or external entities.
3. **Define Lifelines**: Draw a vertical dashed line for each participant, representing the lifeline of each object over time. The lifeline represents the existence of an object during the interaction.
4. **Arrange Lifelines**: Position the lifelines horizontally in the order of their involvement in the interaction. This helps in visualizing the flow of messages between participants.
5. **Add Activation Bars**: For each message, draw an activation bar on the lifeline of the sending participant. The activation bar represents the duration of time during which the participant is actively processing the message.
6. **Draw Messages**: Use arrows to represent messages between participants. Messages flow horizontally between lifelines, indicating the communication between objects. Different types of messages include synchronous (solid arrow), asynchronous (dashed arrow), and self-messages.

7. **Include Return Messages**: If a participant sends a response message, draw a dashed arrow returning to the original sender to represent the return message.
8. **Indicate Timing and Order**: Use numbers to indicate the order of messages in the sequence. You can also use vertical dashed lines to represent occurrences of events or the passage of time.
9. **Include Conditions and Loops**: Use combined fragments to represent conditions (like if statements) and loops in the interaction. This adds complexity to the sequence diagram and helps in detailing the control flow.
10. **Consider Parallel Execution**: If there are parallel activities happening, represent them by drawing parallel vertical dashed lines and placing the messages accordingly.
11. **Review and Refine**: Review the sequence diagram for clarity and correctness. Ensure that it accurately represents the intended interaction. Refine as needed.
12. **Add Annotations and Comments**: Include any additional information, annotations, or comments that provide context or clarification for elements in the diagram.
13. **Document Assumptions and Constraints**: If there are any assumptions or constraints related to the interaction, document them alongside the diagram.

### 4.3.3 Class Diagram (Module structure)

A **Class Diagram** is a **type of static structure** diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

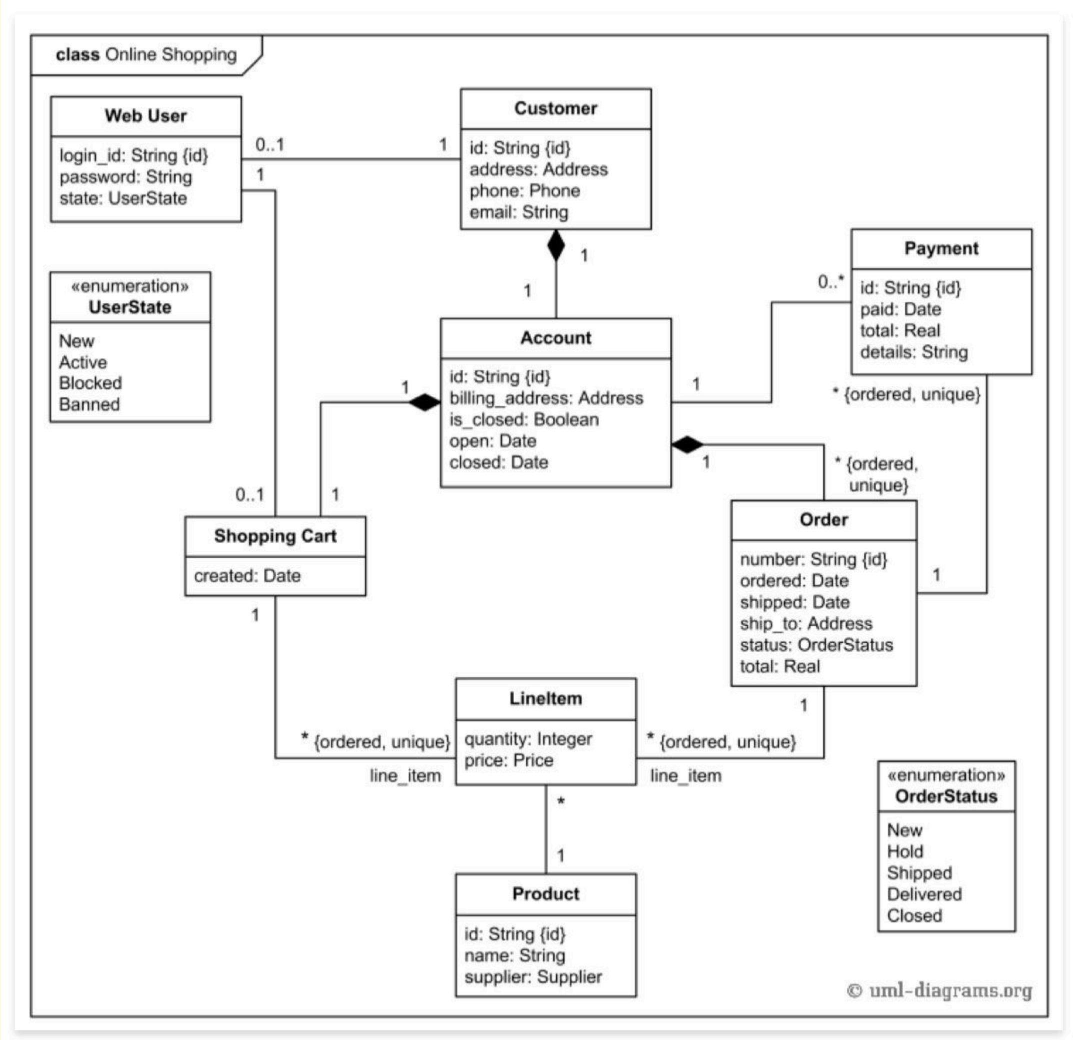



Figure 4.38: Class Diagram

### 4.3.4 Package Diagram (Module structure)

A **Package Diagram**, a kind of structural diagram, shows the **arrangement and organization of model elements in middle to large scale project**. Package diagram can show both structure and dependencies between sub-systems or modules, showing different views of a system, for example, as multi-layered (aka multi-tiered) application — multi-layered application model.

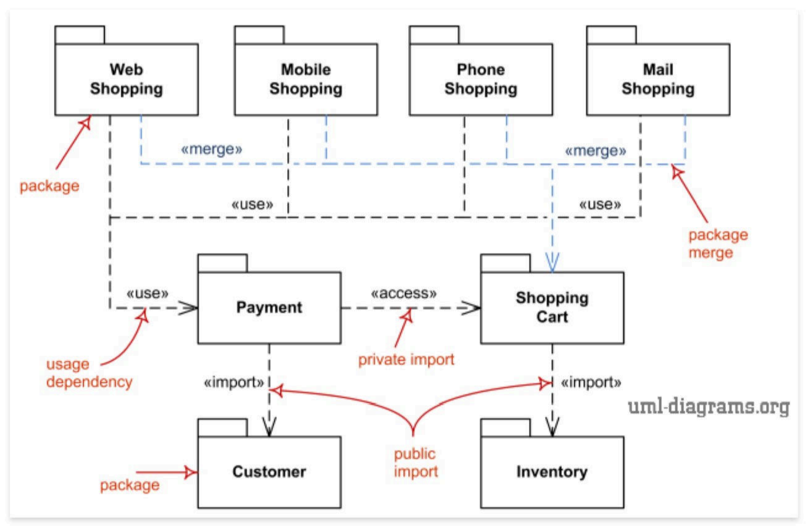



Figure 4.39: Package Diagram

### 4.3.5 Deployment Diagram (allocation structure)

A **Deployment Diagram** is a diagram that shows the configuration of runtime processing nodes and the components that live on them. Deployment diagrams is a kind of structure diagram used in modeling the physical aspects of an object-oriented system. They are often be used to model the static deployment view of a system (topology of the hardware).

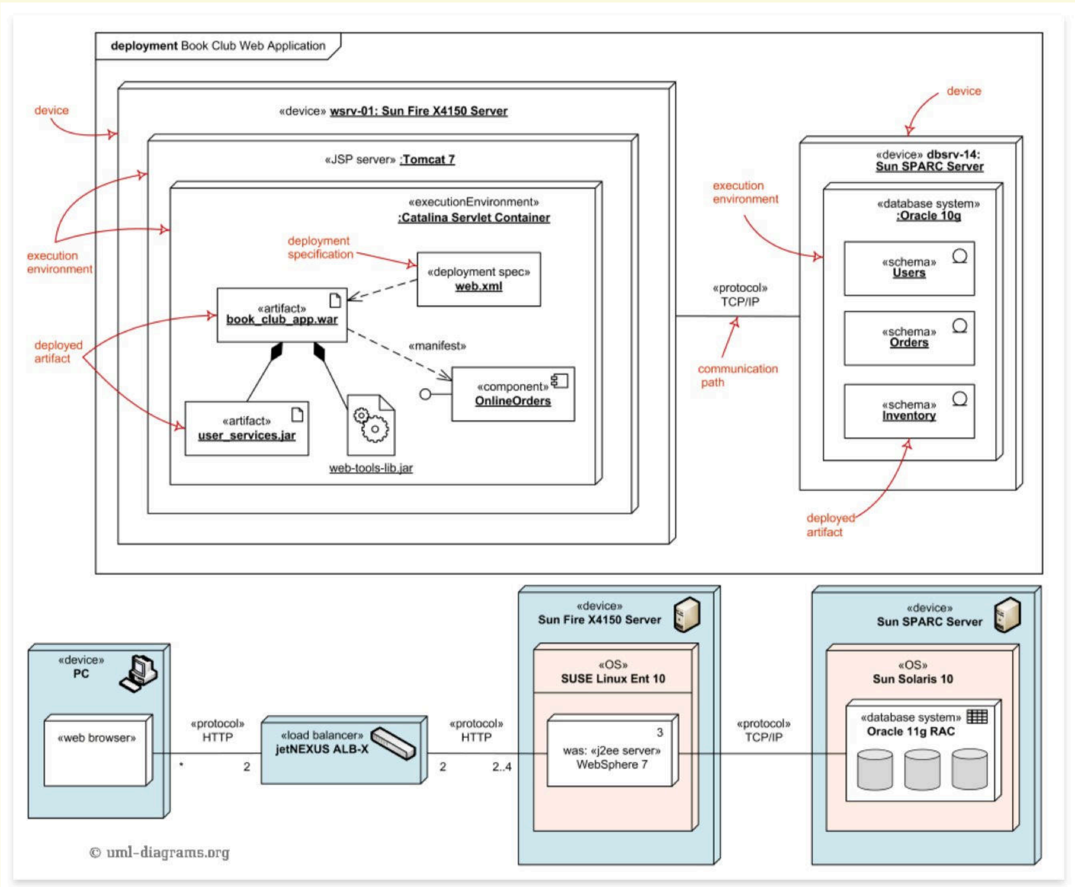



Figure 4.40: Deployment Diagram

## 4.4 Design Principal

### 4.4.1 Divide and Conquer

Divide and Conquer is a **problem-solving strategy** that involves breaking down a complex problem into smaller, more manageable parts, solving each part individually, and then combining the solutions to solve the original problem.

### 4.4.2 Keep the level of abstraction as high as possible

Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity. A good abstraction is said to provide information hiding. Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details.

### 4.4.3 Increase cohesion where possible

In general, a file, module, class or whatever should contain the same logical methods. For example, in the following class we have two functions with two different purposes (error!).

```java
Class Utility {
  ComputeAverageScore ( Student s [])
  ReduceImage ( Image i )
}
```

### 4.4.4 Reduce coupling where possible

Coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules. There are different types of couplings:

- **Content Coupling** is said to occur when one module uses the code of another module, for instance a branch. This violates information hiding (2nd design principle).
- **Communication coupling** is said to occur when one module sends too many messages to another module. The creation of a message can be optimized and the number of messages sent between these two modules can be reduced.
- **Control coupling** is one module controlling the flow of another, by passing it information on what to do. For example, passing a what-to-do flag or the following code:

```java
class b {
    func(flag f) {
        if (f == flag1) do this
        else if (f == flag2) do that
        else...
    }
}
```

### 4.4.5 Design for reusability

Design the various aspects of your system so that they can be used again in other contexts. To do this, you need to follow these rules:

- Generalize your design as much as possible;
- Simplify your design as much as possible;
- Follow the preceding all other design principles;
- Design your system to be **extensible**.

### 4.4.6 Reuse existing designs and code

Design with reuse is **complementary** to design for reusability. Take advantage of the investment you or others have made in reusable components. Note: cloning should not be seen as a form of reuse.

### 4.4.7 Design for flexibility

Actively anticipate changes that a design may have to undergo in the future, and prepare for them. To do this, you need to follow these rules:

### 4.4.8 Anticipate obsolescence

Plan for changes in the technology or environment so the software will continue to run or can be easily changed. So do not rush using early releases of technology. If possible:

- Avoid using software libraries that are specific to particular environments;
- Avoid using undocumented features or little-used features of software libraries;
- Avoid using software or special hardware from companies that are less likely to provide long-term support;
- Use standard languages and technologies that are supported by multiple vendors.

### 4.4.9 Design for portability

Have the software run on as many platforms as possible. Avoid, if possible, the use of facilities that are specific to one particular environment (e.g. a library only available in Microsoft Windows).

### 4.4.10 Design for testability

Design the system so that it can be tested easily. To do this, you need to follow these rules:

- Ensure that all the functionality of the code can be driven by an external program, bypassing a graphical user interface;
- Create proper code to exercise the other methods/functions;
- Use unit test automation frameworks.

### 4.4.11 Design defensively

Be careful when you trust how others will try to use a component you are designing. Handle all cases where other code might attempt to use your component inappropriately. Check that all of the inputs to your component are valid: the preconditions. Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking.

# 5 Architectural Style

An architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined.

## 5.1 Client-Server

A **Client-Server Architecture** is a **network-based computing structure** where responsibilities and operations get **distributed between clients and servers**. Client-Server Architecture is widely used for network applications such as email, web, online banking, e-commerce, etc.

**When to use it** **tip 5.1.1**

The three most common cases are:

- When **multiple users** need to access a **single resource** (e.g. database).
- When there is a **preexisting software** and we must access remotely (e.g. email server).
- When it is convenient to organize the system around a shared **piece of functionality used by multiple components** (e.g. authentication or authorization server).

With this architecture, it's necessary to design and **document proper interfaces** for our server. It is also necessary to ensure that the server can handle **multiple simultaneous requests**.

### 5.1.1 Interface Design

An interface is a **boundary** across which components interact. Proper definition of interfaces is an architectural concern (affects maintainability, usability, testability, performance, integrability).

There are two important guiding principles for interface design: **information hiding and low coupling**. An interface should **encapsulate** a component implementation so that it can be changed without affecting other components. There are several aspects to interface design that need to be considered:

- **Contract principle**: any resource (operation, data) added to an interface implies a commitment to maintaining it.
- **Least surprise principle**: interfaces should behave consistently with expectations.
- **Small interfaces principle**: interfaces should limit the exposed resources to the minimum.

There are also some important elements to define: **interaction style** (e.g. sockets, RPC, REST); representation and **structure of exchanged data** (affecting expressiveness, interoperability, performance and transparency); **error handling**.

### 5.1.2 Error handling, multiple interfaces and interface evolution

Sometimes there may be some problems, for example: an operation is called with invalid parameters and consequently the call doesn't return anything. This simple example can provoke some scenarios: the component cannot handle the request in its current state; or hardware/software errors prevent successful execution; or there is a misconfiguration issue (e.g. the server is not correctly connected to the database).

A **server** can offer **multiple interfaces** at the same time. This enables separation of concerns, different levels of access rights and support ot **interface evolution**.

**Interface evolution** occurs for many reasons (e.g. to support new requirements). Several strategies are needed to support continuity:

- **Deprecation**: declare well in advance that an interface version will be retired by a certain date;
- **Versioning**: maintain multiple active versions of the interface;
- **Extension**: a new version extends the previous one.

### 5.1.3 Handling multiple requests

The server must be able to receive and process requests from multiple clients. There are two main approaches to this: *forking and worker pooling.*

#### Forking

The forking approach is the same as that used by the Apache Web Server: one process per request or per client.

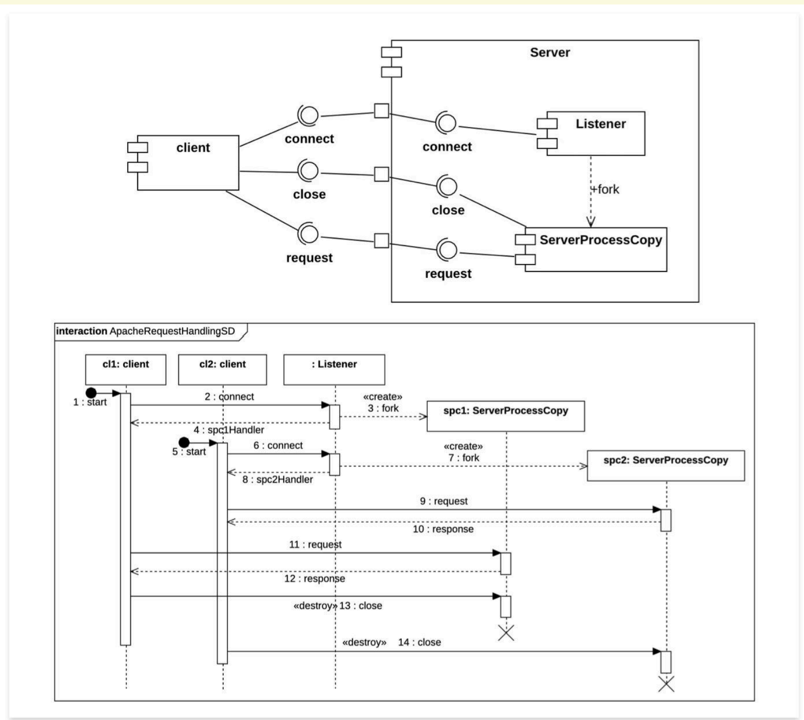



Figure 5.41: Forking diagrams

Forking Advantages:

- Architectural **simplicity**.
- **Isolation** and **protection** given by the one-connection-per-process model. Note: slow processes do not affect other incoming connections.
- **Simple to program**.

Forking Issues:

- Growth of the WWW over the last 20 years (number of users and weight of web pages).
- The number of **active processes** at time $t$ is **difficult to predict** and may **saturate resources**.
- **Expensive** fork-kill operations for each **incoming connection**.

**Worker pooling**

It is an alternative approach adopted by NGINX Web Server. It is designed for high concurrency but has to deal with scalability issues.

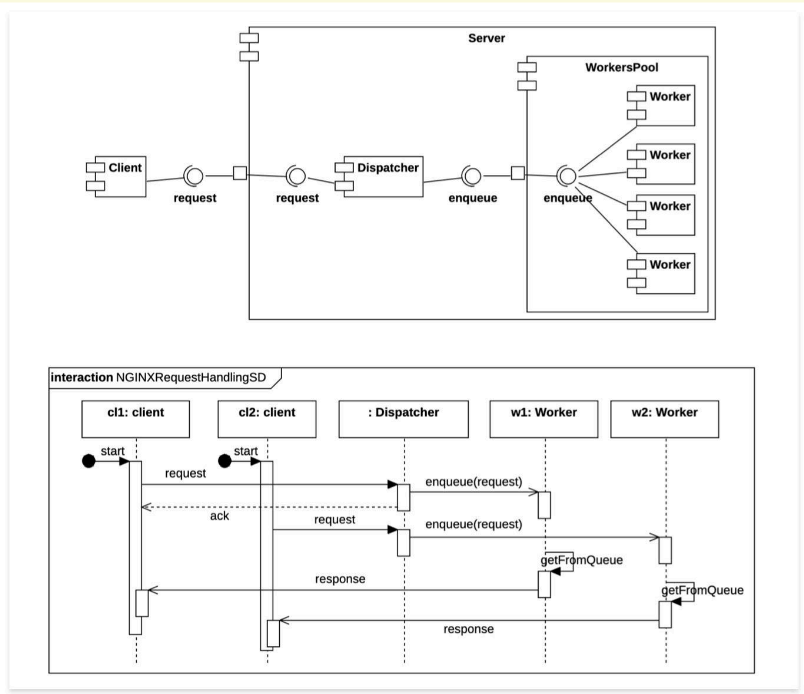



Figure 5.42: Worker pooling diagrams

Despite the well-known problem of this architecture (scalability), NGINX addresses the previous problems by introducing a new **architectural tactic**. A tactic is a design decision that affects the control of one or more quality attributes.

Worker Pooling Advantages (quality attribute trade-offs)

- **Number of workers** is **fixed**, so they do not saturate available resources.
- **Each worker** has a **queue**.

- When **queues** are **full** the dispatcher drops the incoming requests to keep high performance (optimize scalability and performance by sacrificing availability).
- Dispatcher balances the workload among available workers according to specific policies.

## 5.2 Three-Tier Architecture

**Three-tier architecture** is a well-established software application architecture that organizes applications into three logical and physical computing tiers:

- The **presentation tier**, or user interface;
- The **application tier**, where data is processed;
- The **data tier**, where application data is stored and managed.

### 5.2.1 Benefits

The chief benefit of three-tier architecture is its logical and physical separation of functionality. Each tier can run on a separate operating system and server platform - for example, web server, application server, database server -that best fits its functional requirements. And each tier runs on at least one dedicated server hardware or virtual server, so the services of each tier can be customized and optimized without impacting the other tiers. Other benefits include:

- **Faster development**: Because each tier can be developed simultaneously by different teams, an organization can bring the application to market faster. And programmers can use the latest and best languages and tools for each tier.
- **Improved scalability**: Any tier can be scaled independently of the others as needed.
- **Improved reliability**: An outage in one tier is less likely to impact the availability or performance of the other tiers.
- **Improved security**: Because the presentation tier and data tier can't communicate directly, a well-designed application tier can function as an internal firewall, preventing SQL injections and other malicious exploits.

## 5.3 Microservice architectural style

The microservice architectural style is an approach to developing a single application as a suite of **small services**, each running in its own process and communicating **lightweight mechanisms**, often an HTTP resource API.

There are two main benefits:

- **Technology heterogeneity. Each service uses its own technology stack**. The technology stack can be selected to fit the task best (e.g. data analysis vs video streaming). The teams can experiment with new technologies within a single microservice (e.g. we can deploy two versions and do A/B testing). Also, no unnecessary dependencies or libraries for each service.

- **Scaling. Each microservice can be scaled independently**. Also, identified bottlenecks can be addressed directly. Parts of the system that do not represent bottlenecks can remain simple and unscaled.

## 5.4 Event-Driven Architecture

An **Event-Driven Architecture** uses **events to trigger and communicate between decoupled services** and is common in modern applications built with microservices. An event is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website. Events can either carry the state (the item purchased, its price, and a delivery address) or events can be identifiers (a notification that an order was shipped).

Often it's called **publish-subscribe** (publish is the event generation, and subscribe is the declaration of the interest).

### 5.4.1 Benefits

- **Very common in modern development practices** (e.g. continuous integration and deployment, such as GitHub Actions).
- **Easy addition/deletion of components** (publishers and subscribers are decoupled; the event dispatcher handles this dynamic set).

### 5.4.2 Problems

- **Potential scalability problems** (the event dispatcher may become a bottleneck under high workload).
- **Ordering of events** (not guaranteed, not straightforward).

Other characteristics of this architecture:

- The messages and the events are **asynchronous**.
- Computation is **reactive** (driven by receipt of message).
- **Destination** of messages determined by receiver, not sender (location/identity abstraction).
- **Loose coupling** (senders and receivers added without reconfiguration).
- **Flexible** communication means (one-to-many, many-to-one, many-to-many).

Some examples of relevant technologies are: `Apache Kafka` and `RabbitMQ`.

### 5.4.3 Apache Kafka Architecture

Kafka is a framework for the event-driven paradigm:

- Includes primitives to create **event produces** and **consumers** and a runtime infrastructure to handle **event transfer** from producers to consumers.
- **Stores events** durably and reliably.
- Allow **consumers** to process events as they occur or retrospectively.

These services are offered in a **distributed, highly scalable, elastic, fault-tolerant, and secure manner**.

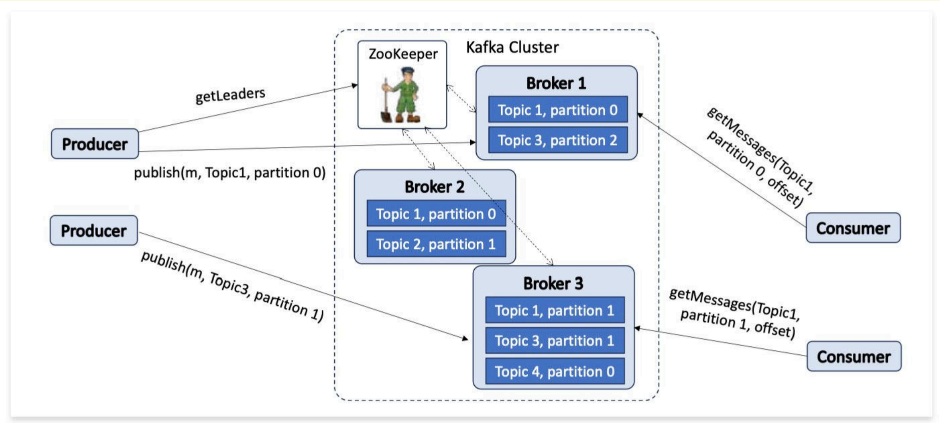



Figure 5.43: Kafka architecture (the ZooKeeper is a "health manager")

Some important features:

- Each **broker** handles a set of **topics** and **topic partitions**, parts including sets of messages on the topic.
- The partitions are independent from each other and can be **replicated** on multiple brokers for fault tolerance.
- There is one **leading broker** per partition. The other brokers containing the same partition are followers.
- The **producers** know the available leading brokers and send messages to them.
- Messages in the same topic are organized **in batches** at the producers'side and then sent to the broker when the batch size overcomes a certain threshold.
- Consumers adopt a **pull approach**. They receive in a single batch all messages belonging to a certain partition starting from a specified offset.
- Messages remain available at the brokers' side **for a specified period** and can be **read multiple times** in this period.
- The leader keeps track of the **in-synch followers**.
- **ZooKeeper is used to monitor the correct operation of the cluster.** All brokers send heartbeats to ZooKeeper. ZooKeeper will replace a failed broker by electing a new leader for all partitions that the failed broker was leading. It can also start/restart brokers.

**Producer**

1. Brokers commit messages by storing them in the corresponding partition;
2. Leader adds the message to followers (replicas) if available.

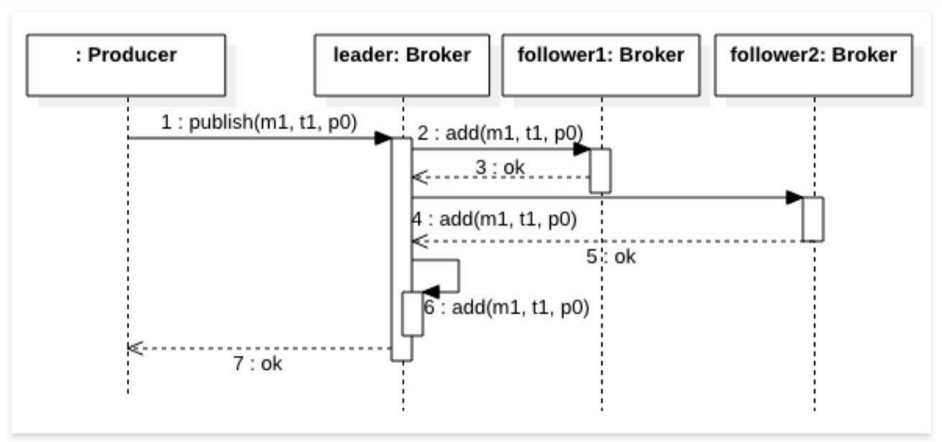



Figure 5.44: Sequence diagram of a producer sending a message to a Kafka broker

A possible **issue**: in case of failure, the producer may not get the response (message number 7 in figure). In this case, the producer has to resend the message and kafka brokers can identify and eliminate duplicates.

Synchronization with replicas can be transactional and it's possible to choose between the following options:

- **Exactly-once** semantics is possible but long waiting time. So **replicas are not allowed**, but the problem is that Kafka spent a long time trying to guarantee uniqueness.
- **At-least-once** can be chosen by excluding duplicates' management.
- **At-most-once** can be chosen by publishing messages asynchronously.

**Consumer**

Each consumer can rely on a persistent log to keep track of the offset so that it is not lost in case of failure.

**Issue case**: if the consumer fails after having elaborated messages and before storing the new offset in the log, the same messages will be retrieved again (**at-least-once semantics**). Note that the delivery semantics can be changed if the new offset is store before the elaboration and we can choose **at-most-once semantics** because, if failing after storing the offset, the effect of the received messages does not materialize. Finally, transactional management of the log also allows **exactly-once semantics**.

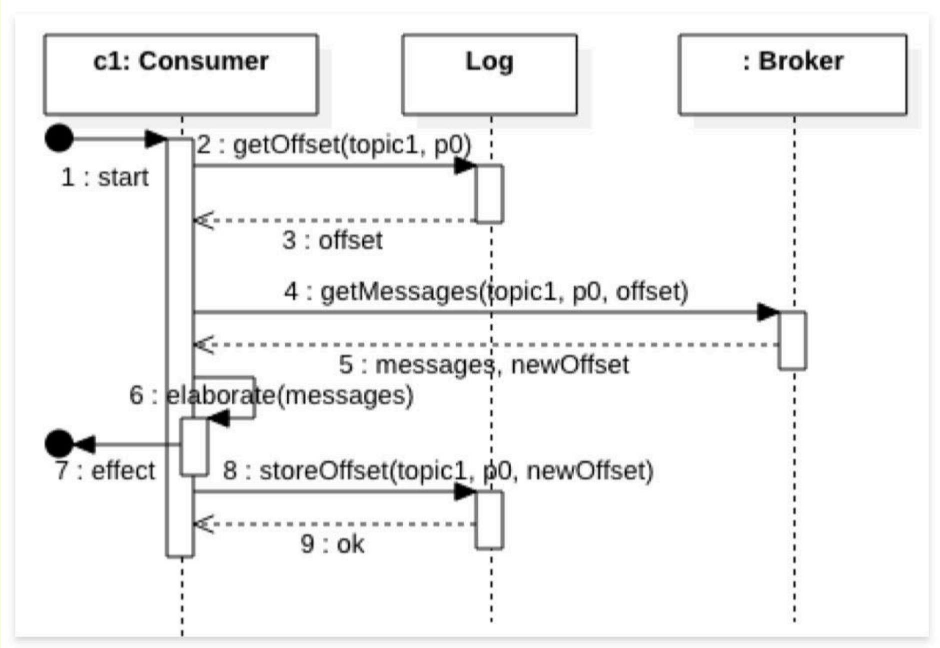



Figure 5.45: Sequence diagram of a consumer reading messages from a Kafka broker

### 5.4.4 Kafka architectural tactics

There are some tactics used to improve some features of Kafka. In the following section we can see scalability and fault tolerance.

#### Improve Scalability

By **creating multiple partitions and multiple brokers**, we can create the ability to distribute producers/consumers to different partitions handled by different brokers. We can also **scale the operations** because Kafka supports the **creation of clusters of brokers**. Consider that each cluster contains up to a hundred brokers capable of handling trillions of messages per day.

#### Improve Fault Tolerance

By **creating partitions**, we use the **persistence** of the partitions. Replication also reduces the risk of data loss. Finally, cluster management takes care of restarting brokers and setting leaders as needed.

## 5.5 Data-Intensive applications

Before we introduce the architectural styles for data-intensive applications, we explain the difference between **batch and stream processing**.

**Batch processing** is a method of running software programs called jobs in batches automatically. While users are required to submit the jobs, no other interaction by the user is required to process the batch.

**Stream processing** (also known as event stream processing, data stream processing, or distributed stream processing) is a programming paradigm which views streams, or sequences of events in time, as the central input and output objects of computation.

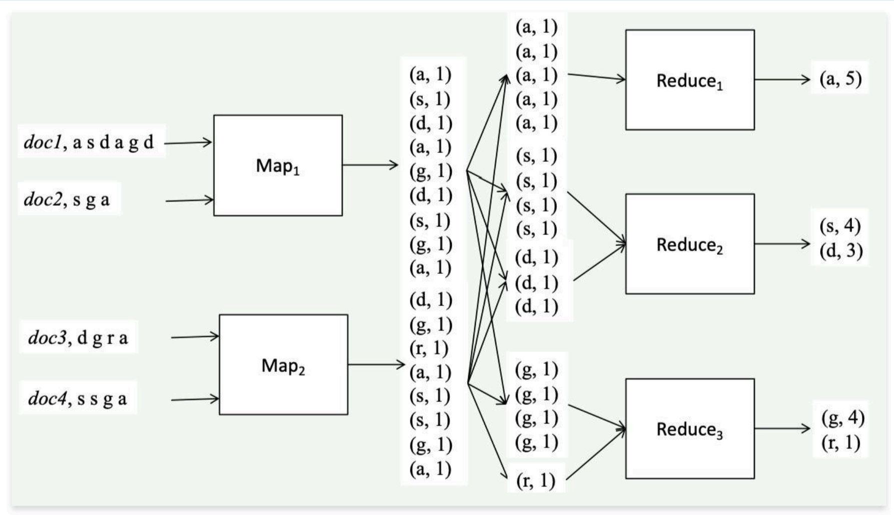
| Batch | Stream |
|---|---|
| Has access to all data. | Computes a function of one data element, or a smallish window of recent data. |
| Might compute something big and complex. | Computes something relatively sim-ple. |
| Is generally more concerned with throughput than latency of individual components of the computation. | Needs to complete each computation in near-real-time — probably seconds at most. |
| Has latency measured in minutes or more. | Computations are generally independent. |
| | Asynchronous — source of data doesn't interact with the stream processing directly, like by waiting for an answer. |

### 5.5.1 Batch approach: MapReduce

**MapReduce** is a **programming architecture** and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.

A MapReduce is composed of a **map procedure**, which performs filtering and sorting (such as sorting students by first name into queues, one queue for each name), and a **reduce method**, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "MapReduce System" (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

**e.g. an example of a batch approach using MapReduce** **example 5.5.1**




The workflow is the following:

1. Read a set of input files and break it into records;
2. Call the **map** function. It extracts a key and a value from each record (the assigned value is application-dependent);
3. Sort all the key-value pairs by key;
4. Call the reduce function. It iterates over the ordered sets of key-value pairs and combines the values (the combination logic is application-dependent)

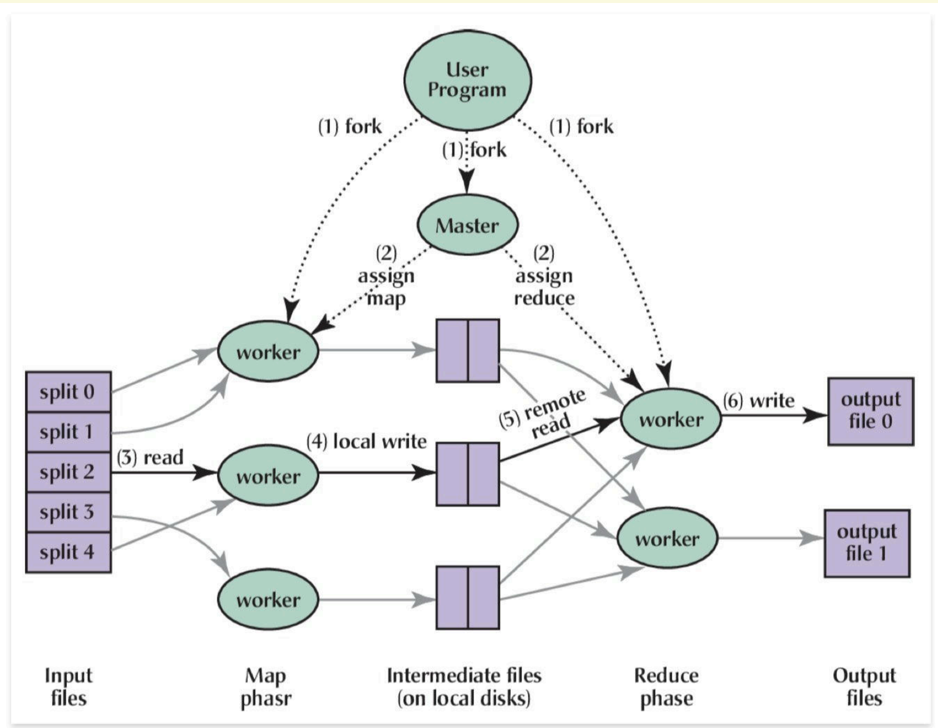



Figure 5.47: MapReduce architecture

**Advantages**

- Works well on **commodity hardware**: **Commodity hardware** in computing is computers or components that are readily available, inexpensive and easily interchangeable with other **commodity hardware**. Almost all PCs use **commodity hardware**.

**Disadvantages**

- Implementing a complex processing job is not simple (high level programming model have been built on top of it);
- Reducers have to wait until the preceding Mappers have concluded their job;
- Materialization of intermediate states can be overkilling;
- Sometimes it is not necessary to sort the results of mappers;
- New batch computation approaches supported by frameworks as Spark, Tez, Flink, etc.

#### 5.5.2 Stream approach: Apache Storm

**Apache Storm** is a **distributed stream processing computation framework** written predominantly in the Clojure programming language. Originally created by Nathan Marz and team at BackType, the project was open sourced after being acquired by Twitter. It uses custom created “spouts” and “bolts” to define information sources and manipulations to allow batch, distributed pro-cessing of streaming data.

Some features:

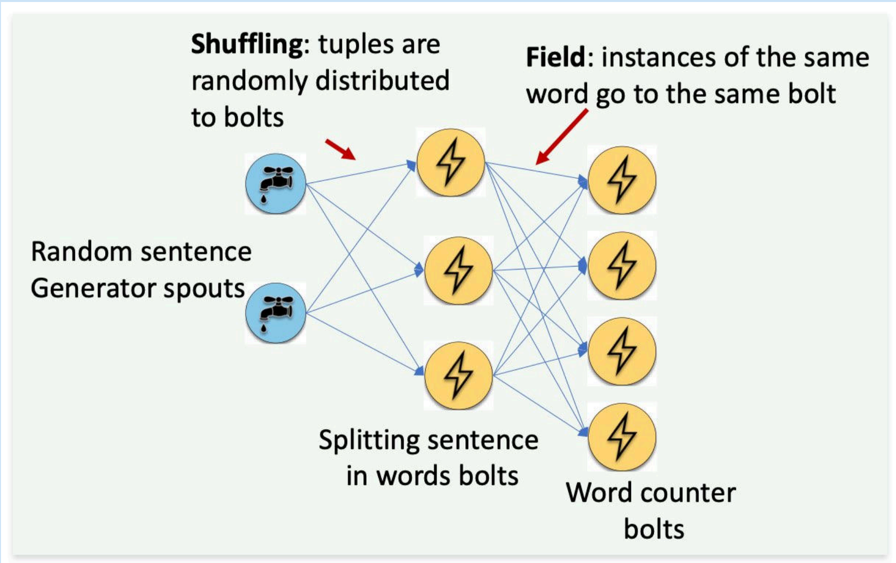
- Support stream processing.
- More than 1 million messages per second per node.
- Can scale up to thousands of nodes per cluster.
- Expects and manages failures (fully fault tolerant).
- Provides guaranteed message delivery with exactly once semantics (reliable).

A Storm application is designed as a “topology” in the shape of a **directed acyclic graph (DAG)** with **spouts** (source of streams) and **bolts** (receives messages) acting as the graph vertices. **Edges on the graph are named streams** and direct data from one node to another. Together, the topology acts as a data transformation pipeline. At a superficial level the general topology structure is similar to a MapReduce job, with the main difference being that data is processed in real time as opposed to in individual batches. Additionally, Storm topologies run indefinitely until killed, while a MapReduce job DAG must eventually end.

| Stream Grouping | Description |
|---|---|
| **Shuffle** | Sends messages to bolts in random, round robin sequence. Use for atomic operations, such as math. |

| | |
|---|---|
| **Fields** | Sends messages to a bolt based on one or more fields in the tuple. Used to segment an incoming stream and to count tuples of a specified type with a specified value. |
| **All** | Sends a single copy of each message to all instances of a receiving bolt. Use to send a signal, such as clear cache or refresh state, to all bolts. |
| **Custom** | Customized processing sequence. Use to get maximum flexibility of topology processing based on factors such as data types, load, and seasonality. |
| **Direct** | Source decides which bolt receives a message. |
| **Global** | Sends messages generated by all instances of a source to a single target instance. Use for global counting operations. |

**e.g. example of topology with different groupings** **example 5.5.2**




### 5.5.3 Combining batch and stream: Lambda Architecture

**Lambda architecture** is a **data-processing architecture** designed to handle massive quantities of data by taking advantage of both batch and stream-processing methods.

This approach to architecture attempts to balance latency, throughput, and fault-tolerance by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using real-time stream processing to provide views of online data. The two view outputs may be joined before presentation.

The rise of lambda architecture is correlated with the growth of big data, real-time analytics, and the drive to **mitigate the latencies of map-reduce**.

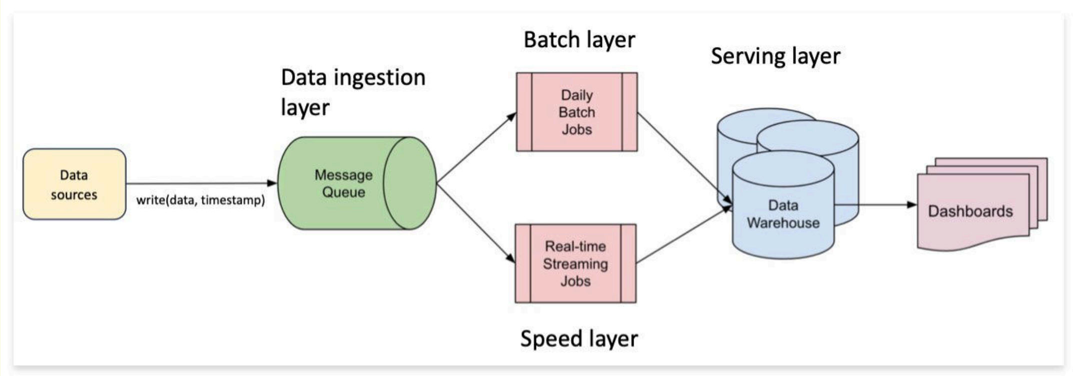



Figure 5.49: Lambda architecture

Exist also **Kappa architecture**. Kappa architecture is a software architecture used for processing streaming data with a single technology stack. It is a simplification of Lambda architecture, where the data is processed in batches. Kappa architecture ingests data into a messaging system like Apache Kafka, and performs both real-time and batch processing, especially for analytics, on the same stream. It allows for recomputation on the data by streaming it through the pipeline again.

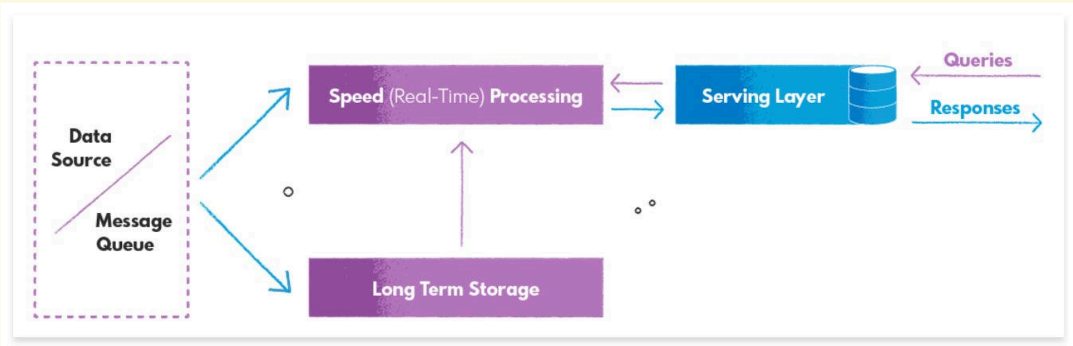


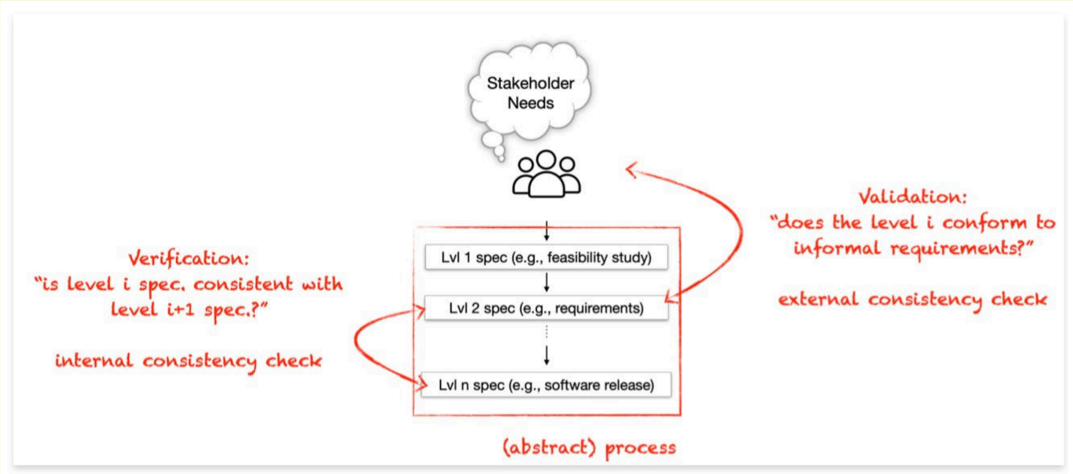

Figure 5.50: Kappa architecture

# 6 Verification and Validation

## 6.1 Terminology

There are some differences between the terms **verification** and **validation**.

First of all, the **verification** is internal, while the **validation** is external. Assuming an abstract process with the following levels:




The verification is intended as: "Is level $i$ consistent with level $i + 1$?". It's an internal consistency check. The validation is: "Does level i conform to needs?". It's an external consistency check.

Another fundamental topic when we speak about verification and validation is **Quality Assurance** (QA). It **defines the policies and processes to achieve quality**. So it can **judge the quality and find defects**.

A direct **consequence** of the QA is the improvement of the quality. With the term "quality", we refer to an ideal absence of defects (impossible) and an absence of other issues that prevent the fulfillment of non-functional requirements or the degradation of some software qualities.

Since it is impossible to have zero defects, a **periodic quality assurance evaluation is critical**. Ideally, every artifact shall be the subject of QA; even the verification artifact must be verified!

The V-model is a **graphical representation of a systems development lifecycle**. It is used to produce rigorous development lifecycle models and project management models. It describes the activities and the results that must be made during product development.

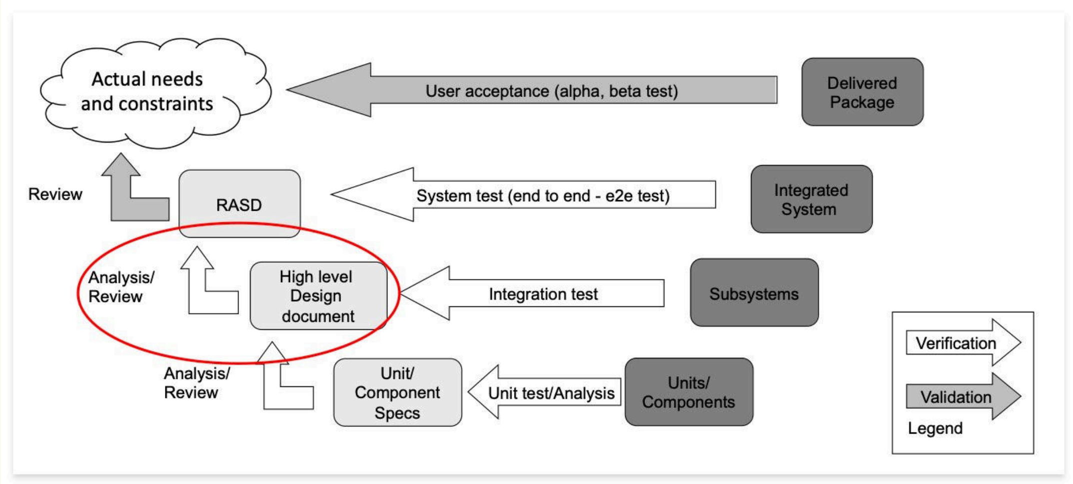



Figure 6.52: V-model

The *left side* of the "V" represents the decomposition of requirements and the creation of system specifications. The *right side* of the "V" represents an integration of parts and their validation.

We have presented the V-model to help you understand where the verification can be placed. Now, the **verification** is concerned with the code and the architecture. Considering the **software** side, it has two possible approaches:

- **Static Analysis**. It is done using source code or other software artifacts but without execution. Note that the analysis is static, but the properties are dynamic.
- **Dynamic Analysis (Testing).** It is done by executing the sources. The analysis is made by comparing the actual behavior and the expected one.

On the other hand, to verify the architectural level, it is necessary to consider some aspects:

- The **structure must be consistent**. Some examples:
  - For every required interface, a corresponding provided interface exists.
  - Sequence diagrams are consistent with component diagrams and with the defined interfaces.
  - Each component has one or more modules that implement it.
- All **functional requirements must have the possibility to be satisfied**. Some examples:
  - Each requirement is mapped on one or more components.
  - Each use case event flow is detailed in terms of one or more sequence diagrams.
- **Concurrent use of resources must be correctly defined.** Problems like order violation or a deadlock are expected. Some techniques must be applied to analyze these problems.
- **Non-functional requirements must have the possibility to be fulfilled**.

**Def DevOps** **definition 6.1.1**

DevOps is a **set of practices, tools, and a cultural philosophy that automate and integrate the processes between software development and IT teams**. It emphasizes team empowerment, cross-team communication and collaboration, and technology automation.

## 6.2 Petri Net

It is necessary to model distributed systems to study the concurrent use of resources at the architectural level.

A **Petri Net (PT Net or P/T Net)**, a place/transition net (PT net), is one of several mathematical modelling languages used to describe distributed systems. Like industry standards such as UML activity diagrams, **Petri nets offer a graphical notation for stepwise processes** that include choice, iteration, and concurrent execution.

The Petri net uses a graphic tool. It is a bipartite-directed graph containing places (circles), transitions (bars), and directed arcs.

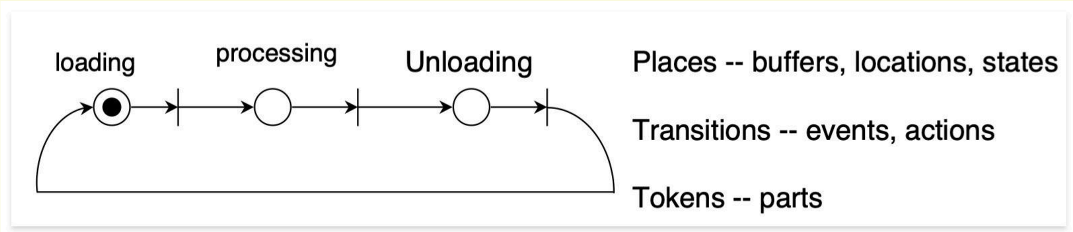



Figure 6.53: Petri net graphic

A Petri net is a four-tuple:

$$PN =< P, T, I, O > \tag{6.1}$$

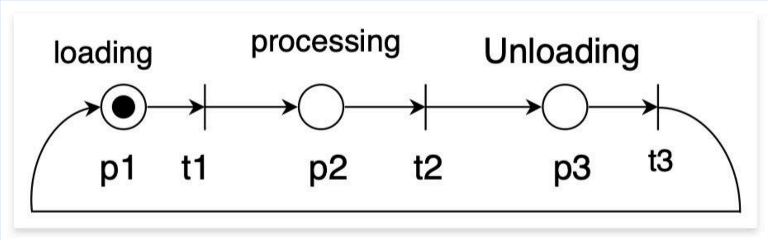
- $P$: a **finite set of places** $\{p_1, p_2, ..., p_n\}$
- $T$: a **finite set of transitions** $\{t_1, t_2, ..., t_s\}$
- $I$: an **input function** $I : T \times P \longrightarrow \{0, 1\}$
- $O$: an **output function** $O : T \times P \longrightarrow \{0, 1\}$

It's also possible to add another term called $M^0$, which is an initial marking $P \longrightarrow N$:

$$PN =< P, T, I, O, M^0 > \tag{6.2}$$

Formula called also **marked Petri net**.

**An example of a Petri net** **example 6.2.1**




- $P = \{p_1, p_2, p_3\}$
- $T = \{t_1, t_2, t_3\}$
- $I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, O = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$
- $M^0 = (1, 0, 0)$

Note:

- $p_1$ is the input place of transition $t_1$
- $p_2$ is the output place of transition $t_1$

Some observations of the Petri net:

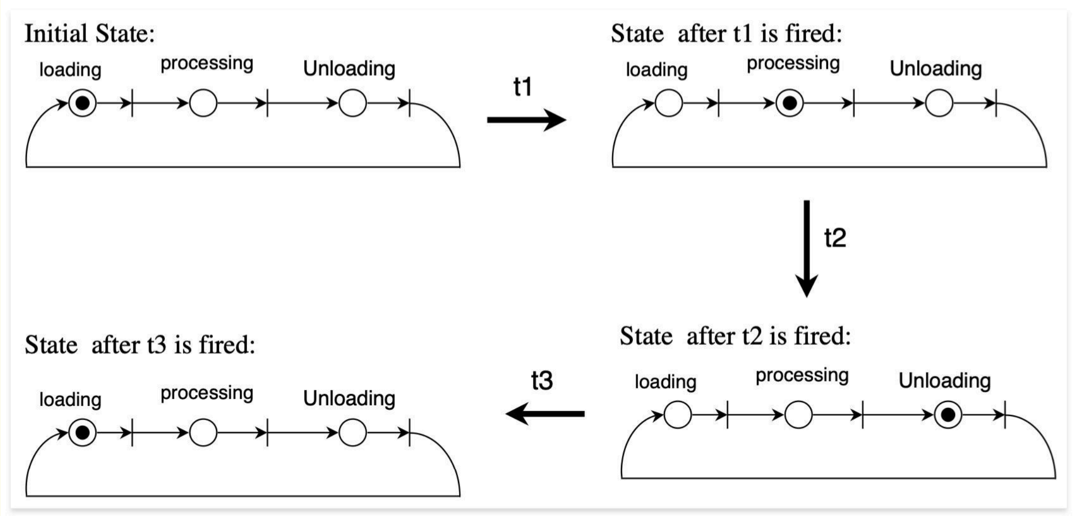
- In a given marking $M$, a transition $t$ can fire only if it is enabled.
- An enabled transition not necessarily fires.
- More than one transition can be enabled in a marking.
- If two transitions are enabled at the same time:
  - Which one fires first is not determined;
  - Petri nets are an intrinsically nondeterministic model;
  - The firing of a transition might disabled another enabled transition.

### 6.2.1 Dynamics

**Enabling Rule**: A transition $t$ is enabled if every input place contains at least one token.

**Firing Rule**: Firing an enabled transition

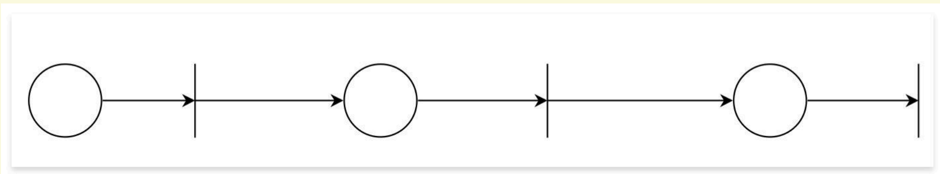
- removes one token from each input place of the transition
- adds one token to each output place of the transition




### 6.2.2 Basic Constructs

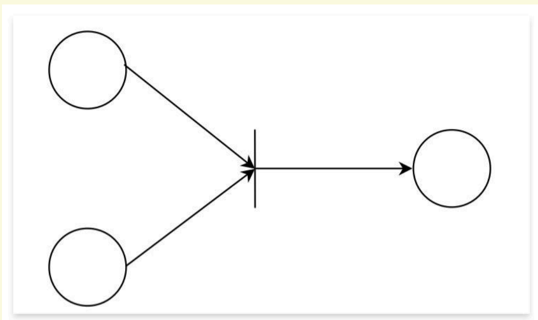
#### Sequential Actions

Each action is a transition.



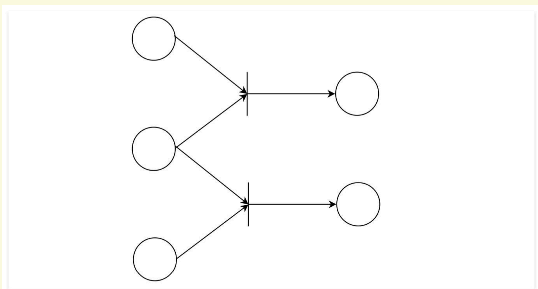
#### Dependency

A transition requires two inputs.



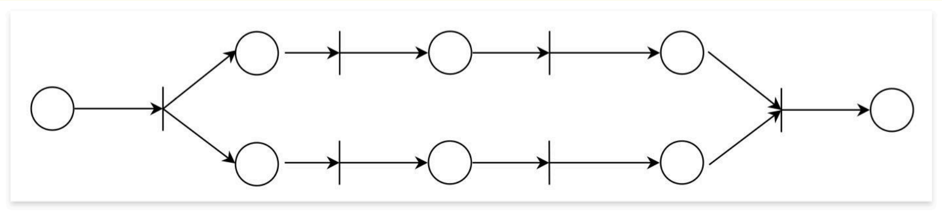
#### Conflict Construct

Only one of the two transitions can fire.



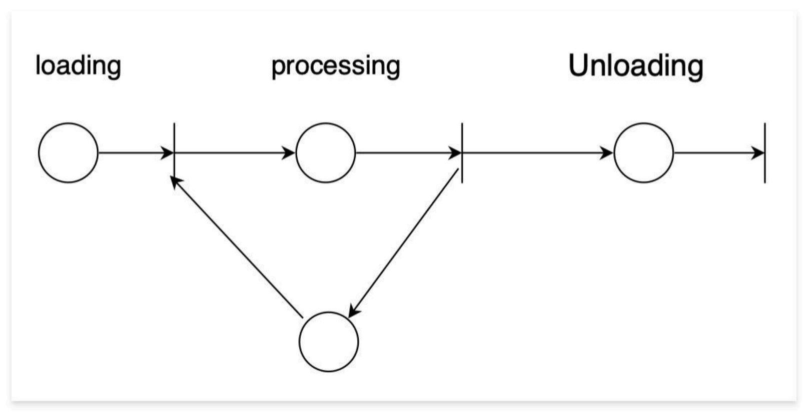
### Concurrency Construct

These two sequences can occur simultaneously.



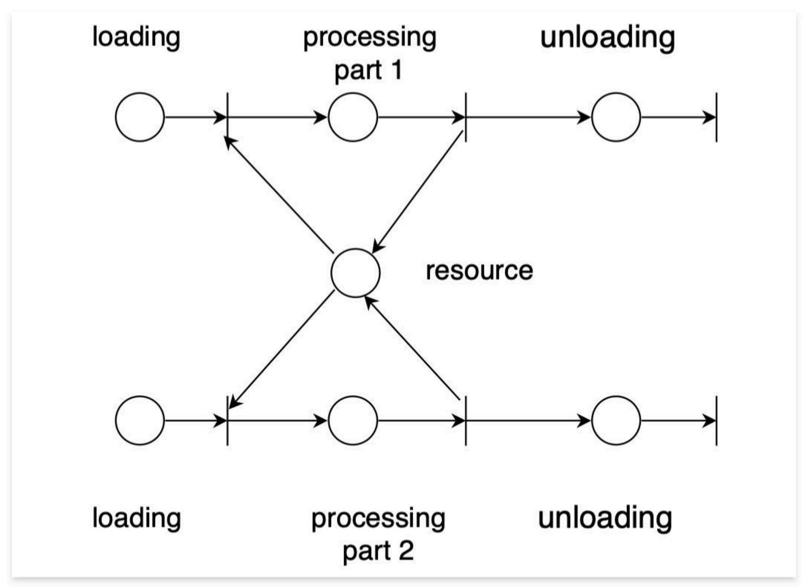
### Synchronization

Machine can process one part at once.



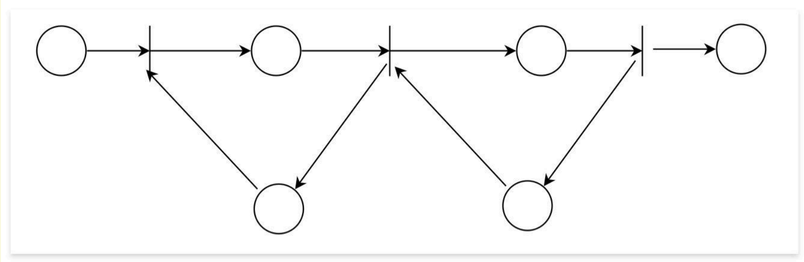

### Resource Sharing

One worker for two machines. The worker can work at one machine at a time.




### Buffer (Queue)

The buffer can hold a limited number of parts.



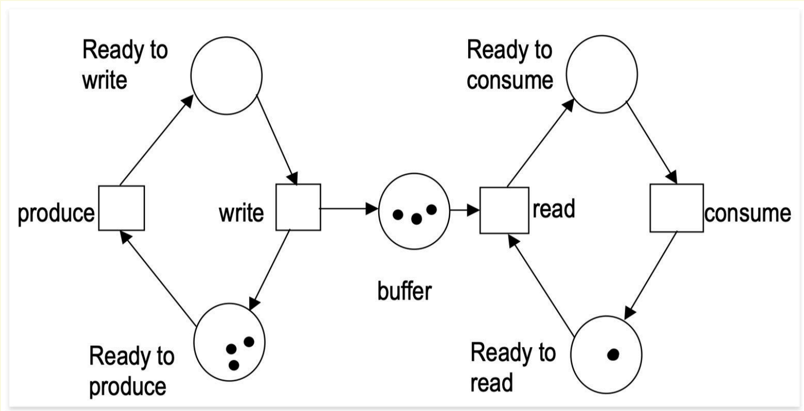



Figure 6.63: Example of Petri nets of producer-consumer model with unbounded buffer.

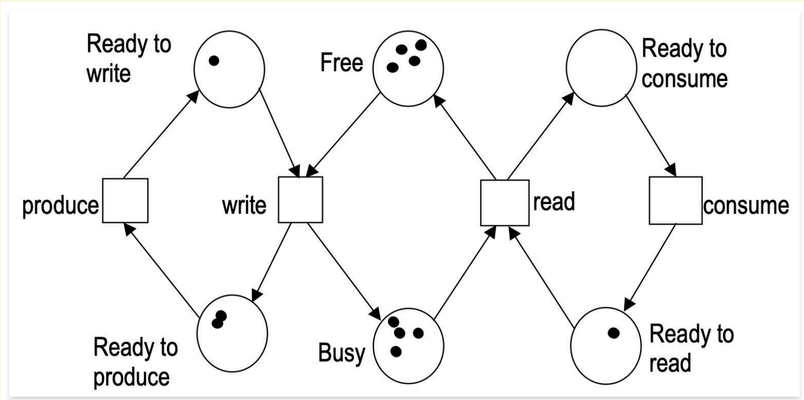



Figure 6.64: Example of Petri nets of producer-consumer model with finite buffer with a parametric number of positions.

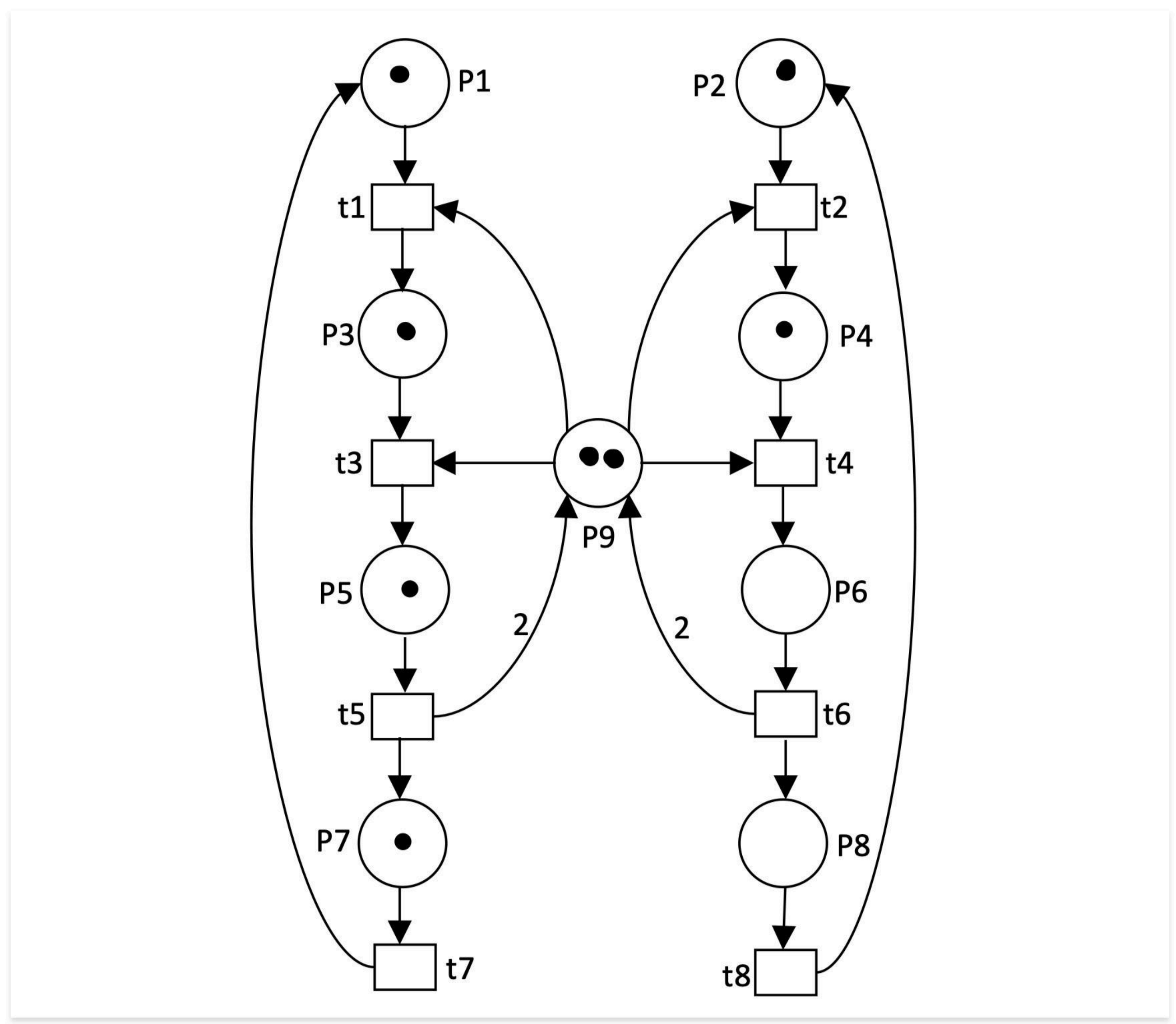



Figure 6.65: Example of Petri nets of deadlock.

## 6.3 Quantitative impact of architectural decisions

Architectural choices directly influence several software qualities (e.g., scalability, reliability, availability, usability).

To cope with this, we need metrics to quantify qualities and specific methodologies to analyze the quantitative impact of architectural choices on these qualities. The tactics are also foundational to address the issues.

First, before discussing how to evaluate the quantitative impact of architectural decisions, we must introduce the availability concept and explain a system life-cycle to introduce some exciting **metrics**.

In general, a **service shall be continuously available** to the user, and if it fails after a bit of downtime, it should be a **rapid service recovery**. So the availability of a service depends on:

- The **complexity** of the **infrastructure** architecture.

- **Reliability** of the individual components.
- **Ability to respond** quickly and effectively to faults.
- **Quality of the maintenance** by support organizations and suppliers.
- **Quality** and scope of the **operational management** processes.

### 6.3.1 System Life-Cycle

The System Life-Cycle relates to failures in the following way:

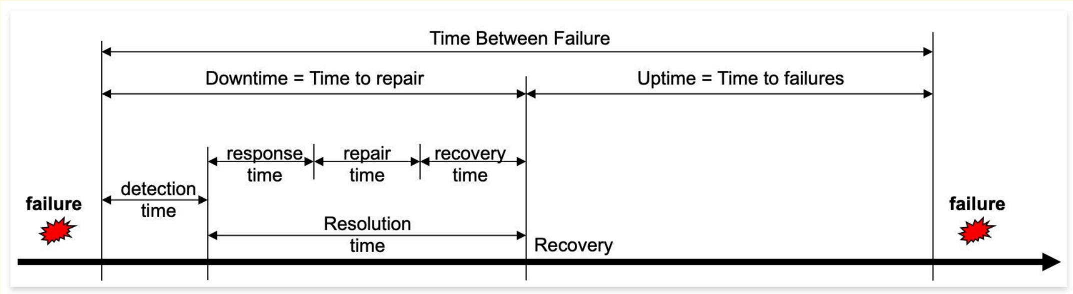



Figure 6.66: The System Life-Cycle when faults occur.

- **Time of occurrence**. Time at which the user becomes aware of the failure.
- **Detection time**. Time at which operators become aware of the failure.
- **Response time**. Time required by operators to diagnose the issue and respond to users.
- **Repair time**. Time required to fix the service/components that caused the failure.
- **Recovery time**. Time required to restore the system (re-configuration, re-initialization, ...).
- **Mean Time to Repair (MTTR)**. Average time between the occurrence of a failure and service recovery, also known as the *downtime*.
- **Mean Time to Failures (MTTF)**. Mean time between the recovery from one failure and the occurrence of the next failure, also known as *uptime*.
- **Mean Time Between Failures (MTBF)**. Mean time between the occurrences of two consecutive failures.

**Def Availability Metric** **definition 6.3.1**

The **availability metric** is the probability that a component works correctly at time $t$. As a mathematician term, we can express this definition as the relationship between the Mean Time to Failures (MTTF) and the MTTF plus the Mean Time to Repair (MTTR):

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \tag{6.3}$$

Note that if the Mean Time to Repair (MTTR) is small, then the Mean Time Between Failures (MTBF) is approximately equal to the Mean Time to Failures(MTTF): MTBF $\approxeq$ MTTF

Availability is typically specified in "**nines notation**". For example, 3-nines availability corresponds to 99.9%, 5-nines availability corresponds to 99.999% availability.

| Availability | Downtime per year |
|---|---|
| 90% (1-nine) | 36.5 days/year |
| 99% (2-nines) | 3.65 days/year |
| 99.9% (3-nines) | 8.76 hours/year |
| 99.99% (4-nines) | 52 minutes/year |
| 99.999% (5-nines) | 5 minutes/year |

### 6.3.2 Methodology

The Analysis Methodology depends on the system. The Availability is calculated by **modelling the system** as an interconnection of elements in series and parallel:

- **Elements operating in <u>series</u>** mean that if one element fails, the whole combination fails.
- **Elements operating in <u>parallel</u>** mean that if a component fails, the other elements take over the operations of the failed element.

**Availability in series**

The combined system is **operational only if every part is available**. Then, the combined Availability is the product of the parts' Availability.

$$A = \Pi_{i=1}^{n} A_i \tag{6.4}$$

**e.g. Example of availability in series** **example 6.3.1**

We assume there is a system composed of two components with the following availability and downtime:

- Component 1 has 99% (2-nines) of availability and 3.65 days/year of downtime.
- Component 2 has 99.999% (5-nines) of availability and 5 minutes/year of downtime.

So the combined availability is 98.999% with 3.65 days/year of downtime. This result means that a chain is as strong as the weakest link.

**Availability in parallel**

The combined system is operational if at least one part is available. Then, the combined Availability is $1 - P$, where $P$ indicates all parts that a re not available.

$$P = \Pi_{i=1}^{n} (1 - A_i) \tag{6.5}$$

**Example of availability in parallel** **example 6.3.2**

We assume there is a system composed by two components with the following Availability and downtime:

- Component 1 has 99% (2-nines) of Availability and 3.65 days/year of downtime.
- Component 2 has 99% (2-nines) of Availability and 3.65 days/year of downtime.

Despite the previous example, the combined availability is 99.99% (4-nines) with 52 minutes/year of downtime.

### 6.3.3 Tactics for Availability

As we explained in the past pages, Availability is crucial, but it's also fundamental to use intelligent tactics to improve the quality of the attributes.

**Tactics** **definition 6.3.2**

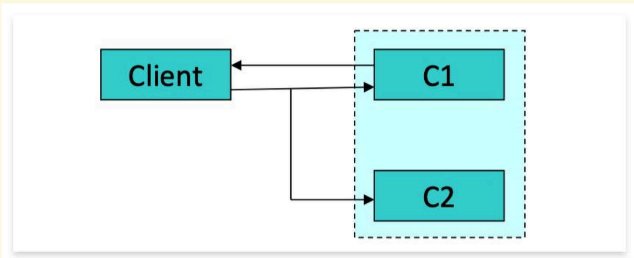
The **Tactics** are design decisions that influence the control of one or more quality attributes.

Some well-known tactics are:

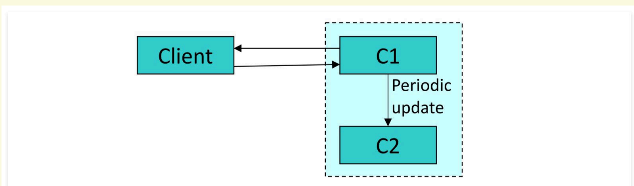
#### Replication approaches

The **Replication** is very simple to manage in the case of stateless components. The approaches are different:

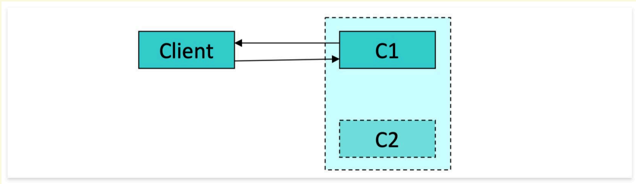
1. **Hot spare**: One component leads, and another is always ready to take over. In the following example, C1 leads, C2 is always ready to take over.



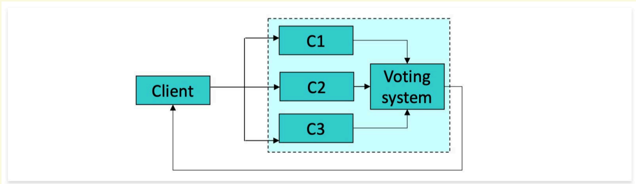

2. **Warm spare**: One component leads and periodically updates another component. If the primary component fails, the second component takes time to update itself fully. C1 is leading and periodically updating C2. If C1 fails, some time might be needed to fully update C2.



3. **Cold spare**: A second component is dormant, started, and updated only if required. In the following example, C2 is dormant, started, and updated only if required.

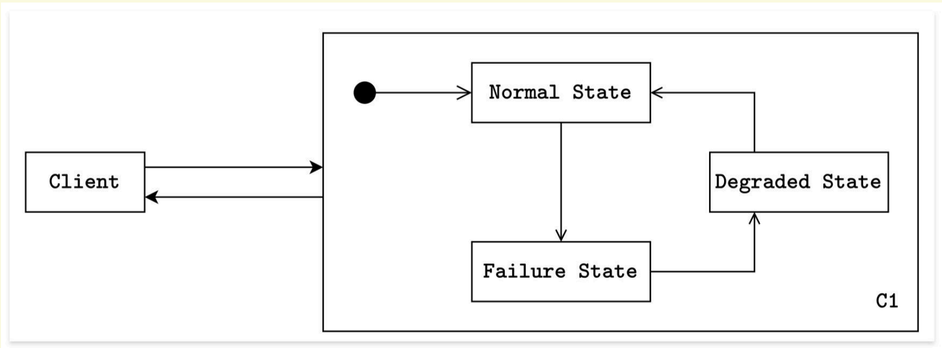



4. **Triple modular redundancy**: Three components are always active, and the result is the one produced by the majority. This is good when reliability is also important. In the following example, C1, C2, and C3 are all active. The result is the one produced by the majority.




**Forward error recovery**

**Forward Error Recovery** is a **tactic** in which a recovery mechanism moves the failed component to a degraded state. In a degraded state, a component continues to be available even if it is not fully functional. Here is an example:




**Circuit breaker**

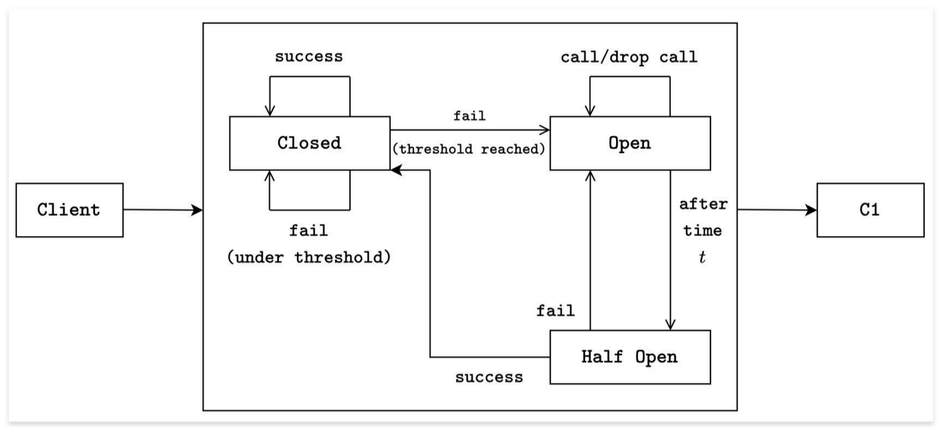
The **Circuit Breaker (CB)** tactic is a client-side resiliency pattern. The CB acts as a proxy for a remote component:

1. A component is called;
2. The CB monitors the call.

But note that there should be possible failures:

- CB receives an error;
- The call takes "too long" (CB kills the call).

If there are too many failures, the circuit breaker inhibits future calls by moving to the open state.




## 6.4 Static Analysis

**Static Analysis** analyzes the source code, and each analyzer targets a fixed set of hard-coded (pre-defined, not custom) properties. It is entirely automatic, and the output reports two types of results: **safe (no issues)** and **unsafe (potential problems)**. Also, the analysis is made on generic (or symbolic) inputs.

The properties that we have mentioned are safety properties, such as:

- No overflow for integer variables
- No type errors
- No null-pointer dereferencing
- No out-of-bound array accesses
- No race conditions
- No useless assignments
- No usage of undefined variables
- No execution of specific paths

Using the static analysis, we can use the symbolic execution.

**Def Symbolic Execution** **definition 6.4.1**

The **symbolic execution** is a technique to analyze the program by executing it with symbolic inputs instead of concrete values. The symbolic execution engine generates a set of constraints that must be satisfied for the program to reach a specific state.

The symbolic execution analyzes actual source code and reachability and path feasibility properties. It is automatic and may fail to explore all possible paths. Sometimes, it is used to support testing.

The checked properties by the static analysis can be of different types:

- **Reachability**. Does some program execution reach location L (generic line of code) in S (source code)? With the reachability property, the symbolic execution tries:
  - To verify that L cannot be reached;
  - Or spots the condition under which L can be reached.

  For example, in the following code:

```
k:    try {
k+1:       ...
L-1:     } catch (e) {
L:          /* error */
... }
```

Static analysis checks the reachability properties and verifies that $L$ cannot be reached, or discovers the condition under which $L$ can be reached.

- Path Feasibility. Is the given path $p$ feasible? With the path feasibility property, the symbolic execution tries:
  - To verify that $p$ cannot be executed;
  - Or **spots the condition under which p can be executed**.

  The $p$ will be

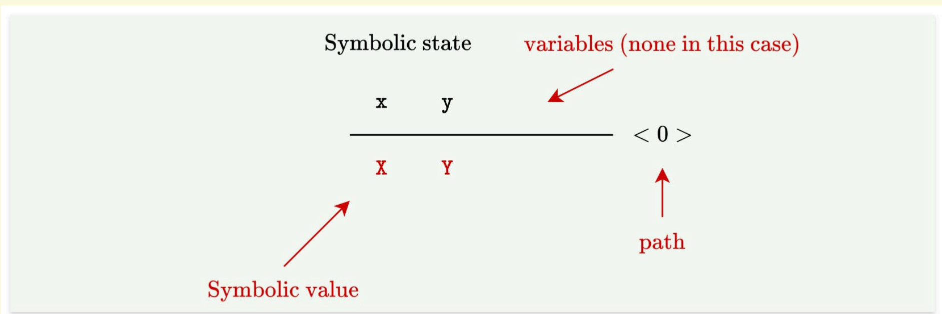
$$p =< 0, 1, ..., k, ..., n > \tag{6.6}$$

Symbolic execution **executes programs on symbolic values**. Each symbolic value has its **symbolic states**, which keep track of the variables' (symbolic) values. The inputs are initialized with symbolic (generic) values.

### 6.4.1 Example

First we introduce the annotation, inputs are initialized with symbolic (generic) values:

```
void foo(int x, int y) {
  ...
```




We introduce a local variable:

```
void foo ( int x , int y) {
  int z := x
```

Symbolic state

$$\begin{array}{ccc} \mathtt{x} & \mathtt{y} & \mathtt{z} \\ \hline \mathtt{X} & \mathtt{Y} & \color{red}{\mathtt{X}} \end{array} < 0, 1 >$$

We introduce a condition. A path condition $\pi$ represents a constraint on a path:

```
void foo ( int x , int y) {
  int z := x
  if (z < y)
```

- If condition is true

Symbolic state

$$\begin{array}{cccc} \mathtt{x} & \mathtt{y} & \mathtt{z} & \pi \\ \hline \mathtt{X} & \mathtt{Y} & \mathtt{X} & \color{red}{\mathtt{X} < \mathtt{Y}} \end{array} < 0, 1, 2 >$$

- If condition is false

Symbolic state

$$\begin{array}{cccc} \mathtt{x} & \mathtt{y} & \mathtt{z} & \pi \\ \hline \mathtt{X} & \mathtt{Y} & \mathtt{X} & \color{red}{\mathtt{X} \geq \mathtt{Y}} \end{array} < 0, 1, 2 >$$

Execution continues along feasible paths. In this case, the path condition $\pi$ is satisfiable:

```
void foo ( int x , int y) {
  int z := x
  if (z < y)
    z := z *2
```

Symbolic state

$$\begin{array}{cccc} \mathtt{x} & \mathtt{y} & \mathtt{z} & \pi \\ \hline \mathtt{X} & \mathtt{Y} & \color{red}{2\mathtt{X}} & \mathtt{X} < \mathtt{Y} \end{array} < 0, 1, 2, 3 >$$

Another if condition:

```
void foo ( int x , int y) {
```

```
2   int z := x
3   if (z < y)
4     z := z *2
5   if (x < y && z >= y )
```

- If condition is true

Symbolic state

$$\begin{array}{cccc} \mathtt{x} & \mathtt{y} & \mathtt{z} & \pi \\ \hline \mathtt{X} & \mathtt{Y} & \mathtt{2X} & \mathtt{X} < \mathtt{Y} \end{array} \; < 0, 1, 2, 3, 4 >$$

$$\mathtt{X} < \mathtt{Y} \wedge \mathtt{2X} \geq \mathtt{Y}$$

- If condition is false

Symbolic state

$$\begin{array}{cccc} \mathtt{x} & \mathtt{y} & \mathtt{z} & \pi \\ \hline \mathtt{X} & \mathtt{Y} & \mathtt{2X} & \mathtt{X} < \mathtt{Y} \end{array} \; < 0, 1, 2, 3, 4 >$$

$$\mathtt{X} \geq \mathtt{Y} \vee \mathtt{2X} < \mathtt{Y}$$

Possible outcomes of symbolic execution:

- Satisfiable exit ($\pi$ is satisfiable): every satisfying assignment to variables in $\pi$ is an input that satisfies the given property in a **concrete execution**.

Symbolic state

$$\begin{array}{cccc} \mathtt{x} & \mathtt{y} & \mathtt{z} & \pi \\ \hline \mathtt{X} & \mathtt{Y} & \mathtt{2X} & \mathtt{X} < \mathtt{Y} \end{array} \; < 0, 1, 2, 3, 4, 5 >$$

$$\mathtt{X} < \mathtt{Y} \wedge \mathtt{2X} \geq \mathtt{Y}$$

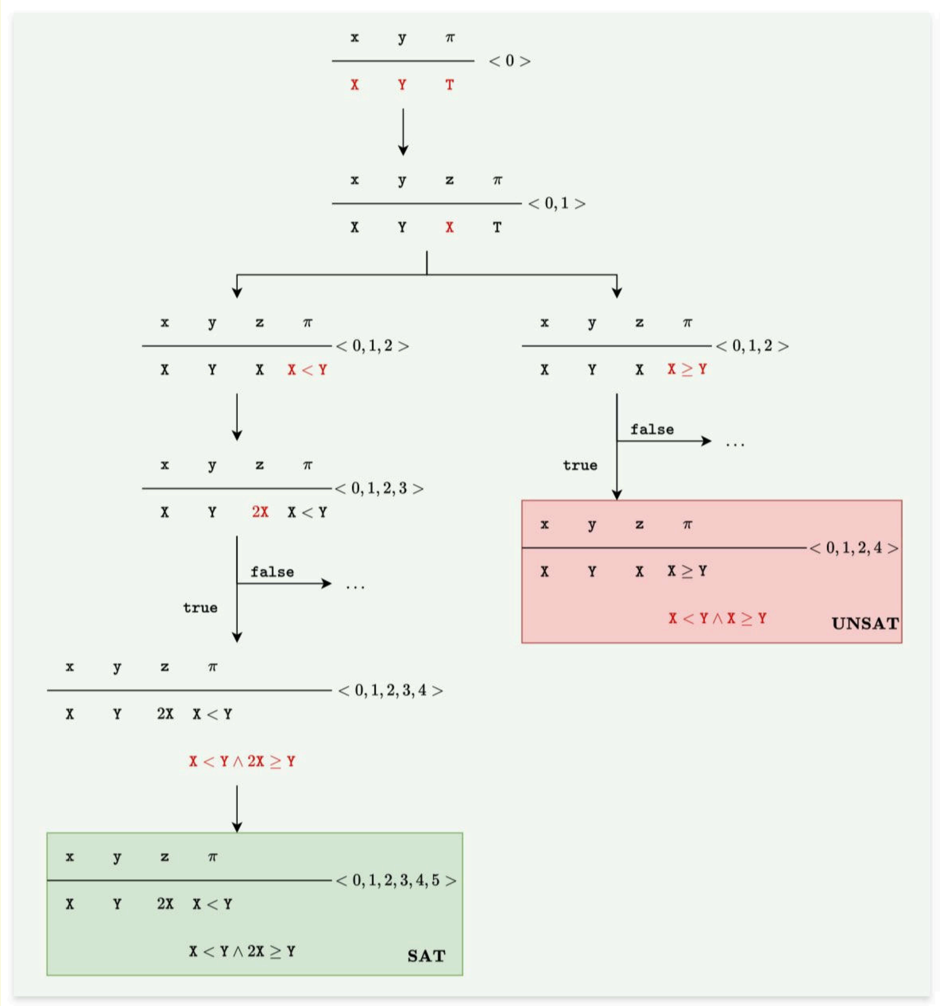
- Unsatisfiable exit ($\pi$ is unsatisfiable): the given property cannot be satisfied by any **concrete execution**.

Symbolic state

$$\begin{array}{cccc} \mathtt{x} & \mathtt{y} & \mathtt{z} & \pi \\ \hline \mathtt{X} & \mathtt{Y} & \mathtt{2X} & \mathtt{X} < \mathtt{Y} \end{array} \; < 0, 1, 2, 4 >$$

$$\mathtt{X} < \mathtt{Y} \wedge \mathtt{X} \geq \mathtt{Y}$$

Finally, we can draw the **Execution Tree**. The execution paths can be collected in an execution tree, where end states are marked as `SAT` or `UNSAT`.




### 6.4.2 Limitations

- The **path conditions may be too complex for constraint solvers**. Because solvers are very good at checking linear constraints, but it is harder for them to reason about non-linear arithmetic, bit-wise operations, string manipulation, etc.
- It is **impossible** or **difficult to use when the number of paths to be explored is infinite** or **huge**. For example, unbounded loops give rise to infinite sets of paths. Although the set of paths is finite, checking all loops is expensive and impractical.
- Finally, there may be **external code**. Then the sources are not available, such as a precompiled library, or the behavior is unknown to the solver.

# 7 Testing

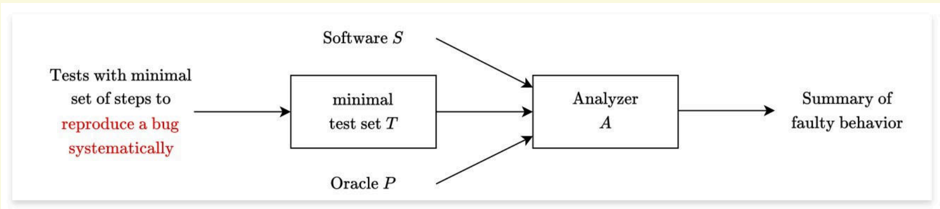
**Testing** (dynamic analysis) is an approach to verification. The **main goal of testing is to make programs fail**. Other common goals are:

- Exercise different parts of a program to increase coverage;
- Make sure the interaction between components works (integration testing);
- Support **fault localization** and **error removal** (debugging);
- Ensure that **bugs introduced in the past do not happen again** (regression testing).

Program testing can be used to show the presence of bugs, but never to show their absence!

## 7.1 Debugging

**Debugging** is a systematic approach to **fault localization and error removal**. The output is often used to support debugging.




## 7.2 Test case

> **Def** **Test case** **definition 7.2.1**
>
> A **Test Case** is a set of inputs, **execution conditions**, and a **pass/fail criterion**.

Running a test case typically involves setup, execution and teardown.

- Setup. Bring the program to an initial state that fulfils the execution conditions.
- Execution. Run the program on the actual inputs.
- Teardown. Record the output, the final state, and any failure determined based on the pass/fail criterion.

A test set, or test suite, can include multiple test cases. Finally, a **Test Case Specification** is a requirement to be satisfied by one or more test cases.

## 7.3 Unit Testing

When discussing test cases, it's necessary to introduce **Unit Testing**. This is conducted by developers and aims to test small pieces (units) of code in isolation. However, when we test in isolation, there should be a problem: the units may depend on other units. Then, we need to simulate missing units.

The **Integration Testing** (integration of the unit tests) aims to exercise the interaction between **interfaces and components**. The faults discovered by integration testing are multiple; some examples:

- Inconsistent interpretation of parameters (e.g. mixed units meters or yards)
- Violations of assumptions about domains (e.g. buffer overflow)
- Side effects on parameters or resources (e.g. conflict on temporary file)
- Nonfunctional properties (e.g. unanticipated performance issues)
- Concurrency-specific problems

Typically, the integration test is defined by the Design Document. In the Design Document, we can find two types of plans:

- Build Plan that establishes the order of the implementation;
- A Test Plan that defines how to carry out integration testing is needed.

## 7.4 Integration testing: strategies

### 7.4.1 Big Bang

Test only after integrating all modules together (not even a real strategy).

**Pros**: Does not require stubs, requires less drivers/oracles

**Cons**:

- Minimum observability, fault localization/diagnosability, efficacy, feedback
- High cost of repair

### 7.4.2 Iterative and incremental strategies

The main action is run after components are released (not just at the end). The strategy can be done in three different ways:

#### Hierarchical

Based on the hierarchical structure of the system. It can be done **top-down** or **bottom-up**.

**Top-down strategy**. Work from the top level (in terms of "use" or "include" relationship) down to the bottom level. As modules are completed (according to the building plan), more functionality is testable. We also need to replace some stubs, and we need other stubs for lower levels. **When all modules are incorporated, the whole functionality can be tested.** **Pros**: The drivers use the top level interfaces (e.g. REST APIs). **Cons**: This strategy requires stubs of used modules at each step of the process.

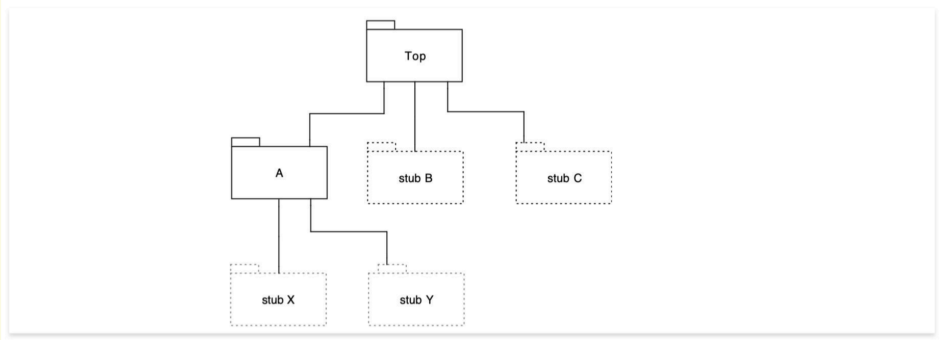



Figure 7.84: Example of top-down strategy

**Bottom-up strategy**. Starting from the leaves of the "uses" hierarchy. **Pros**: An advantage is that it doesn't require stubs. **Cons**: Typically requires more drivers (one for each module, as in unit testing). Another thing to consider is that it may create several work-ing subsystems, and each working subsystem will eventually be integrated into the final one.

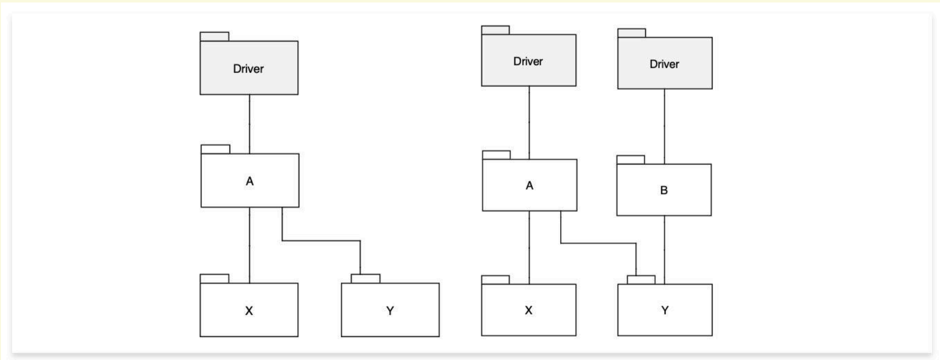



Figure 7.85: Example of bottom-up strategy

**Threads**. A thread is a part of several modules that together provide a user-visible programme function. By using the thread strategy we can have some advantages.

**Pros**:

- We can maximize the progress visible to the user (or other stakeholders);
- Reduce drivers and stubs;
- An integration plan is usually more complex.

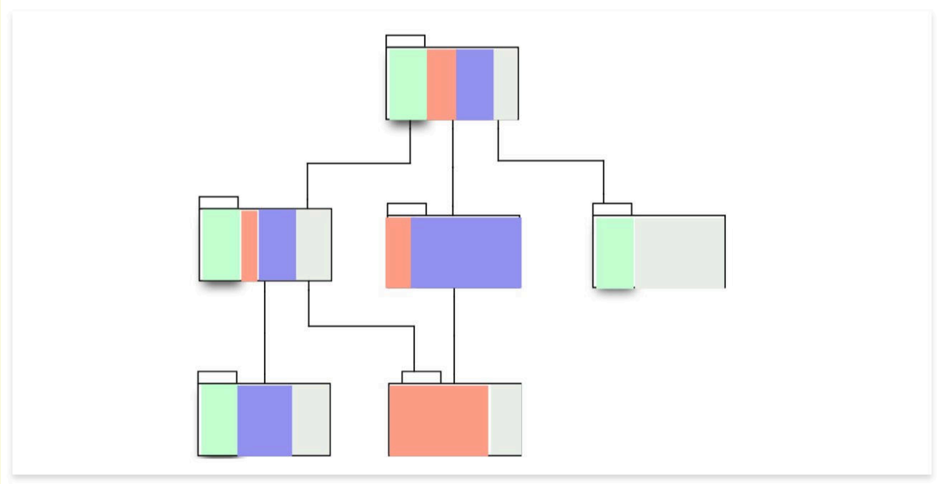


Figure 7.86: Example of thread strategy

**Critical**. The critical modules strategy starts with the highest risk modules. Risk assessment is a necessary first step. **The key point of this strategy is the risk-oriented process**. Integration and testing as a risk mitigation activity, designed to deliver any bad news as early as possible.

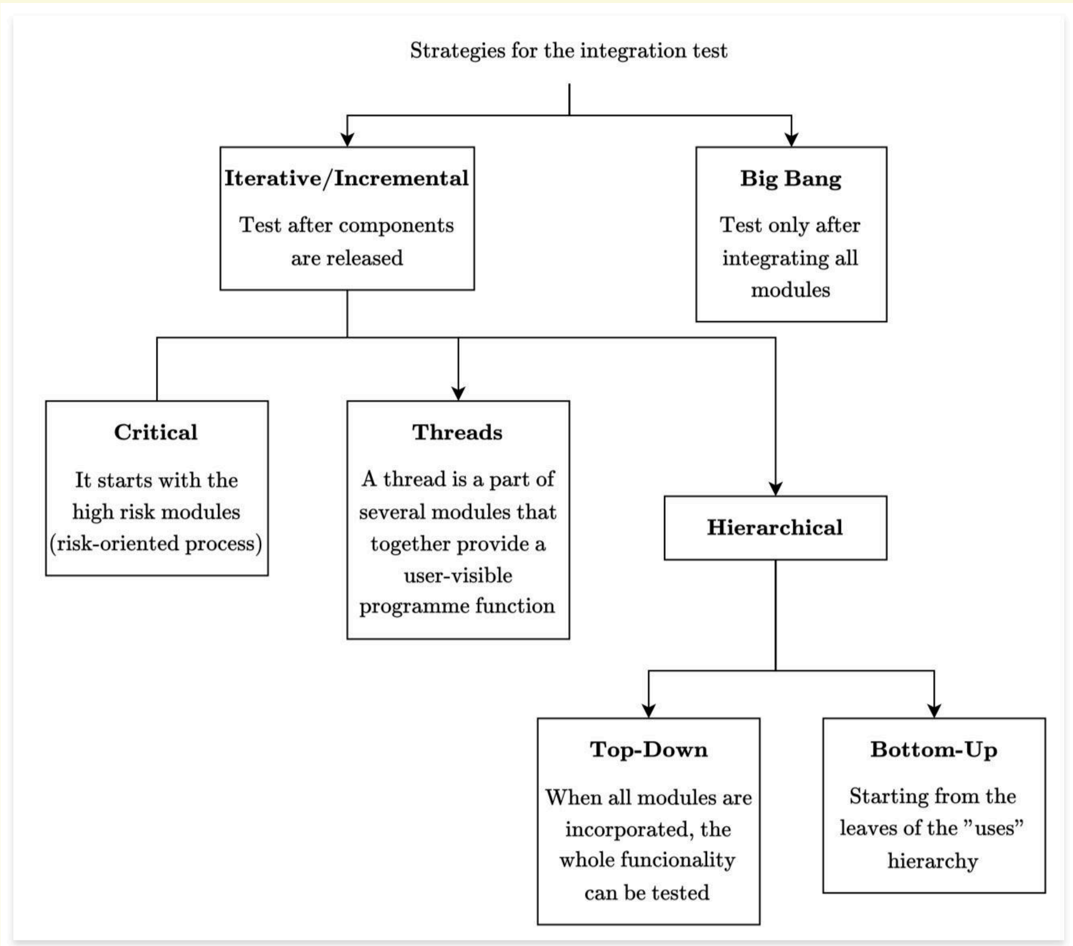



Figure 7.87: Summary of integration test strategies

Given the three strategies above, which one should we choose? Well, the structural strategies (bottom-up or top-down) are simpler, but thread and critical modules provide better external visibility of progress (especially in complex systems).

So the **best choice** should be a **combination of different strategies**:

- Use **top-down/bottom-up** for relatively small components and subsystems;
- **Combinations of thread and critical module** integration testing for larger subsystems.

## 7.5 E2E Testing

> **Def End-to-end (E2E)** **definition 7.5.1**
>
> **End-to-end (E2E)** testing is a software testing methodology to test a functional and data application flow consisting of several sub-systems working together from start to end.

At times, these systems are developed in different technologies by different teams or organizations. Finally, they come together to form a functional business application. Hence, testing a single system would not suffice. Therefore, **end-to-end testing** verifies the application from start to end putting all its components together.

The following is a list of common types of tests that use the E2E system:

- **Functional testing**: Check whether the software meets the functional requirements.
- **Performance testing**:
  - Detect **bottlenecks** affecting response time, utilization, throughput
  - Detect **inefficient algorithms**
  - Detect **hardware/network issues**
  - Identify **optimization possibilities**
- **Load Testing**:
  - **Expose bugs** such as memory leaks, mismanagement of memory,buffer overflows
  - Identify **upper limits of components**
  - **Compare alternative architectural options**
- **Stress Testing**: Make sure that the system recovers gracefully after failure.

## 7.6 Test case generation

### 7.6.1 Introduction

**Testing Workflow** is a **type of software testing that verifies that each software workflow accurately reflects the given business process**. A workflow is a series of tasks to produce a desired result, usually involving several stages or steps. For any business process, testing these sequential steps is defined as "workflow testing".

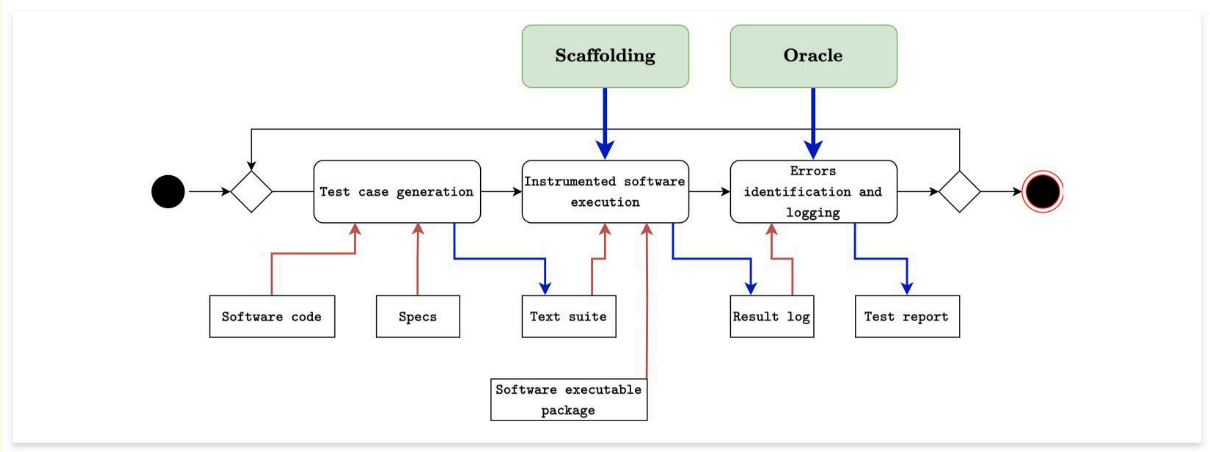



Figure 7.88: Testing workflow

Test cases can be generated in a **black-box** or **white-box manner**. The **White Box** is a generation based on **code features**. Meanwhile, the **Black Box** is a generation based on **specification features**.

Test case generation can be done manually (no need to explain) or automatically. Automatic generation can be done in several ways:

- **Combinatorial testing**. It enumerates all possible inputs according to some policy (e.g. smaller to larger).
- **Concolic Execution**. It's a pseudo-random generation of inputs guided by symbolic path properties.
- **Fuzz testing (fuzzing)**. It's a pseudo-random generation of inputs, including invalid, unexpected inputs.
- **Search-based** testing. It explores the space of valid inputs, looking for those that improve some metric (e.g. coverage, diversity, fault inducing capability).
- **Metamorphic testing**. Generates new test cases based on some metamorphic relationships and other previously defined test cases.

### 7.6.2 Concolic Execution

**Concolic Execution** (concrete-symbolic execution) is an automatic generation of test cases. It's a pseudo-random generation of inputs guided by symbolic path properties.

In other words, the concolic execution **performs symbolic execution** alongside concrete execution (concrete inputs). Under the hood, in concolic execution a **state** combines a symbolic part and a concrete part, used as needed to make progress in the exploration.

The steps are then as follows:

- Concrete $\rightarrow$ Symbolic, derive conditions to explore new paths.
- Symbolic $\rightarrow$ Concrete, simplifying conditions to generate concrete inputs.

Let's take an example to clarify the explanation.

See the code below:

```python
def m2 (x: int , y : int):
  z: int = bb (y) # black - box function
  if z == x:
    z = y + 10
    if x <= z:
      print(" Log message .")
```

Let's explore all the paths of the m2 method, starting with **a (random) concrete input** and at the same time building the **symbolic condition** of the explored path. Unfortunately, in some cases we will not be able to solve the symbolic execution. For example, the behavior of the first if-condition ($z == x$) is unknown in the code. For this reason, we execute it with the identified input cases: given $y = 7$, run bb(7) and return 14. With this arrangement, the condition can be solved.