Software Engineering for HPC - Notes

260236

May 2024

Preface

Every theory section in these notes has been taken from two sources:

- None

About:

GitHub repository

Contents

1 Introduction **5**
- 1.1 The importance of software engineering . . . 5
- 1.2 Software engineering: definition . . . 6
- 1.3 The software product and the process . . . 7
  - 1.3.1 ISO/IEC 25010 . . . 7
  - 1.3.2 Productivity . . . 7
  - 1.3.3 Timeliness . . . 8
- 1.4 Software Lifecycle . . . 9
  - 1.4.1 Waterfall model . . . 9

2 HPC Software, Relevant Qualities and Systems Engineering Methods **11**
- 2.1 High Performance Computing Software . . . 11
- 2.2 Relevant Qualities . . . 13
- 2.3 Systems Engineering Methods . . . 15

3 Requirement Engineering **16**
- 3.1 Definition . . . 16
- 3.2 Studying the interplay between the world and the machine . . . 17
  - 3.2.1 Completeness of Requirements . . . 19
- 3.3 Formulating and classifying requirements . . . 20
- 3.4 Eliciting requirements . . . 22

4 Software Design **27**
- 4.1 Software Architecture . . . 27
- 4.2 Architecture and multiple structures . . . 28
- 4.3 Software design descriptions and UML . . . 29
  - 4.3.1 Component Diagram (C&C structure) . . . 29
  - 4.3.2 Sequence Diagram (C&C structure) . . . 30
  - 4.3.3 Class Diagram (module structure) . . . 31
  - 4.3.4 Package Diagram (module structure) . . . 32
  - 4.3.5 Deployment Diagram (allocation structure) . . . 33
- 4.4 Design principles . . . 34

5 Architectural styles **37**
- 5.1 Definition . . . 37
- 5.2 Client-Server . . . 38
  - 5.2.1 Interface design . . . 38
  - 5.2.2 Error handling, multiple interfaces and interface evolution 39
  - 5.2.3 Handling multiple requests . . . 40
- 5.3 Three-Tier Architecture . . . 43
  - 5.3.1 N-tier architecture . . . 43
- 5.4 Microservice architectural style . . . 44
- 5.5 Event-Driven Architecture . . . 45
  - 5.5.1 Apache Kafka . . . 46
- 5.6 Data-Intensive applications . . . 49
  - 5.6.1 Batch approach: MapReduce . . . 50
  - 5.6.2 Stream approach: Apache Storm . . . 52

5.6.3 Combining batch and stream: Lambda Architecture . . . 54

6 Verification and Validation **55**
6.1 Terminology . . . 55
6.1.1 Study concurrent use of resources at architectural level . . 58
6.1.2 Quantitative impact of architectural decisions . . . 61
6.2 Analysis: Symbolic Execution . . . 69
6.3 Testing: terminology, types of testing activities . . . 74
6.3.1 E2E Testing . . . 79
6.4 Test case generation . . . 81
6.4.1 Introduction . . . 81
6.4.2 Concolic Execution . . . 82

Index **87**

1 Introduction

1.1 The importance of software engineering

Software engineering is so important because it is everywhere. Our society is now totally dependent on software-intensive systems. Think about it. The society could not function without software, for example:

- Transportation systems;
- Energy systems;
- Manufacturing systems.

For these reasons, **software failures cannot be tolerated**.

In the following list, we can see some famous software issues:

- **911 Outage on April 2014**. On 10th April 2014, Washington State had no 911 service for six hours. A software issue causes this event. The software dispatching the calls had a counter used to assign a unique identifier to each call. The counter went over the threshold defined by developers. All calls from that moment on were rejected.

  More info is here.

- **Ariane 5, 1996**. On 4th June 1996, forty seconds after take off, Ariane 5 broke up and exploded. The total cost for developing the launcher has been 8000 million dollars. The launcher contained a cluster of satellites for 500 million dollars. Again, the explosion was caused by software failure.

  More info is here: accident tech report and video.

1.2 Software engineering: definition

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

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

In each field, a software engineer needs to have some skills. In contrast to a *programmer* that has the following abilities:

- They develop a complete program;
- They work on known specifications;
- They work individually.

A **software engineer** has the following **skills**:

- **Identifies** *requirements* and develops *specifications*;
- **Designs** a component to be combined with other components, developed, maintained, and 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 **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.

1.3 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**.

1.3.1 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: ISO/IEC 25010

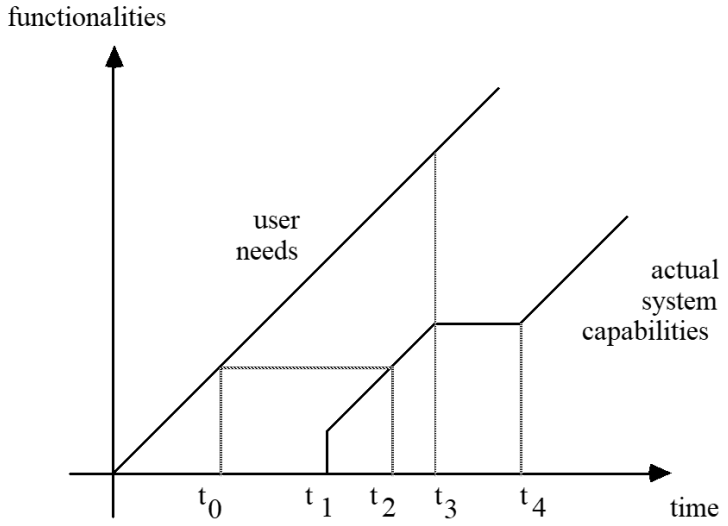
1.3.2 Productivity

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).

1.3.3 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 (and sometimes can be exponential!). 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 (ISO/IEC 25010).

1.4 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.

1.4.1 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. [1] 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 (e.g. buy or make), 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.
  - **Requirements 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 (continuous) 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*** maintenance: consists of **adapting software to changes in the environment** (the hardware or the operating system, business rules, government policies).
    - ***Perfective*** maintenance: mainly deals with accommodating **new or changed user requirements**.
    - ***Preventive*** maintenance: concerns activities aimed at **increasing the system's maintainability**.

⚠ Problems derived from correction and evolution

Note: the **distinction between correction and evolution can be unclear** because specifications often must be completed and clarified. This causes problems because specs are usually part of a developer and customer contract.

- **Early frozen specs** can be problematic because they are more likely to be wrong.
- Another problem is **software evolution** because **it is never anticipated or planned**. Since the software is easy to change, often, under emergency, changes are applied directly to code, and consequently, the state of project documents is inconsistent.

✔ Solutions - Best practices

Some good engineering practices exist to solve the evolution problem: **first, modify the design, then change implementation and apply changes consistently in all documents**. Also, the **software must be designed to accommodate changes cost-effectively**. This is one of the *main goals* of software engineering.

✔ Flexible processes

We can make the **waterfall model more flexible**. In this case, the **main goal is to adapt to changes (especially in requirements and specs)**. The idea is that the stages are **not necessarily sequential**, and **processes** become **iterative and incremental**.

2 HPC Software, Relevant Qualities and Systems Engineering Methods

2.1 High Performance Computing Software

There's no single definition of HPC, but it can be explained in a number of ways:

Definition 1

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 aand 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).

As **Parallel and Distributed Computing (PDC)** exist, it is necessary to explain the difference. The main characteristics of PDC are:

- **Concurrency**: it is a property of software. **A piece of software is also concurrent if it can have more than one active execution context**.
- **Parallelism**: it is a property of software. **The execution of different tasks/pieces of software at the same time**.
- **Distribution**. **The execution of different tasks/pieces of software on physically distinct computing nodes connected through a network, lack of a global clock**.

PDCs are **multi-core machines**, whereas **HPCs** are **quantum computers**. However, **both share parallel machines**, **HPC clusters** and **cloud infrastructures**.

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**: the **software is correct if it satisfies the specifications**, but be careful! Sometimes, modelling reality into a model (using the specifications) isn't the bigger problem. Instead, it is difficult or impossible to show actual correctness concerning reality. For **example**, imagine you are building a simulator of a planet lander before you have ever visited it.

    How can you fix this issue? We can **check that the software output fulfils the important desired properties** and **identify and apply a measure of accuracy**.
  - **Performance**: it is the **efficient use of resources**. Again, be careful! Is it a good idea? Is performance improvement always a good idea? Because it is **not necessarily if**:
    - It makes software **more difficult to read** and **maintain**
    - It **reduces the portability** of software
  - **Portability**.
  - **Maintainability**. A system can have this feature if it follows **three principles**:
    1. **Operability**: **make it easy for the operations team to run the system and keep it running**. There are a number of things that need to be done to achieve this:
       - Provide visibility into the runtime behavior and internals of the system, with good monitoring.
       - Provide good support for automation and integration with standard tools.
       - Avoidance of dependencies on individual machines (allowing machines to be taken down for maintenance while the system as a whole continues to run uninterrupted).
       - Provide good documentation and an easy to understand operational model ("If I do X, Y will happen").
       - Provide good default behavior, but also give administrators the freedom to override defaults when necessary.
       - Self-healing when appropriate, but also giving administrators manual control over system state when needed.
    2. **Simplicity**: **make it easy for other software engineers to understand the system**. This is necessary because complex systems take more time to understand and increase the cost of maintenance. There are several techniques for doing this:
       - Reducing *accidental complexity*.
       - Using abstractions, such as organising the architecture into well-defined components that hide the internal complexity behind a clear and easy-to-use interface; or reusing known solutions.

3. **Evolvability**: make it easy for engineers to change the system as new requirements emerge. There are a number of things that need to be done to achieve this:
   * Organize your development process to cope with evolution.
   * Keep track of how requirements are mapped to your software structure.
   * Update documentation.
   * Continue to ensure simplicity and operability.

- For **Data-intensive applications**:
  - **Reliability**: can be mathematically defined as **probability of absence of failures for a certain period**. The typical expectations are:
    * The application **performs the expected function**
    * It can **tolerate mistakes by users**
    * It **prevents unauthorized access** and **abuse**
  - **Scalability**: the **system ability to cope with increased load**. The load unit depends on the product: for web apps can be represented with the number of requests per second; for databases can be the number of read and write operation (or their ratio).
  - **Maintainability**. Same as above.

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

- A **defect** is an **imperfection or deficiency in a work product** where that work product does not meet its requirements or specifications and needs to be either repaired or replaced.
- A **defect encountered during software execution** is a **fault** (a fault is a subtype of defect, and can be of two types, see below).
- A **system failure** can be:
  - Termination of the ability of a product to perform a required function or its inability to perform within previously specified limits.
  - An event in which a system or system component does not perform a required function within specified limits.

There are some exceptions where systems are **fault-tolerant** or **resilient**. These are systems that can **cope with faults and prevent faults from occurring**. An **advantage of fault-tolerance** is that **reliability is increased**.

The **fault** can be of **two types**:

- **Hardware Faults**.

  ⚠ Description of the problem

  It is a defect encountered during hardware execution. In a large datacenter these can happen on a daily basis. Different pieces of hardware usually fail independently from each other.

  ✔ Possible solutions

  The possible solutions are two: **hardware redundancy** and **software fault-tolerance techniques**.

- **Software Faults**.

  ⚠ Description of the problem

  They result from **software development errors**. Can stay dormant for a long time and appear suddenly. They can **determine failures in multiple components** at the same time.

  ✔ Possible solutions

  There is no single solution! It is a combination of strategies. So use defensive programming, by testing before release and during operation:

  – Reboot the system frequently (rejuvenation)
  – Continuous monitoring and alerting in case of possible symptoms
  – Deliberately introduce failures to train the fault tolerance machinery (chaos engineering)

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.
- Source code management.
- Documentation, standards, support to maintainability.
- Support to scalability.
- Attention to operability and automation.

3 Requirement Engineering

3.1 Definition

Before the definition, we give a possible scenario to understand what requirement engineering is.

The municipality of Milan says the following: "*The time it takes to make decisions on building permits for residential buildings in the city is too long. We want to develop software that will help us reduce this time*". So where do we start? How do we identify the most important aspects? How do we make sure that we have understood what our customers want from us?

Definition 1

Software measure engineering (**Requirement Engineering**) is the process of discovering the purpose for which the software is intended by identifying stakeholders and their needs, and documenting these in a form suitable for analysis, communication and subsequent implementation.

The questions derived from requirements engineering are:

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

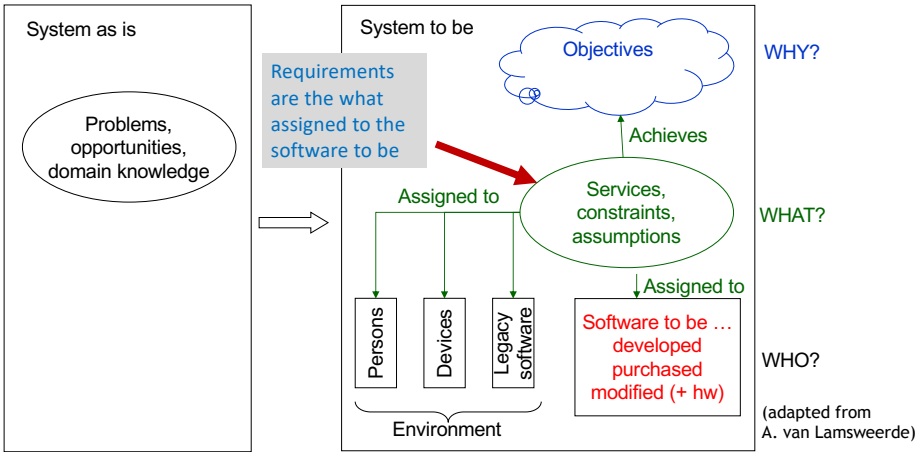


Figure 2: Analyzing the system as is and the system to be.

3.2 Studying the interplay between the world and the machine

Example 1: ambulance dispatching system

For every urgent call reporting an incident, an ambulance should arrive at the incident location within 14 minutes. 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.

Given the previous problem, are you able to extract requirements from this description? Some possible questions might be:

- Should the software system drive the ambulance?
- Who or what will "correctly encode" details about incidents?
- Do terminals already exist or not?

And more in general:

- What are the boundaries of the system? What is inside/outside? What is in-between?
- How do we think about these aspects in a systematic way?

This example is necessary to understand the **phenomena of world and machine**. 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

Requirements models are models of the world!

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

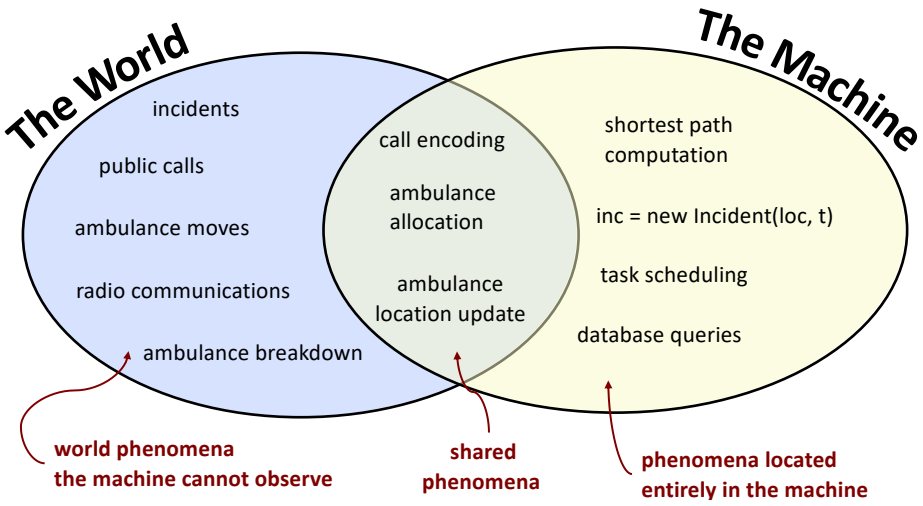


Figure 3: The world and the machine sets, with reference to example on page 17.

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.

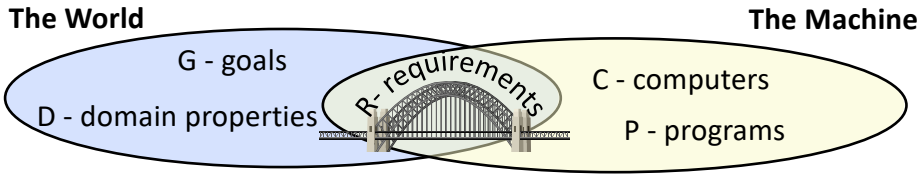


Figure 4: Goals, domain properties, requirements, computers and programs.

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** (or assumptions) are **descriptive assertions assumed to hold in the world**
- **Requirements** are **prescriptive assertions formulated in terms of shared phenomena**

Using the **example** on the page 17, 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.*

3.2.1 Completeness of Requirements

Given the set of requirements **R**, goals **G** and domain assumptions **D**.

Definition 2

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

- **R** ensures satisfaction of **G** in the context of domain assumptions **D**

$$\mathbf{R} \text{ and } \mathbf{D} \models \mathbf{G}$$

We can make an analogy with program correctness. A program P running on a particular computer C is correct if it satisfies the requirement R: P and C $\models$ R.

- **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**.

  For ***example***: "*the word processor shall allow users to search for strings in the text*"; "*the system shall allow users to reserve taxis*".

- **Non-functional requirements (NFRs)**: further characterization of user-visible aspects of the system not directly related to functions.

  For ***example***: "*the response time must be less than 1 second*"; "*the server must be available 24 hours a day*".

- **Constraints requirements**: imposed by the customer or the environment in which the system operates.

  For ***example***: "*the implementation language must be Java*"; "*the credit card payment system must be able to be dynamically invoked by other systems relying on it*".

We make some observations about non-functional requirements. NFRs predicate on **external** non-functional qualities, and these qualities must be **measurable by metrics**. 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).

Example 2: are these requirements?

1. "*The user should insert correct information in the enrolment form*".

This is not a requirement! How can the software prevent a user from entering incorrect information? Specifically, is a domain assumption!

2. "*The system should check whether fiscal code are well formed*".

Yes, the software can do this! So it is a requirement.

Example 3: types of requirements

Example of **functional requirements**:

- "*The system shall allow users to reserve taxis*".
- "*The system should never allowe non-registered users to see the list of other users willing to share a taxi*".

- "*The system should guarantee that the reserved taxi picks the user up*".

But attention! There is **unfeasible** (from the perspective of the software to be) **functional requirements**:

- "*The system should guarantee that the reserved taxi picks the user up*".

This is because the software cannot guarantee this feature!
Example of **non-functional requirements**:

- "*The system has to provide a feedback in 5 seconds*".
- "*The system should be available 24/7*".

Example of **technical requirements**:

- "*The system should be implemented in Java*".
- "*The search for the available taxi should be implemented in class* `Controller`".

Example 4: bad requirements

1. "*The system shall process all mouse clicks very fast to ensure users do not have to wait*".

The problem here is that it **cannot be verified (tested)**, because what does "fast" mean? Do we have a metric? Can you quantify it?

2. "*The user must have Adobe Acrobat installed*".

The problem here is that it **cannot be achieved by the software system itself**. It is not something that the system has to do. But it could be expressed as a domain assumption, so it is not a functional requirement for our software.

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 **scenario**s.

Definition 3

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

Example 5: warehouse on fire

Bob driving down main street in his patrol car notices smoke coming out of a warehouse. His partner, Alice, reports the emergency from her car.

Alice enters the address of the building, a brief description of its location (i.e. north west corner), and an emergency level. In addition to a fire unit, she requests several paramedic units on the scene given that area appears to be relatively busy. She confirms her input and waits for an acknowledgment.

John, the Dispatcher, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and two paramedic units to the incident site and sends their estimated time of arrival (ETA) to Alice.

Alice received the acknowledgment and the ETA.

There are heuristics for finding scenarios, such as asking the customer a few questions:

- Which user groups are supported by the system to perform their work?
- What are the primary tasks that the system needs to perform?
- What data will the actor create, store, change, remove or add in the system?
- What external changes does the system need to know about?
- What changes or events will the actor of the system need to be informed about?

However, it's very important not **to rely on questionnaires alone**! **Insist on task observation** (if possible), ask to **speak to the end user**, not just the software contractor, and expect resistance, but try to overcome it.

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.

Definition 4: Use Cases Schema

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

The following **suggestions** are useful in defining an appropriate use case:

- Use cases named with verbs that indicate what 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)

First of all, we present an example of a **bad use case**.

Example 6: bad use case

Example of a bad use case referring to the ambulance dispatching example on page 17:

- Use case name: Accident
- Participating 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 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.
- There are two main errors in the *Flow of Events* section: the first is in the sentence "*An ambulance is dispatched*". But **who sends it?** The second is in the third sentence, because **who notifies the Dispatcher?**

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

Example 7: use case `ReportEmergency` with reference to the example on page 22

There are two **actors** involved:

- Field Officer (Bob and Alice in the Scenario)
- Dispatcher (John in the Scenario)

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. The FieldOf-

ficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form.

- 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 possible **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 FieldOfficer 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.

Example 8: use case `AllocateResources` with reference to the example on page 22

- Use case name: AllocateResources
- Participating Actors:
  - Dispatcher (John in the Scenario. The Dispatcher allocates a resources to an Emergency if the resource is available; of course, he also updates and removes Emergency Incidents, Actions, and Requests in the system)
  - Resource Allocator (the Resource Allocator is responsible for allocating resources in case they are scarce)
  - Resources (the Resources that are allocated to the Emergency)
- Entry Condition:
  - An Incident has been opened
- Flow of Events:
  - The Dispatcher selects the types and number of Resources that are needed for the incident.

  - Friend replies with a list of Resources that fulfill the Dispatcher's request.
  - The Dispatcher selects the Resources from the list and allocates them for the incident.
  - Friend automatically notifies the Resources.
  - The Resources send a confirmation.

- Exit Condition:
  - The use case terminates when the resource is committed.
  - The selected Resource is now unavailable to any other Emergences or Resource Requests.

- Exceptions:
  - If the list of Resources provided by Friend is insufficient to fulfill the needs of the emergency, the Dispatcher informs the Resource Allocator.
  - The Resource Allocator analyzes the situation and selects new Resources by decommitting them from their previous work.
  - Friend automatically notifies the Resources and the Dispatcher.
  - The Resources send a confirmation.

4 Software Design

4.1 Software Architecture

Definition 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.

The software architecture is a tool for thinking about systems, and it's made up of a set of structures.

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.

4.2 Architecture and multiple structures

There is a set of structures relevant to the software:

- **Component-and-connector (C&C)** structures. Describe how the system is structured as a **set of elements** that have **runtime behavior** (called components) and interactions (called 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?*

  Also, **allow us to study runtime properties** such as availability and performance.

- **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.

  The ***purpose*** of these structures is to **allow us to answer questions** such as:
  - *What is the primary functional responsibility assigned to each module?*
  - *What other software elements is a module allowed to use?*
  - *What other software does it actually use and depend on?*
  - *What modules are related to other modules by generalization or specialization (i.e. inheritance) relationships?*

  Can also be used to answer questions about the **impact on the system when the responsibilities assigned to each module change**.

- **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

In the following pages we present some diagrams that are fundamental to creating appropriate documentation. The following diagrams show how to create some UML diagrams.

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.

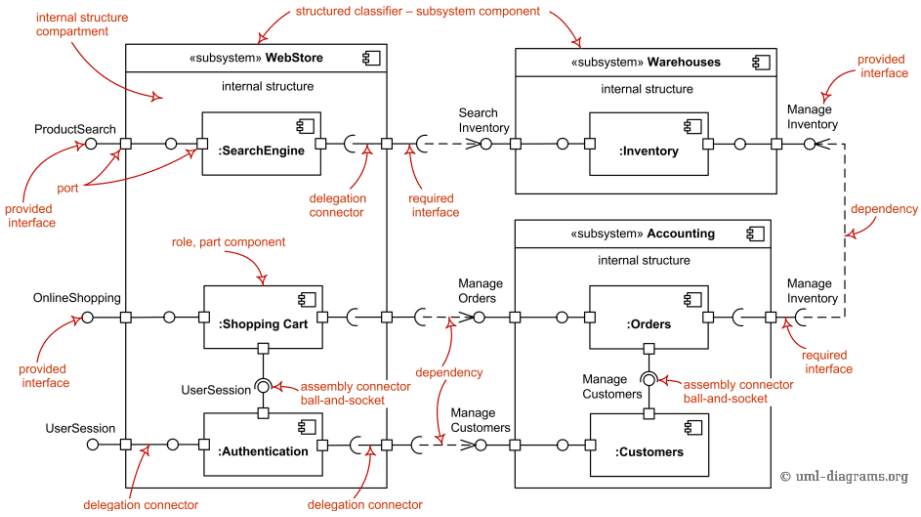


Figure 5: Component Diagram.

To view the component diagram in high resolution, scan (or click) the QR code below.

A complete guide can be found on the following page: What is Component Diagram?

4.3.2 Sequence Diagram (C&C structure)

Sequence Diagram show **elements as they interact over time** and they are organized according to object (horizontally) and time (vertically).

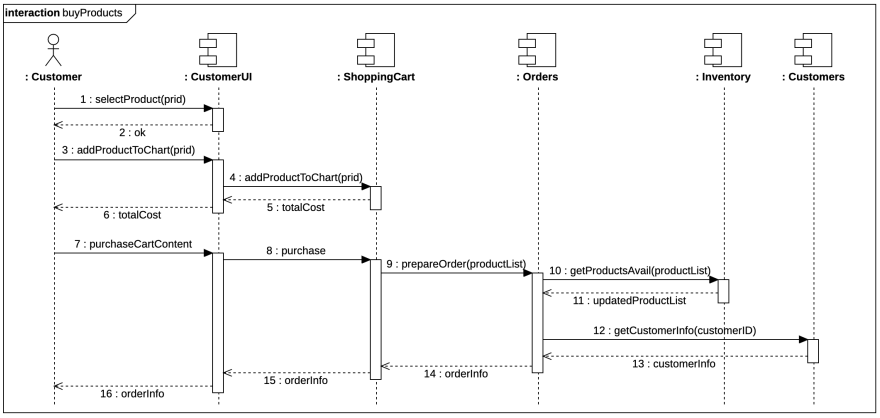


Figure 6: Sequence Diagram.

To view the sequence diagram in high resolution, scan (or click) the QR code below.

A complete guide can be found on the following page: What is Sequence 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.

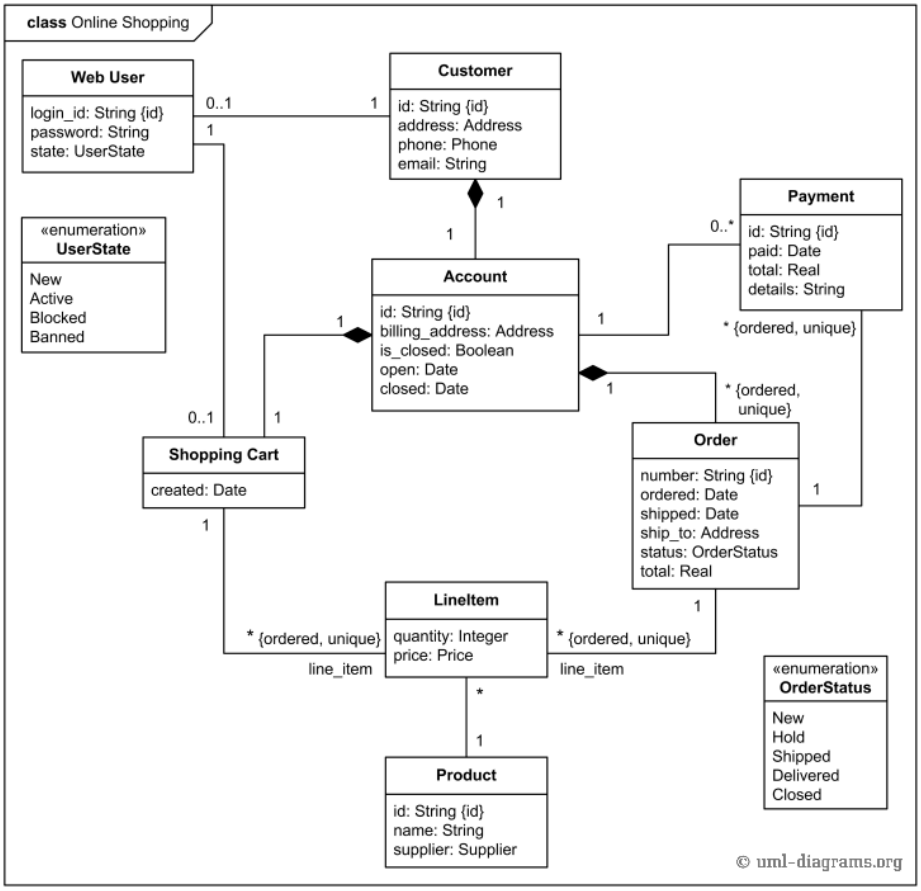


Figure 7: Class Diagram.

To view the sequence diagram in high resolution, scan (or click) the QR code below.

A complete guide can be found on the following page: What is 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.

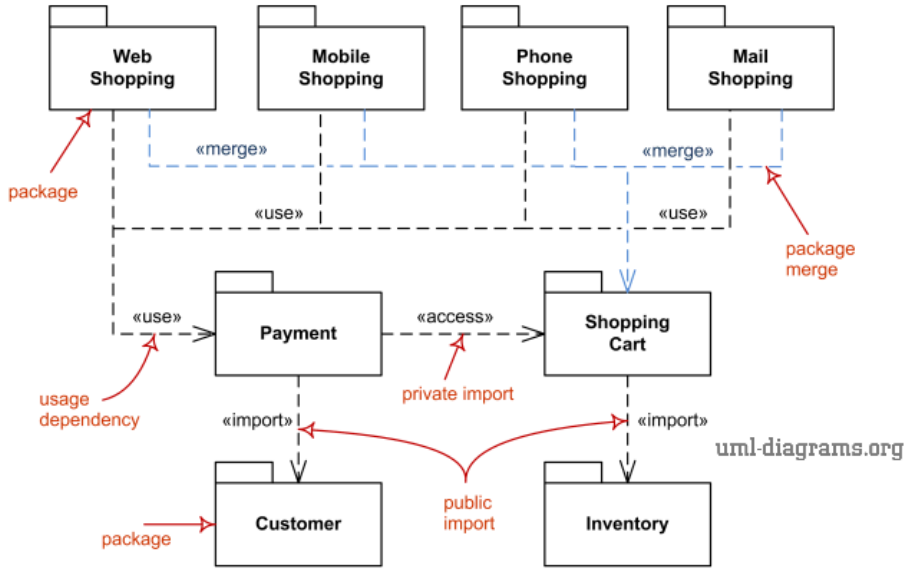


Figure 8: Package Diagram.

To view the sequence diagram in high resolution, scan (or click) the QR code below.

A complete guide can be found on the following page: What is Package Diagram?

4.3.5 Deployment Diagram (allocation structure)

A **Deployment Diagram** is a diagram that shows the **configuration of run time 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).

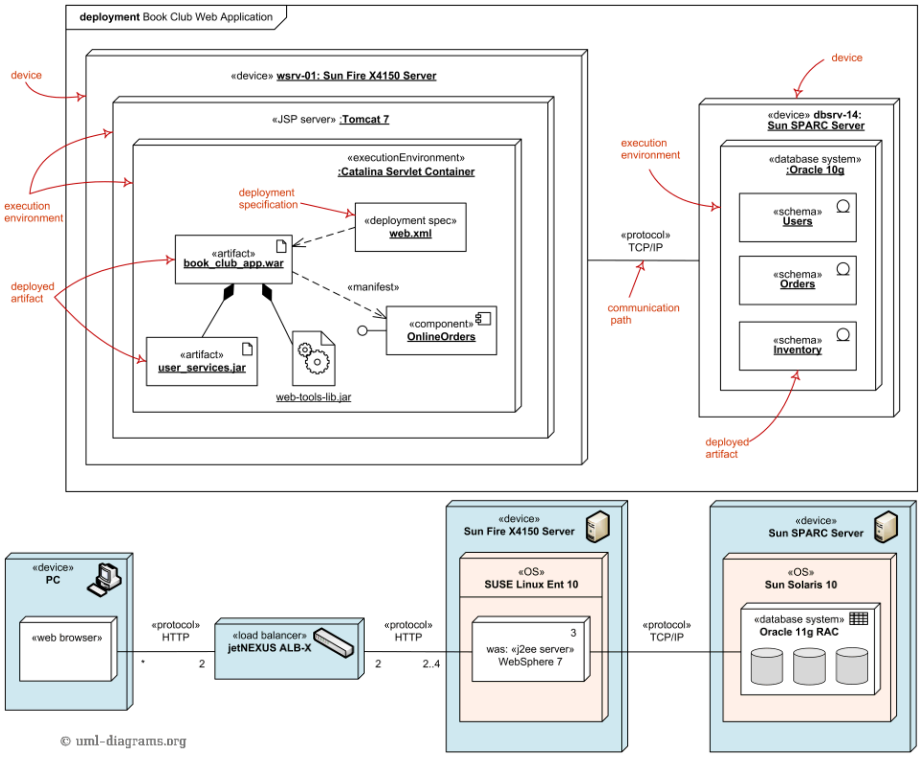


Figure 9: Deployment Diagram.

To view the sequence diagram in high resolution, scan (or click) the QR code below.

A complete guide can be found on the following page: What is Deployment Diagram?

4.4 Design principles

The following is a list of important design principles in software engineering.

Definition 2

1. **Divide et impera** (also called *divide and conquer*)
2. **Keep the level of abstraction as high as possible**
3. **Increase cohesion where possible**
4. **Reduce coupling where possible**
5. **Design for reusability**
6. **Reuse existing designs and code**
7. **Design for flexibility**
8. **Anticipate obsolescence**
9. **Design for portability**
10. **Design for testability**
11. **Design defensively**

1. Divide et impera (*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.

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.

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!).

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

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:

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

Other types can be viewed here.

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.

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.

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:

- Reduce coupling and increase cohesion;
- Create abstractions;
- Use reusable code and make code reusable;
- Do not hard-code anything.

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.

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).

10. Design for testability

Take steps to **make testing easier**. Design a program to automatically test the software:

- 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.

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 styles

5.1 Definition

Definition 1

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.

These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints - say, having to do with execution semantics - might also be part of the style definition.

5.2 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

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).

⚠ Technical issues

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**.

5.2.1 Interface design

An **interface design** 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**.

5.2.2 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).

There are three possible **solutions**: **raising an exception**; **return an error code** (common); **log the problem**. There's no single solution, but we can choose several (e.g. error code and log the problem).

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.

5.2.3 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**.

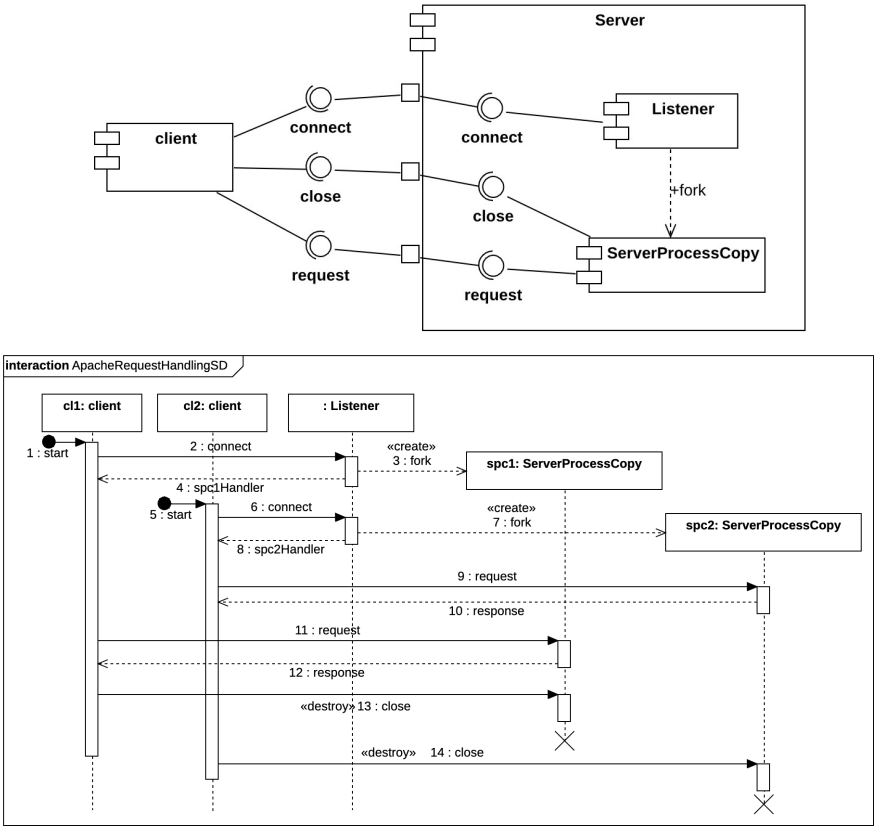


Figure 10: Forking diagrams.

✔ 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**.

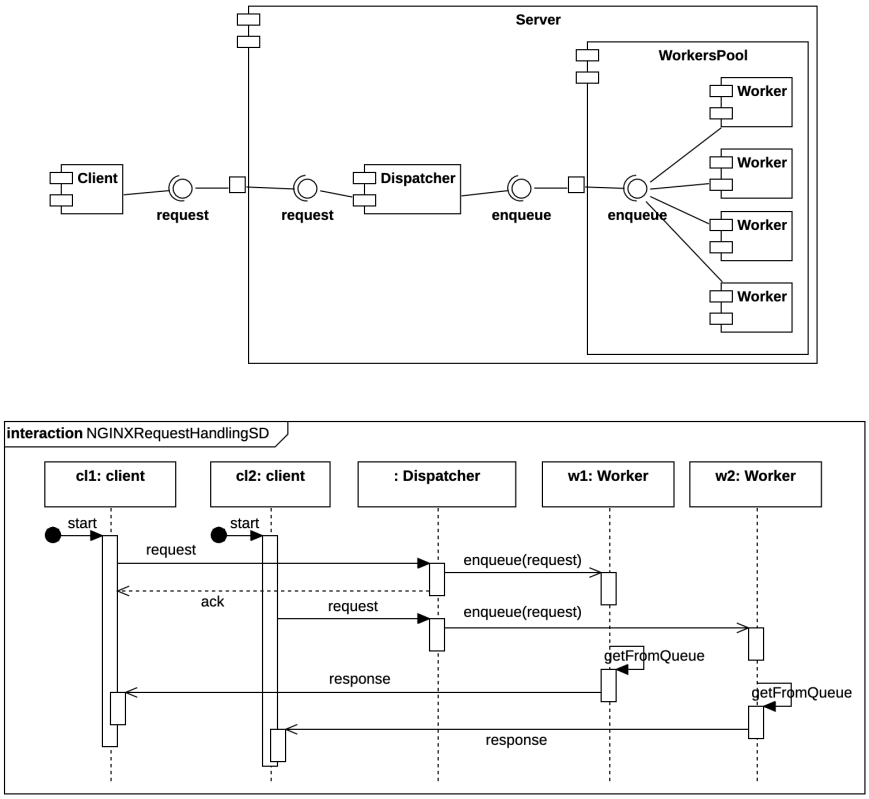


Figure 11: Worker pooling diagrams.

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**.

5.3 Three-Tier Architecture

The following is a summary of the IBM guide.

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.

✔ 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.

5.3.1 N-tier architecture

N-tier architecture (also called or multitier architecture) refers to any application architecture with **more than one tier**. But applications with more than three layers are rare because extra layers offer **few benefits** and can make the **application slower**, **harder to manage** and **more expensive to run**. As a result, n-tier architecture and multitier architecture are usually synonyms for three-tier architecture.

5.4 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.

✔ Benefits

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.

5.5 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).

✔ 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).

⚠ 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.

5.5.1 Apache Kafka

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.

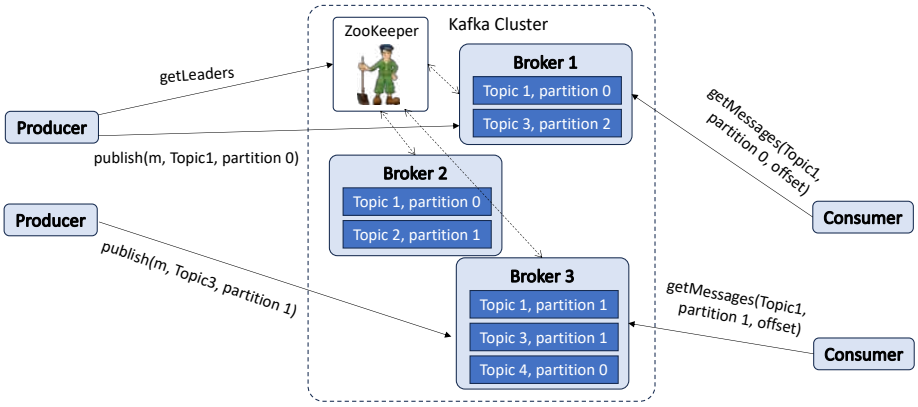


Figure 12: Kafka architecture (the ZooKeeper is a "health manager").

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.

Message delivery

Producer

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

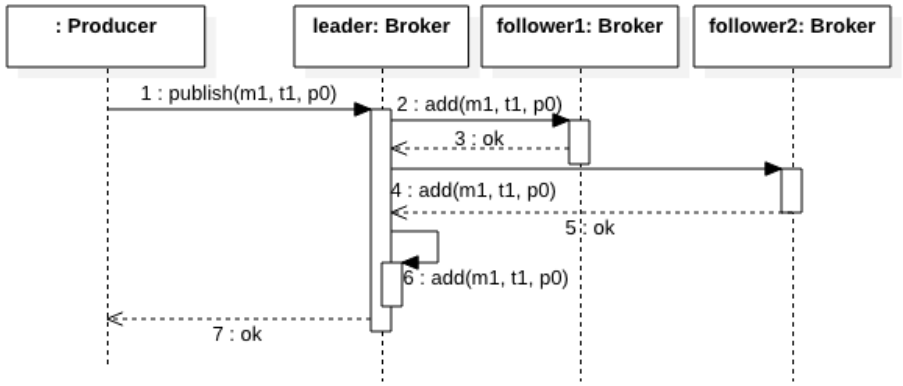


Figure 13: Sequence diagram Kafka producer.

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 for **exactly-once semantics**.

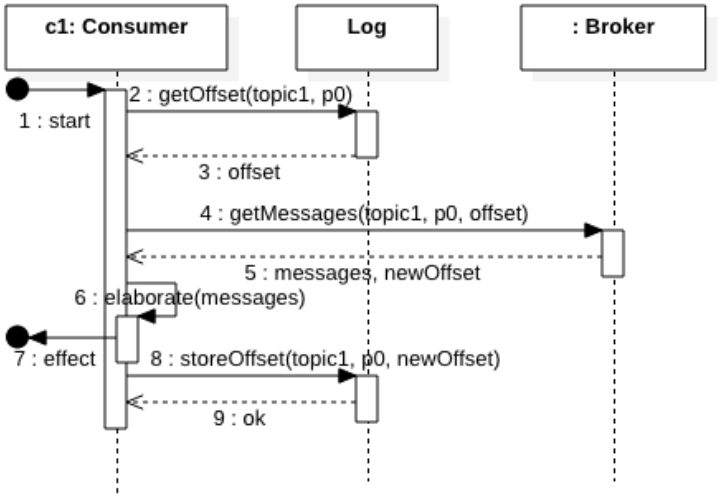


Figure 14: Sequence diagram Kafka consumer.

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.

5.6 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 simple.
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.

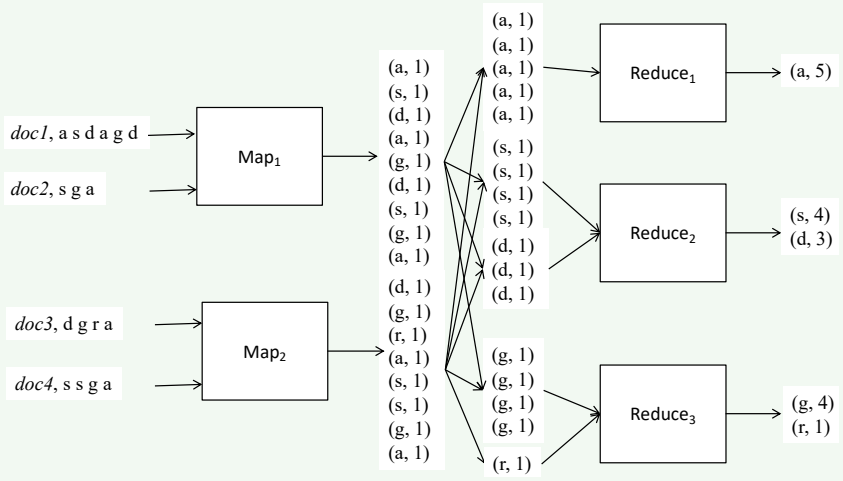
Table 1: Batch vs Stream processing.

5.6.1 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.

Example 1: an example of a batch approach using MapReduce



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)

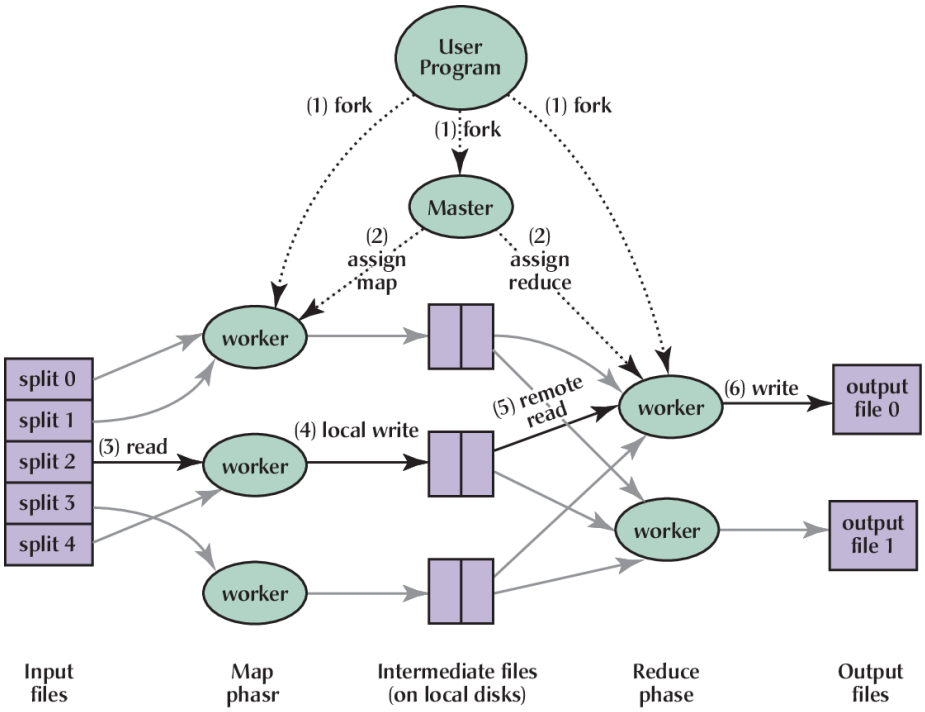


Figure 15: MapReduce architecture.

✔ Advantages

- Works well on commodity hardware.[1]

⚠ 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.

[1] 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.

5.6.2 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 processing of streaming data.

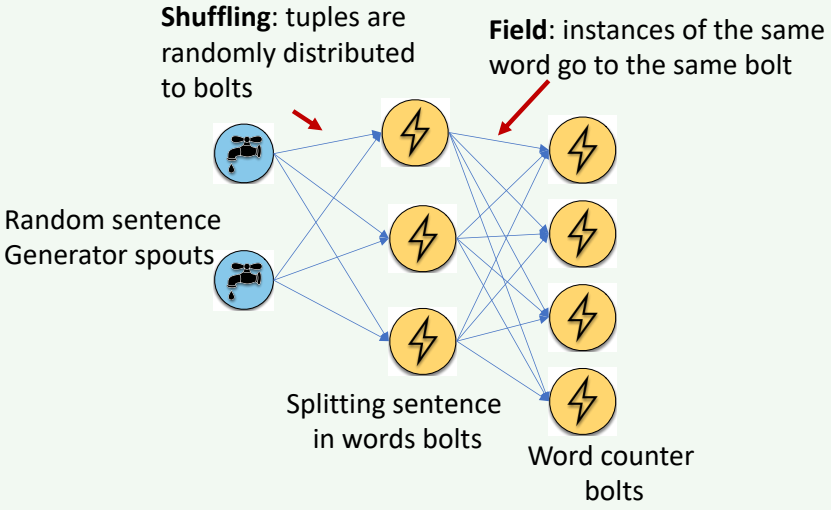
Some features:

- 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.

Example 2: example of topology with different groupings



5.6.3 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.

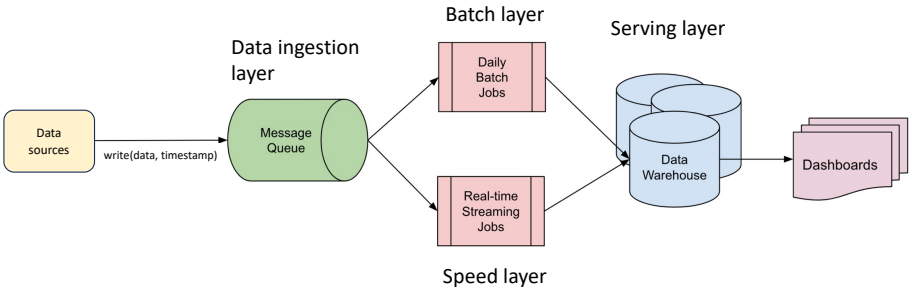


Figure 16: Lambda architecture.

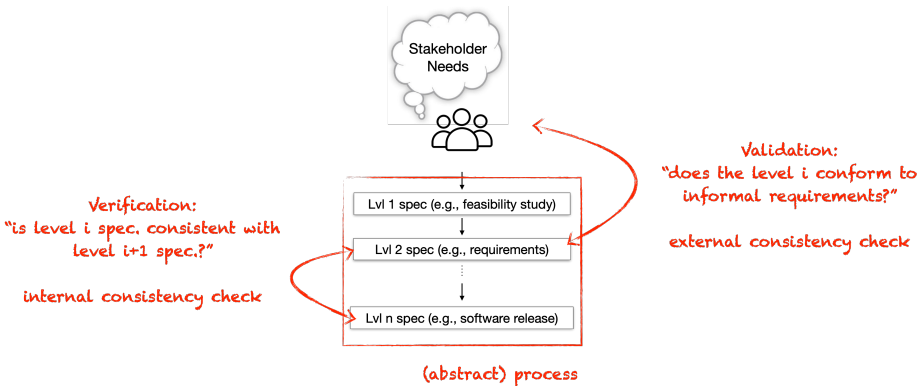
Exist also **Kappa architecture**. Kappa architecture is a software architecture used for processing streaming data with a single technology stack. It is a simplification of Lambda architecture, where the data is processed in batches. Kappa architecture ingests data into a messaging system like Apache 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.

6 Verification and Validation

6.1 Terminology

There are some differences between the terms verification and validation.

First of all, the verification is internal. Despite 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**.

The PMBOK guide, also adopted by the IEEE as a standard, defines them as follows in its 4th edition:

- **Validation**. The assurance that a **product**, **service**, or **system meets the needs of the customer and other identified stakeholders**. It often involves acceptance and suitability with external customers. Contrast with verification.
- **Verification**. The **evaluation of whether or not a product**, **service**, or **system complies with a regulation**, **requirement**, **specification**, or **imposed condition**. It is often an internal process. Contrast with validation.

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 fulfilment 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 artefact shall be the subject of QA; even the verification artefacts 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.

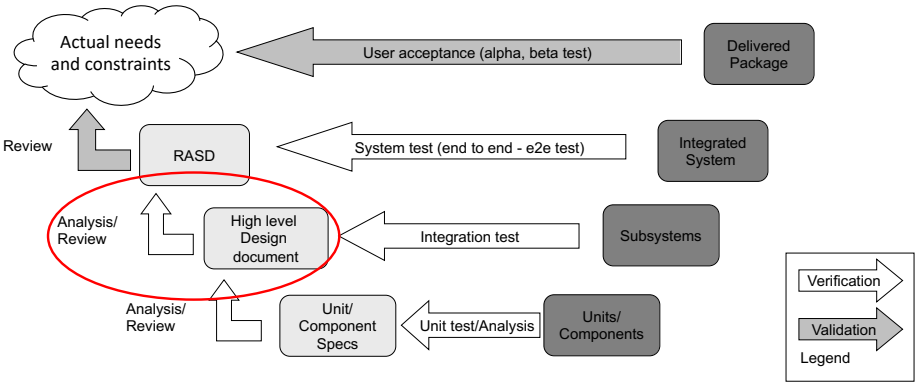


Figure 17: The V-model; verification is emphasized on the left.

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 artefacts 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 behaviour 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**.

6.1.1 Study concurrent use of resources at architectural level

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

A **Petri 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.

A Petri net is a four-tuple:

$$PN =< P, T, I, O > \tag{1}$$

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

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

$$PN =< P, T, I, O, M^0 > \tag{2}$$

Formula called also **marked Petri net**.

You can find a detailed explanation here. 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.

In fact, if two transitions are enabled at the same time, they can fire simultaneously unless the firing of one transition disables the other. Petri nets are **suitable for modelling concurrent systems**. On this page you can see a live simulation by clicking on the nodes.

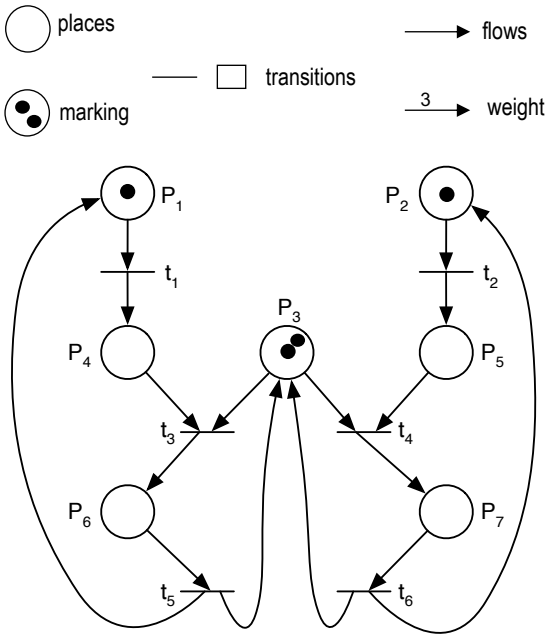


Figure 18: **Example** of Petri nets.

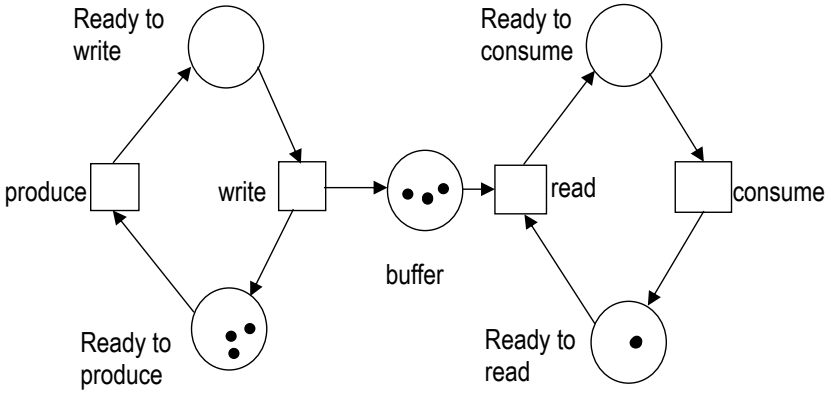


Figure 19: **Example** of Petri nets of producer-consumer model with unbounded buffer.

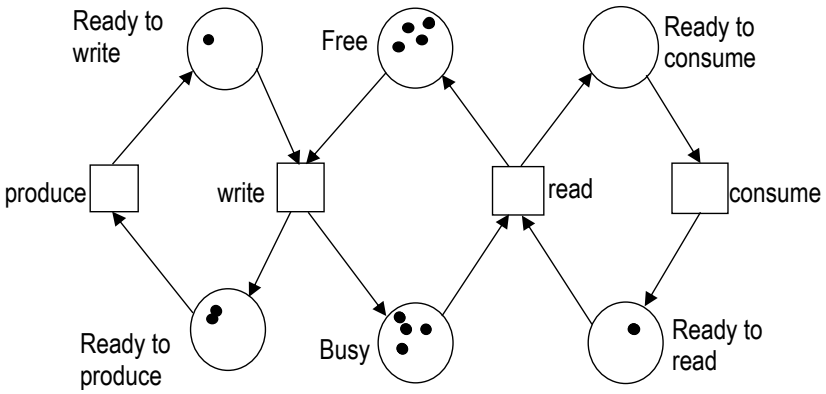


Figure 20: **Example** of Petri nets of producer-consumer model with finite buffer with a parametric number of positions.

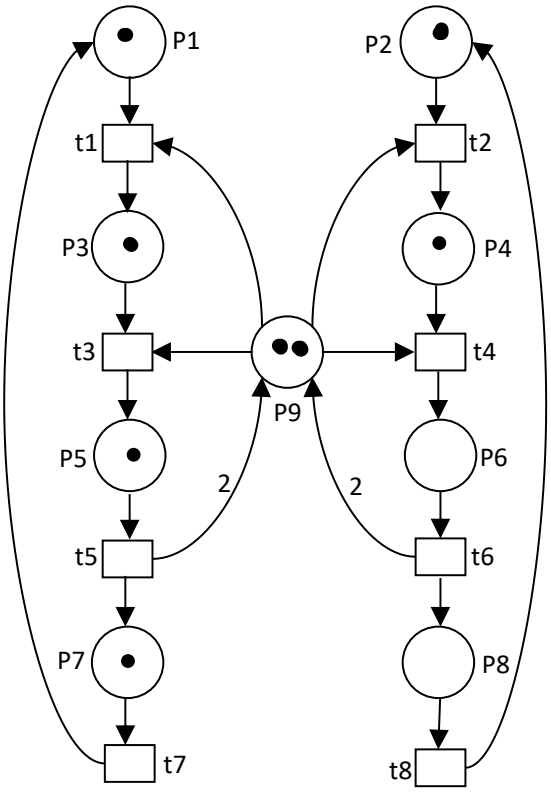


Figure 21: **Example** of Petri nets of deadlock.

6.1.2 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.

Why is availability so important?

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.

Study the System Life-Cycle to use availability metrics

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

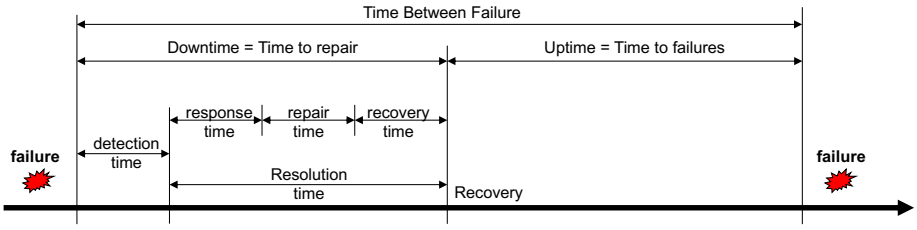


Figure 22: The System Life-Cycle when faults occur.

- **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.

Definition 1: Availability Metric

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):

$$A = \frac{\texttt{MTTF}}{\texttt{MTTF} + \texttt{MTTR}} \tag{3}$$

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): $\texttt{MTBF} \approxeq \texttt{MTTF}$.

Does an easier notation than the availability metric work?

The availability metric is crucial for understanding how a component works at a particular time, but sometimes, we need an easier notation to represent availability. In these cases, we use the **nines notation**.

Nines are an informal logarithmic notation for proportions very near to one or, equivalently, percentages very near 100%. Nines are the number of consecutive nines in a percentage such as 99% (two nines). The number of nines of a proportion x is:

$$\texttt{nines} = -\log_{10}(1 - x) \tag{4}$$

In the computer system availability (our context), a *one nine* (90%) uptime indicates a system that is available 90% of the time or, more commonly described, unavailable 10% of the time.

Availability	Downtime
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

Table 2: Some nines notation and downtime values.

Now that we have the theory and the tools, what methodology should we use to analyze the impact of the architectural choices?

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 series** mean that if one element fails, the whole combination fails.
- **Elements operating in parallel** 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**.

$$A = \prod_{i=1}^{n} A_i \tag{5}$$

Example 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.

The downtime is calculated using the following formula:

$$\texttt{Downtime} = (1 - A) \times 365\,\texttt{days/year}$$

Note that the A value is the Availability in terms of simple values and not as percentages (99% become 0.99).

In the previous example, we can see how the low Availability of Component 1 negatively affects the Availability of the entire system. 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 are not available.

$$A = 1 - \prod_{i=1}^{n} (1 - A_i) \tag{6}$$

Example 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.

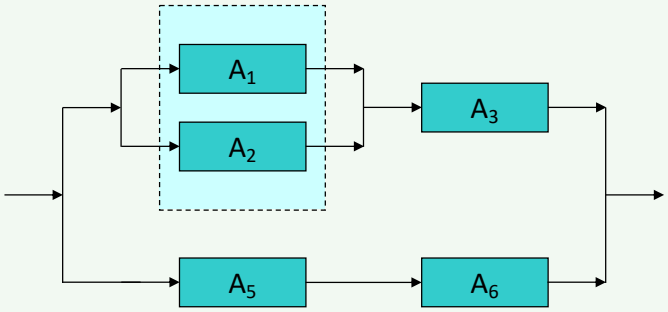
Despite the previous example, the combined availability is 99.99% (4-nines) with 52 minutes/year of downtime.

Even though components with very low Availability are used, the system's overall Availability is much higher than the Availability in series!

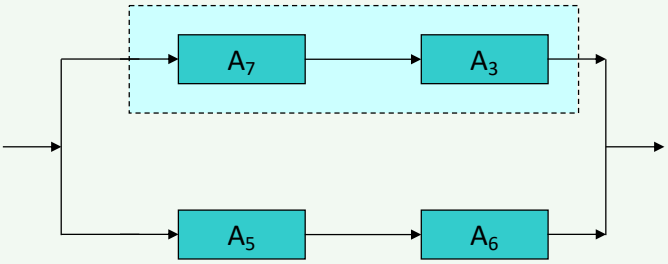
Example 3

Some examples of complex systems:

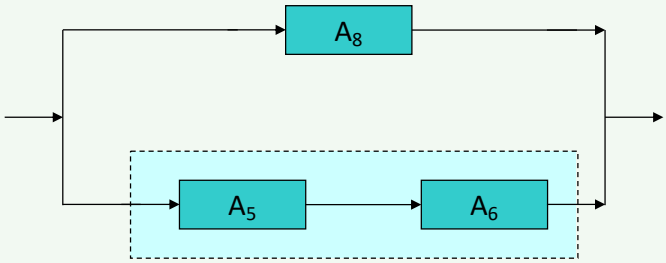
- $A_7 = 1 - (1 - A_1)(1 - A_2)$



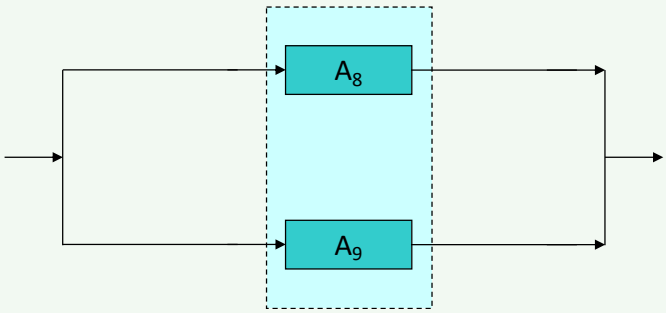
- $A_8 = A_7 A_3$



- $A_9 = A_5 A_6$



- $A = 1 - (1 - A_8)(1 - A_9)$



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**.

Definition 2

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

Some well-known tactics are:

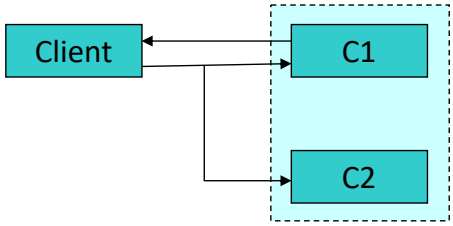
- **Replication**
- **Forward error recovery**
- **Circuit breaker**

Replication approaches

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

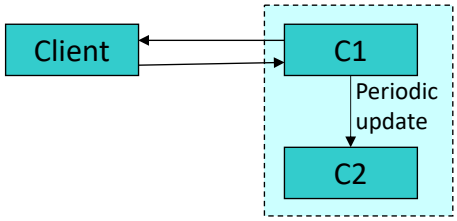
1. **Hot spare**: One component leads, and another is always ready to take over.

   In the following example, `C1` leads, `C2` is always ready to take over.



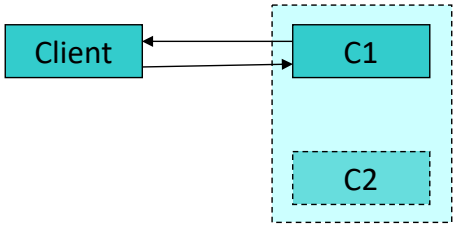
2. **Warm spare**: One component leads and periodically updates another component. If the primary component fails, the second component takes time to update itself fully.

   In the following example, `C1` leads and periodically updates `C2`. If `C1` fails, some time might be needed to fully update `C2`.



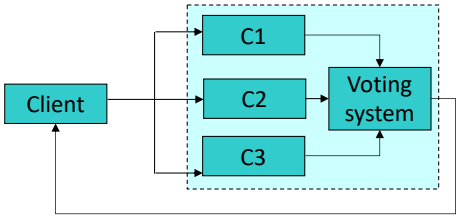
3. **Cold spare**: A second component is dormant, started, and updated only if required.

   In the following example, `C2` is dormant, started, and updated only if required.



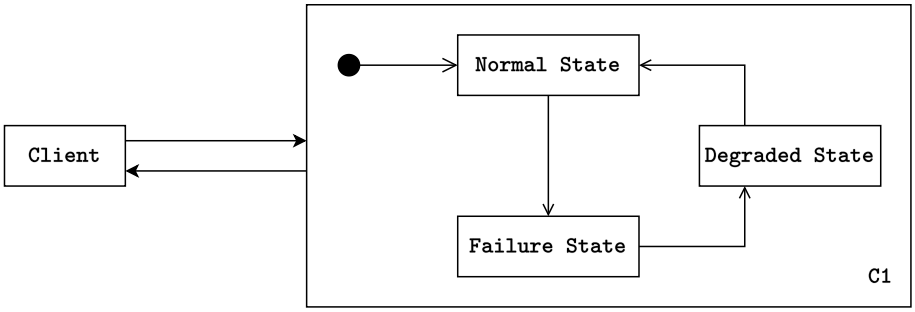
4. **Triple modular redundancy**: Three components are always active, and the result is the one produced by the majority. **This is good when reliability is also important**.

   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

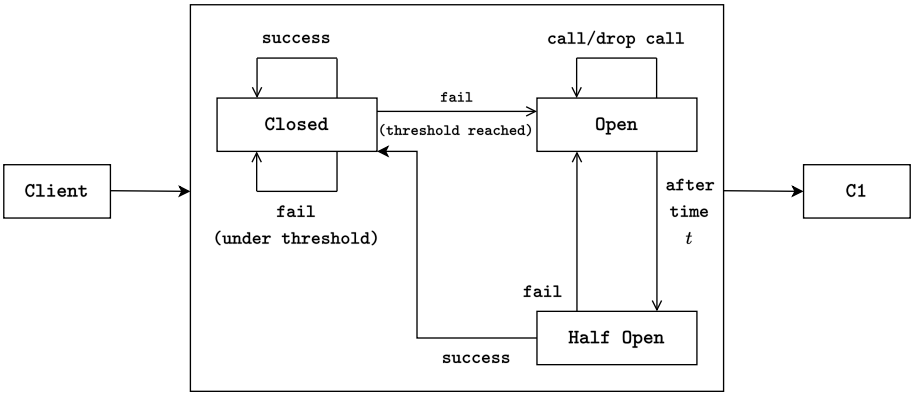
The **Circuit Breaker (CB)** tactic is a client-side resiliency pattern. The CB acts as a proxy for a remote component:

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.



6.2 Analysis: Symbolic Execution

As we have discussed in the previous subsection, there are two types of approaches when we speak about verification:

- **Static Analysis**. It 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*

- Testing (dynamic analysis) is made at runtime and is related to the software's behavior during execution. The analysis is also made on specific inputs.

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

Definition 3

Symbolic Execution (also symbolic evaluation) **analyses a program to determine what inputs cause each part of a program to execute**.

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:

```
1 ...
2 k:       try {
3 k+1:         ...
4 L-1:     } catch (e) {
5 L:           /* error */
6 ...      }
```

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**.

  Then p will be:

$$p =< 0, 1, \ldots, k, \ldots, n >$$

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.

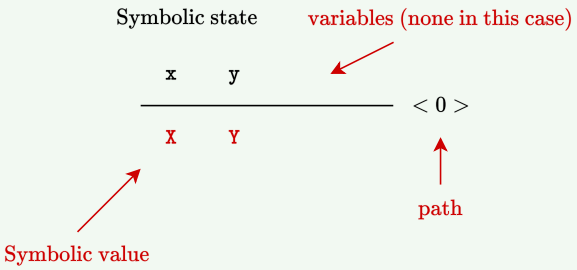
In the following example we can see a complete example of symbolic execution. But before we do, let us introduce some **limitations** of this methodology.

- 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.

Example 4

1. First we introduce the annotation:

```
1 void foo(int x, int y) {
2     ...
```



2. We introduce a local variable:

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

Symbolic state

x	y	z	
X	Y	X	$< 0, 1 >$

3. We introduce a condition. A **path condition** π represents a constraint on a path:

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

- if condition true

Symbolic state

x	y	z	π	
X	Y	X	$X < Y$	$< 0, 1, 2 >$

- if condition false

Symbolic state

x	y	z	π	
X	Y	X	$X \geq Y$	$< 0, 1, 2 >$

4. **Execution continues along feasible paths**. In this case, the path condition π is satisfiable:

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

Symbolic state

x	y	z	π	
X	Y	2X	$X < Y$	$< 0, 1, 2, 3 >$

5. Another if condition:

```
1 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 true

Symbolic state

x	y	z	π
X	Y	2X	X < Y

$< 0, 1, 2, 3, 4 >$

$X < Y \wedge 2X \geq Y$

- if condition false

Symbolic state

x	y	z	π
X	Y	2X	X < Y

$< 0, 1, 2, 3, 4 >$

$X \geq Y \vee 2X < Y$

6. Possible outcomes of symbolic execution:

```
1 void foo(int x, int y) {
2     int z := x
3     if (z < y)
4         z := z*2
5     if (x < y && z >= y)
6         print(z)
7 }
```

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

Symbolic state

x	y	z	π
X	Y	2X	X < Y

$< 0, 1, 2, 3, 4, 5 >$

$X < Y \wedge 2X \geq Y$

(b) **Unsatisfiable** exit (π is not satisfiable): the given **property cannot be satisfied by any concrete execution**.

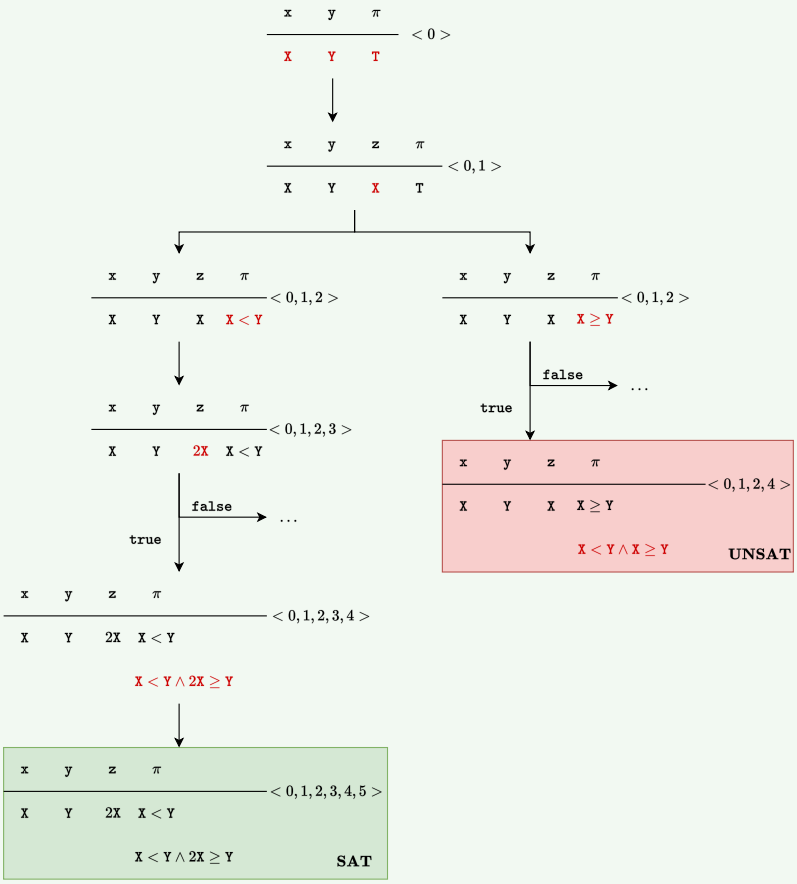
Symbolic state

x	y	z	π
X	Y	2X	X < Y

$< 0, 1, 2, 4 >$

$X < Y \wedge X \geq Y$

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`.



To view the tree in high resolution, scan (or click) the QR code below.

6.3 Testing: terminology, types of testing activities

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

Other *common goals* are:

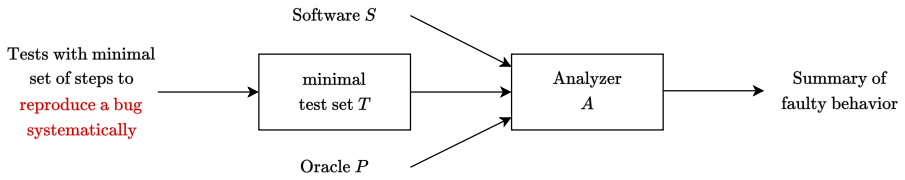
- Exercise different parts of a program to **increase coverage**;
- Make sure the **interaction between components works** (*integration testing*);
- Support **fault localization** and **error removal** (*debugging*);
- Ensure that **bugs introduced in the past do not happen again** (*regression testing*).

The dynamic analysis **analyzes program behavior**. The **properties** are **encoded as executable oracles representing expected outputs** and **desired conditions** (assertions).

It can **run only finite test cases**, so it's not exhaustive verification. The **failures have concrete inputs that trigger them**, and the **execution is automatic**.

We have often heard about *debugging*, but what is it?

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



... and *test case*?

Definition 4

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**. An example of test case specification can be *the input must be a sentence composed of at least two words*, and an example of test case input is *this is a good test case input*.

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.

The strategies for the integration test are many:

- **Big Bang**: **test only after integrating all modules** (not even a real strategy).

  ✔ Pros

  It doesn't require stubs; it only **requires fewer drivers/oracles**.

  ⚠ Cons

  1. Minimum: observability, fault localization/diagnosability, efficacy, feedback;
  2. **High cost of repair** (cost of repairing a fault increases as a function of time between the introduction of an error in the code and repair).

- **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.

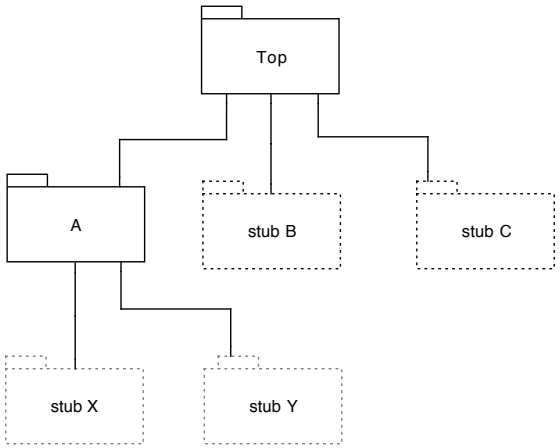


Figure 23: **Example** of top-down strategy.

* ***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). Can this be a disadvantage? Maybe, because the newly developed module may replace an existing driver, and new modules require new drivers.
Another thing to consider is that **it may create several working subsystems**, and each working subsystem will eventually be integrated into the final one.

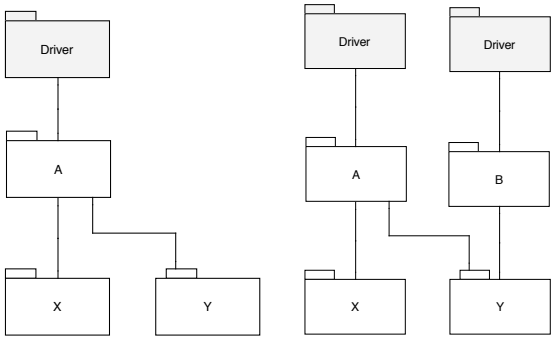


Figure 24: **Example** of bottom-up strategy.

– **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.

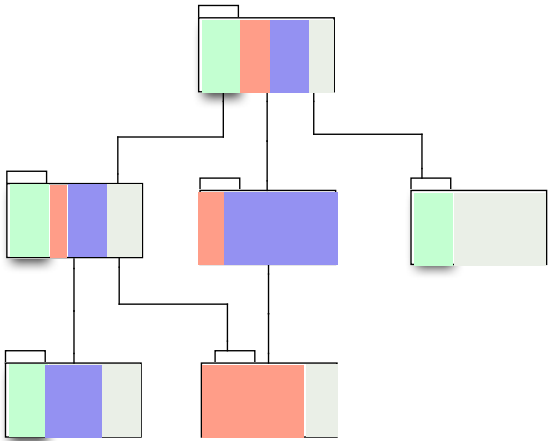


Figure 25: **Example** of threads strategy.

– **Critical**. The critical modules strategy **starts with the highest risk modules**. Risk assessment is a necessary first step. It can include technical risks (e.g. is X feasible?) and process risks (e.g. is the schedule for X realistic?). It may also be similar to a priority process.

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*.

Which one should we choose?

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**.

6.3.1 E2E Testing

Definition 5

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.**[a]

[a] Engineering Fundamentals Playbook

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**

  Purpose

  Check whether the **software meets the functional requirements**.

  How?

  Use the software as described by use cases in the RASD (pag. 9), check whether requirements are fulfilled.

- **Performance Testing**

  Purpose

  1. Detect **bottlenecks** affecting response time, utilization, throughput
  2. Detect **inefficient algorithms**
  3. Detect **hardware/network issues**
  4. Identify **optimization possibilities**

  How?

  Load the system with the expected workload and measure and compare acceptable performance.

- **Load Testing**

  Purpose

  1. **Expose bugs** such as memory leaks, mismanagement of memory, buffer overflows
  2. Identify **upper limits of components**
  3. **Compare alternative architectural options**

  How?

  Test the system at increasing workload until it can support it, and load the system for a long period.

- **Stress Testing**

  Purpose

  Make sure that the **system recovers gracefully after failure**.

  How?

  Trying to break the system under the test by overwhelming its resources or by reducing resources.

  For **example** double the baseline number for concurrent users/HTTP connections, or randomly shut down and restart ports on the network switches/routers that connect servers.

6.4 Test case generation

6.4.1 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 26: Testing Workflow.

As we can see in Figure 26, 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** (analyzed in the section 6.4.2 on page 82). 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.

6.4.2 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 (definition on page 69) **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 $\xrightarrow{\text{to}}$ Symbolic, derive conditions to explore new paths.
- Symbolic $\xrightarrow{\text{to}}$ Concrete, simplifying conditions to generate concrete inputs.

Let's take an example to clarify the explanation.

Example 5

See the code below:

```
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.")
# end
```

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.
The annotation used will be the same as in the Symbolic Execution example on page 70.

1. Let's start with defined parameters and no condition state:

x	y	z	π	$< 0 >$
X	Y		T	
22	7			

2. Call the function with the parameter `y`, i.e. value `7`. The return value of the called function will be `14`.

x	y	z	π	
X	Y	bb(Y)	T	$< 0, 1 >$
22	7	14		

3. The if condition `14 == 22` is obviously false, so the path will be `6` and `7`. By the way, the condition can also be seen as a logical condition $\neg(\texttt{bb}(\texttt{Y}) \neq \texttt{X})$. But it can't be solved, so the concolic execution goes from symbolic to concrete $\neg(14 \neq \texttt{X}) \equiv 14 = \texttt{X}$.

x	y	z	π	
X	Y	bb(Y)	bb(Y) ≠ X	$< 0, 1, 2, 6, 7 >$
22	7	14		

---

4. After a random concrete input, we can solve the constraint and start a new exploration. The parameter value of `X` is now `14`. This is because we do not want the condition from the previous step to be *not equal*.

x	y	z	π	
X	Y		T	$< 0 >$
14	7			

5. The result of the function `bb(7)` is `14`.

x	y	z	π	
X	Y	bb(Y)	T	$< 0, 1 >$
14	7	14		

6. Unlike before, the `if` condition is now true!

x	y	z	π	
X	Y	bb(Y)	bb(Y)=X	$< 0, 1, 2 >$
14	7	14		

7. The evaluation of the third line of code needs no explanation.

x	y	z	π	$< 0, 1, 2, 3 >$
X	Y	Y+10	bb(Y)=X	
14	7	17		

8. The path just explored explores the whole code. To try another path, we can negate the condition, but be careful! We only need to negate the second condition, because we already negated the first condition in the first exploration.

$$\texttt{bb}\,(\texttt{Y}) = \texttt{X} \wedge \texttt{X} \leq \texttt{Y+10} \implies \texttt{bb}\,(\texttt{Y}) = \texttt{X} \wedge \texttt{X} > \texttt{Y+10}$$

x	y	z	π	$< 0, 1, 2, 3, 4, 5, 6, 7 >$
X	Y	Y+10	bb(Y)=X	
14	7	17	$\texttt{X} \leq \texttt{Y+10}$	

---

9. Explore the new path using two (random) values that satisfy the previous logical condition.

x	y	z	π	$< 0 >$
X	Y		T	
34	17			

10. The result of the bb function is 34.

x	y	z	π	$< 0, 1 >$
X	Y	bb(Y)	T	
34	17	34		

11. The `if` condition is obviously true.

x	y	z	π	$< 0, 1, 2 >$
X	Y	bb(Y)	bb(Y)=X	
34	17	34		

12. We add `10` to the variable `Y`.

x	y	z	π	
X	Y	Y+10	bb(Y)=X	$< 0, 1, 2, 3 >$
34	17	27		

13. In this case the `if` condition is false and we complete the exercise because we have explored every possible path!

x	y	z	π	
X	Y	Y+10	bb(Y)=X	$< 0, 1, 2, 3, 4, 6, 7 >$
34	17	27	X > Y+10	

References

[1] F. Bomarius, M. Oivo, P. Jaring, and P. Abrahamsson. *Product-Focused Software Process Improvement: 10th International Conference, PROFES 2009, Oulu, Finland, June 15-17, 2009, Proceedings*. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2009.

Index

A

Allocation 28
Analysis Methodology 63
Apache Storm 52
availability 61
availability metric 62

B

Batch processing 49
Big Bang 75
Black Box 81
Build Plan 75

C

Circuit Breaker (CB) 68
Class Diagram 31
Client-Server Architecture 38
Coding and Unit Test 9
Cold spare 67
Combinatorial 81
Communication coupling 35
Component Diagram 29
Component-and-connector (C&C) 28
Compute-intensive applications 12, 13
Concolic Execution 81, 82
Constraints requirements 20
Content coupling 35
Contract principle 38
Control coupling 35
Correctness 13
Coupling 35

D

Data-intensive applications 12, 14
Debugging 74
defect 14
Deployment 10
Deployment Diagram 33
Design 9
Divide and Conquer 34
Domain properties 18
Dynamic Analysis (Testing) 56

E

End-to-end (E2E) 79
Event-Driven Architecture 45
Evolvability 14
Execution Tree 73

F
fault 14
fault-tolerant 14
Feasibility Study 9
forking 40
Forward Error Recovery 67
Functional requirements 20
Functional Testing 79
Fuzz 81

G
Goals 18

H
Hardware Faults 15
Hot spare 66

I
Integration and System Test 9
Integration Testing 75
interface design 38
Iterative and incremental strategies 76

K
Kafka 46
Kappa architecture 54

L
Lambda architecture 54
Least surprise principle 38
Load Testing 80

M
machine 17
Maintainability 13, 14
Maintenance 10
MapReduce 50
Mean Time Between Failures (MTBF) 62
Mean Time to Failures (MTTF) 62
Mean Time to Repair (MTTR) 62
Metamorphic 81
Module 28
multiple interfaces 39

N
N-tier architecture 43
nines notation 62
Non-functional requirements (NFRs) 20

O
Operability 13

P

Package Diagram 32
Parallel and Distributed Computing (PDC) 11
Path Feasibility 70
Performance 13
Performance Testing 79
Petri net 58
Portability 13

Q

Quality Assurance (QA) 55

R

Reachability 69
Reliability 14
Replication 66
Requirement Engineering 16
Requirements 18
Requirements Analysis and Specification 9
Requirements Elicitation 22
resilient 14

S

Scalability 14
scenario 22
Search-based 81
Sequence Diagram 30
Simplicity 13
Small interfaces principle 38
Software Architecture (SA) 27
Software Faults 15
Static Analysis 56, 69
Stream processing 49
Stress Testing 80
Symbolic Execution 69
system failure 14
System Life-Cycle 61

T

Tactic 66
Test Case 74
Test Case Specification 75
Test Plan 75
Testing 74
Testing Workflow 81
Three-tier architecture 43
Triple modular redundancy 67

U

Unit Testing 75
Use Cases 23

V

V-model 56
Validation 55
Verification 55

W

Warm spare 66
waterfall model 9
White Box 81
world 17