Software Engineering II
Theory

Christian Rossi

Academic Year 2023-2024

Abstract

The objective of the course is to teach the principals methods and processes of software engineering needed to develop complex and qualitative software.

The course covers the following arguments:

- Software process and its organization.
- Modelling languages.
- Requirements analysis and definition.
- Software development methods and tools.
- Approaches for verify and validate the software.

Contents

CHAPTER 1

Introduction

1.1 Definition

The realm of software engineering is dedicated to unraveling the multitude of challenges that emerge during the creation of extensive software systems. These systems are inherently intricate due to their substantial scale, the collaboration of individuals from diverse fields, and the necessity for continuous adjustments to meet evolving demands (both during development and post-installation).

Definition. *Software engineering* is a methodical and managerial discipline that revolves around the systematic creation and upkeep of software products, all of which are crafted and sustained within predefined, controlled timeframes and cost constraints.

In contrast to traditional programming, where a programmer typically crafts an entire piece of software based on known specifications independently, software engineers embark on a distinct path. They identify requirements, shape specifications, design components meant to interconnect with others, and engage in collaborative efforts within a team setting. The key competencies of a software engineer encompass technical acumen, managerial prowess, cognitive aptitude, and organizational skills.

1.2 History

In its early days, software was regarded as an art form. Computers primarily served the purpose of mathematical problem-solving, with the designers themselves acting as the end users. The initial programs were crafted using low-level languages and were subject to stringent resource constraints.

However, as the demand for customized software surged, this artistic endeavor transitioned into a more methodical craft. Developers began creating programs tailored for a broader audience, employing new high-level languages. Towards the culmination of this era, a "software crisis" loomed, marked by the escalating complexity of software and a dearth of effective software development techniques.

With the intention of tackling this growing challenge, the term was coined during a NATO conference in 1968. The primary focus of this pivotal gathering encompassed the following key areas:

- Development of software and the establishment of standards.
- Strategic planning and proficient management.
- Automation of software development processes.
- Modularization of software design.
- Rigorous quality assurance and verification.

1.3 The process and product

The creation of a software program necessitates a well-defined process. Both the software itself and the procedures employed possess inherent quality, and it is imperative for the software engineer to strive for the highest quality possible. This is essential because the process directly influences the ultimate outcome.

Software distinguishes itself from conventional product types in several ways, as it is:

1. Intangible, making it challenging to precisely describe and evaluate.
2. Adaptable or malleable, capable of being modified to suit evolving requirements.
3. Human-intensive, devoid of straightforward manufacturing processes.

Software quality is influenced by various critical factors, including development technology, process quality, the caliber of individuals involved, cost considerations, and adherence to predefined schedules. Software quality attributes encompass:

- Correctness: ensuring that the software aligns with specified requirements.
- Reliability: minimizing the probability of failure absence over a defined period.
- Robustness: assessing the software's ability to perform reasonably in unforeseen scenarios.
- Performance: evaluating the efficient utilization of resources.
- Usability: focusing on the ease of use for the anticipated user base.
- Maintainability: the capacity to facilitate software maintenance.
- Reusability: extending maintainability to components, promoting their reuse.
- Portability: the adaptability of software to diverse target environments.
- Interoperability: ensuring harmonious coexistence and cooperation with other applications.

Process quality attributes consist of:

- Productivity: assessing the efficiency and output of the development process.
- Unity of effort (person-month): measuring the collective contribution of effort over a month.
- Delivered item (lines of code and function points): quantifying the tangible output of the development process.
- Timeliness: the capability to respond to change requests in a timely fashion.

1.4 Development process

In the initial stages of software development, there was a lack of reference models, leading to a straightforward "code and fix" approach. In response to the previously mentioned software crisis, the necessity for a structured model became evident. The first comprehensive model introduced was the "waterfall" model, characterized by the following fundamental requirements:

1. Identification of distinct phases and associated activities.
2. Enforcement of a linear progression, with no backtracking between phases.
3. Standardization of outputs generated at each phase.
4. Viewing software development as akin to a manufacturing process.

Following the advent of the waterfall model, numerous more flexible methodologies emerged, including iterative models, the agile movement, and DevOps, which aimed to offer more adaptable approaches to software development.

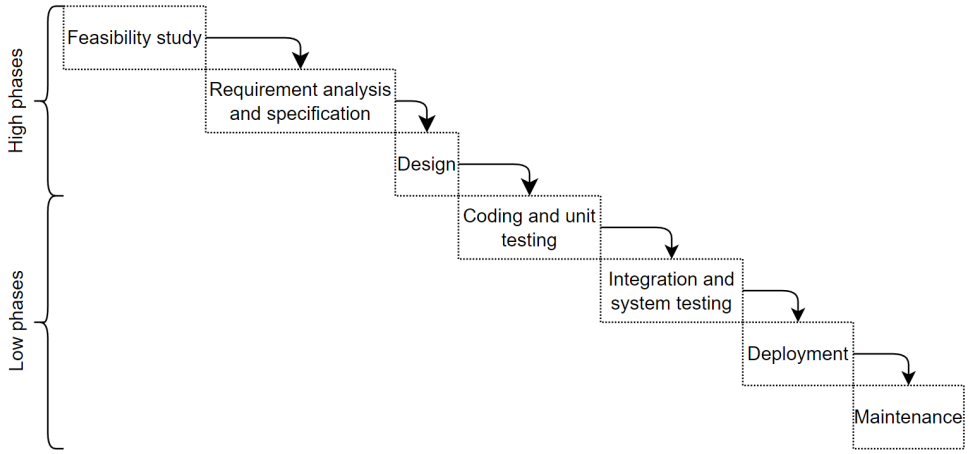


Figure 1.1: Waterfall process model

The key phases depicted in the image encompass the following:

1. Feasibility study and project estimation: this stage determines whether the project should commence, exploring possible alternatives and necessary resources. It yields a "feasibility study document" comprising an initial problem description, scenarios outlining potential solutions, and cost and schedule estimates for various options.
2. Requirement analysis and specification: in this phase, the domain in which the application operates is meticulously analyzed. Requirements are identified, and software specifications are derived, leading to the creation of a "requirement analysis and specification document".
3. Design: this step outlines the software architecture, defining components, their relationships, and interactions. The objective is to facilitate concurrent development and allocate responsibilities. It culminates in the production of a "design document".
4. Coding and unit test: each module is implemented and rigorously tested. Additional quality assurance measures, such as inspections, may be employed. Programs are accompanied by their respective documentation.

5. Integration and system test: modules are integrated into systems, and the integrated systems undergo testing. This phase, as well as the preceding one, can be integrated into an incremental implementation scheme.
6. Deployment.
7. Maintenance; the maintenance phase comprises various aspects, including:
    - Corrective: addressing and rectifying identified faults or defects.
    - Adaptive: adapting the software to changes in its environment.
    - Perfective: accommodating new or modified user requirements.
    - Preventive: involving activities aimed at enhancing the system's maintainability.

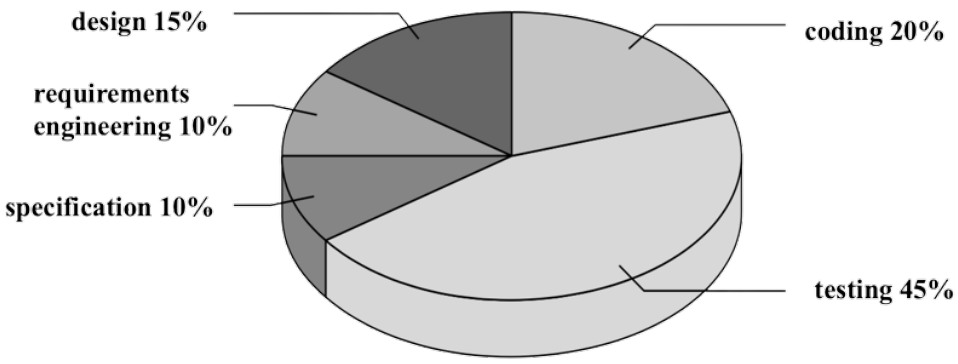


Figure 1.2: Effort in each phase

The primary challenges associated with software evolution include:

- It is seldom anticipated and strategically planned.
- Software is highly amenable to change, with modifications often directly affecting the code, leading to inconsistencies in project documentation.

Effectively addressing software evolution necessitates the implementation of sound engineering practices, comprising two key steps: first, modifying the design, and then making corresponding adjustments in the implementation, ensuring the consistent updating of all associated documents. Indeed, one of the central objectives of software engineering is to create software that can be designed to accommodate future changes in a dependable and cost-effective manner.

The waterfall model operates as a black-box system, as the company seeking the software provides initial requirements and remains relatively uninvolved throughout the development phase. If a higher degree of transparency and customer interaction is required, an alternative development model must be adopted, one that permits regular customer feedback. With each customer interaction, two key aspects can be evaluated:

- Validation: ensuring that the product aligns with the customer's specific requests.
- Verification: confirming that the product functions correctly and in the intended manner.

The concept of a flexible development process is centered on its adaptability to changes, particularly in requirements and specifications. This approach involves incremental processes that allow for feedback at various stages. There are various forms of flexible development processes, such as SCRUM, extreme programming, incremental releases, rapid prototyping, DevOps, and more.

CHAPTER 2

Requirements engineering

2.1 Definition

The primary measure of success of a software system is the degree to which it meets the purpose for which it was intended.

Definition. Software systems *requirements engineering* is the process of discovering that purpose, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation.

This phase entails pivotal considerations: stakeholder identification, needs assessment, documentation generation, and the analysis, communication, and implementation of requirements. An alternate definition is as follows:

Definition. *Requirements engineering* is the branch of software engineering concerned with the real-world goals for, function of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software engineering behavior, and to their evolution over time and across software families.

2.2 Importance and difficulties

Customer-provided requirements can be categorized into three primary types:

- Functional requirements: These delineate the system's interactions with its environment, irrespective of implementation details. They represent the core objectives that the software must achieve.
- Nonfunctional requirements: These encompass user-visible aspects of the system that aren't directly tied to functional behavior.
- Constraints: these restrictions are either set by the client or are inherent to the system's operational environment.

Nonfunctional requirements, often referred to as quality of service attributes, specify how functionality should be delivered to the end user. While they transcend specific application domains, their relevance and prioritization are influenced by the application's context.

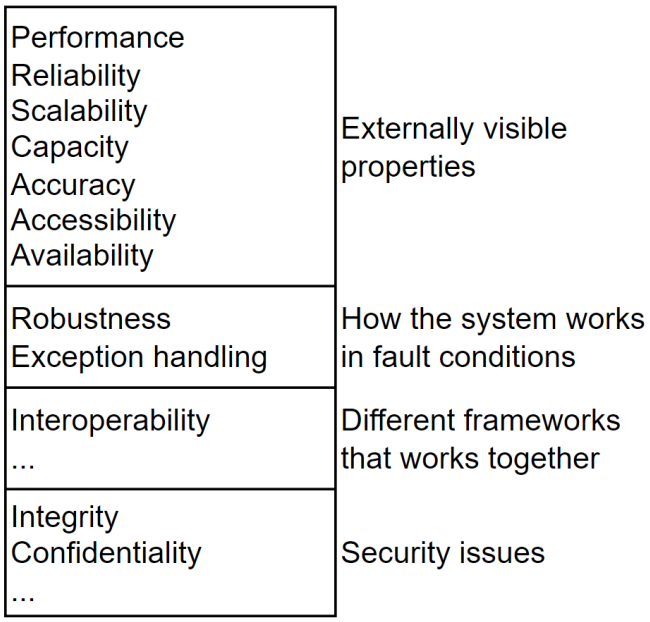


Figure 2.1: Some relevant QoS characteristics

2.3 Requirement engineering process

Inadequate requirements are widespread, and the process of requirement engineering is both challenging and pivotal. This is because issues in the initial stages can potentially escalate costs by a factor of up to two hundred during the final phase.

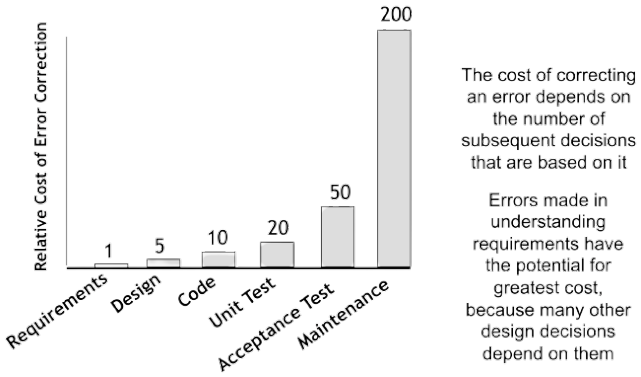


Figure 2.2: Cost of late correction [Boehm, 1981]

The complexity of requirement engineering arises from several factors, including composite systems, the presence of multiple systems, varying abstraction levels, diverse concerns, and involvement of stakeholders with different backgrounds.

Requirement engineers are tasked with various responsibilities, including:

- Eliciting information, which involves gathering details about project objectives, context, scope, domain boundaries, and requirements.
- Modeling and analysis, encompassing the definition of goals, objects, use cases, and scenarios.

- Communicating requirements, which includes providing analysis feedback, documenting in the Requirements Analysis and Specification Document (RASD), and creating system prototypes.
- Negotiating and agreeing upon requirements, which involves resolving conflicts and managing risks, assisting in requirement selection and prioritization.
- Managing and evolving requirements, involves overseeing them across the entire development lifecycle, ensuring backward and forward traceability, and effectively managing changes and their consequences.

2.4 World-machine relationship

The machine signifies the portion of the system to be developed, while the world denotes the segment of the real-world influenced by the machine. It is crucial to note that the machine's purpose always resides within the context of the world.

Requirements engineering is primarily concerned with phenomena occurring in the world, rather than those confined to the machine itself. In essence, requirements models serve as representations of real-world phenomena.

Certain phenomena are shared between the world and the machine. These shared phenomena fall into two categories:

- Phenomena controlled by the world and observed by the machine.
- Phenomena controlled by the machine and observed by the world.

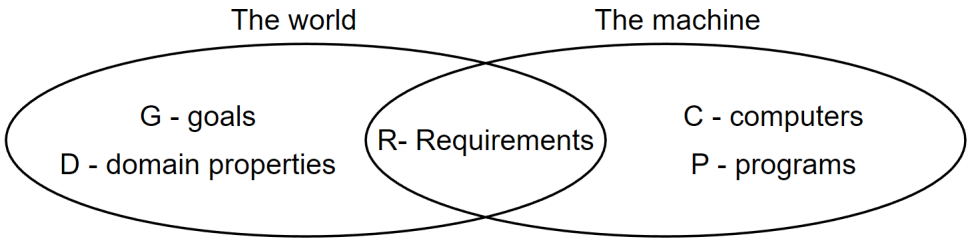


Figure 2.3: Goals, domain assumptions, and requirements

Goals are declarative statements formulated in terms of real-world phenomena, which may or may not be shared. Domain properties and assumptions, on the other hand, are descriptive statements presumed to be valid in the world. Requirements, as prescriptive assertions, are expressed in terms of shared phenomena.

The completeness of requirements, denoted as R, is achieved if:

- R guarantees the satisfaction of goals G within the context of domain properties D, expressed as $R \wedge D \models G$.
- G comprehensively capture all the needs of stakeholders.
- D accurately represent valid properties and assumptions about the world.

2.5 Elicitation of requirements

The complexity in requirement engineering can be managed through various means, including:

- Employing diverse approaches and strategies, and synthesizing the outcomes from each.
- Maintaining proximity to stakeholders.
- Enabling stakeholders to articulate their perspectives.

This scenario can be generalized in terms of:

- Involving participation actors.
- Defining entry conditions.
- Outlining the flow of events.
- Specifying exit conditions.
- Identifying exceptional cases.
- Documenting special requirements.

2.6 Modeling requirements

Definition. A *model* is a depiction, typically in one medium, of an entity in the same or a different medium. It encapsulates the essential facets of the subject while simplifying or omitting extraneous details.

Reality, denoted as R, comprises tangible entities, individuals, processes, and their interconnections. In contrast, a model, symbolized as M, represents an abstraction of these entities, individuals, processes, and the relationships between these abstractions.

To comprehend reality, it necessitates interpretation denoted as I through a mapping function. For a model to be considered effective, it is imperative that the relationships that hold true in reality R also hold true in the model M.

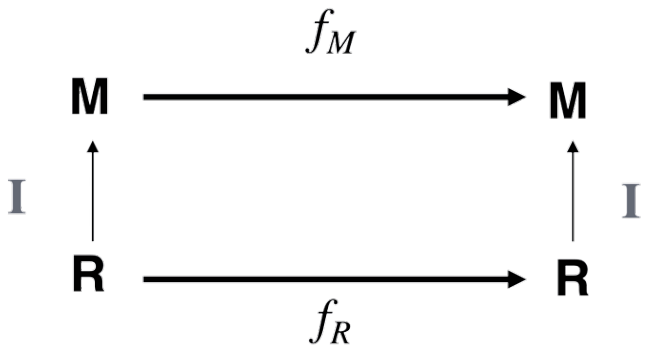


Figure 2.4: Relationship between model M and reality R

Software models serve a variety of purposes, including:

- Capturing and articulating requirements and domain knowledge with precision.

- Facilitating software system design and the generation of practical deliverables.
- Offering a simplified perspective on intricate systems and enabling evaluation and simulation of such complexity.
- Generating prospective system configurations.

The primary modeling challenges revolve around ensuring coherence among different perspectives of the system and addressing the potential for variations in interpretation and ambiguity by establishing clear boundaries for acceptable differences in how the model is understood.

In the domain of requirements engineering, our modeling efforts should primarily focus on:

- Representing the pertinent entities and individuals relevant to the given problem.
- Describing the relevant phenomena.
- Documenting the objectives, requirements, and domain assumptions.

When it comes to choosing the appropriate modeling tools, we have several options at our disposal:

- Natural language (e.g., English, Italian, etc.):
  - Pros: user-friendly and easy to use.
  - Cons: prone to a high level of ambiguity, and it's possible to overlook relevant information.
- Formal language (e.g., FOL, Alloy, Z, etc.):
  - Pros: offers the possibility of employing tools for analysis and validation; this approach compels the user to specify all crucial details.
  - Cons: requires expertise in using the language.
- Semiformal language (e.g., UML):
  - Pros: simpler than a formal language, providing some degree of structure to the models.
  - Cons: not amenable for automated analysis, and still entails some level of ambiguity.
- Mixed approach:
  - Utilizing a semiformal language for foundational aspects.
  - Augmenting semiformal models with explanatory informal text.
  - Leveraging a formal language for the most critical components.

The choice of the modeling tool should align with the specific requirements and characteristics of the project at hand.

2.7 Use cases and requirements

The fundamental steps in formulating use cases encompass the following:

1. Assign a name to the use case.
2. Identify the actors: generalize specific names to encompass participating actors.
3. Focus on the flows of events, entry and exit conditions using natural language.
4. Give due attention to exceptional cases and special requirements.

Each use case can give rise to one or more requirements.

A use case represents a sequence of events within the system, involving interactions with actors. These use cases are initiated by an actor and conclude under specific termination conditions.

Definition. The *use case model* is the set of all use cases specifying the complete functionality of the system.

A *use case association* is a relationship between use cases. The primary types of use case associations include:

- Include, where one use case utilizes another use case.
- Extends, where a use case extends the behavior of another use case.
- Generalization, when an abstract use case has several specialized versions.

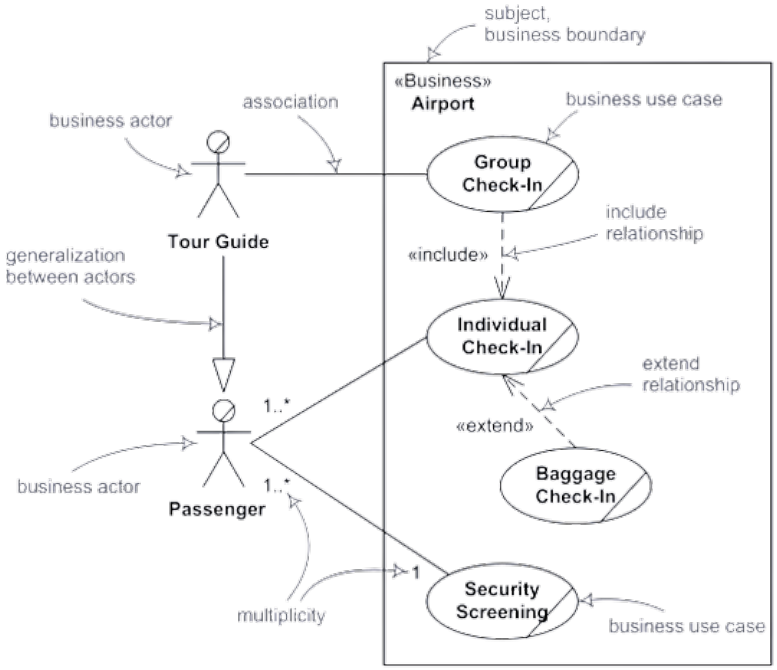


Figure 2.5: Use case model example

2.8 Requirements-level class diagrams

The requirements-level class diagrams are conceptual models for the application domain. They may model objects that will not be represented in the software-to-be. Usually, they do not attach operations to objects: it's best to postpone this kind of decisions until software design.

To find objects and classes we need to:

- Analyze any description of the problem and application domain you may have.
- Analyze your scenarios and use cases descriptions.

Finding objects is the central piece in object modeling. A possible tool to use in the analysis is the Abbott Textual Analysis also called noun-verb analysis: nouns are good candidates for classes and verbs are good candidates for associations and operations.

Example	**Grammatical construct**	**UML component**
"Monopoly"	concrete person, thing	object
"toy"	noun	class
"3 years old"	adjective	attribute
"enters"	verb	operation
"depends on"	intransitive verb	operation (event)
"is a", "either", "or", "kind of"	classifying verb	inheritance
"has a", "consists of"	possessive verb	aggregation
"must be", "less than"	modal verb	constraint

Table 2.1: Abbott textual analysis example

2.9 Dynamic modeling

Dynamic modeling serves the purpose of providing methodologies for modeling interactions, the behavior of participants, and workflow within a system. This is achieved through various diagram types, including sequence diagrams, state machine diagrams, and activity diagrams. During the creation of these diagrams, certain objects become apparent.

A sequence diagram is established by following the flow of events outlined in the use case diagram. It represents objects engaged in a use case scenario using a directed acyclic graph notation. The fundamental rules for creating sequence diagrams include:

- Every event involves both a sender and a receiver.
- The representation of the event is sometimes referred to as a message.
- The sender and receiver for each event must be identified.

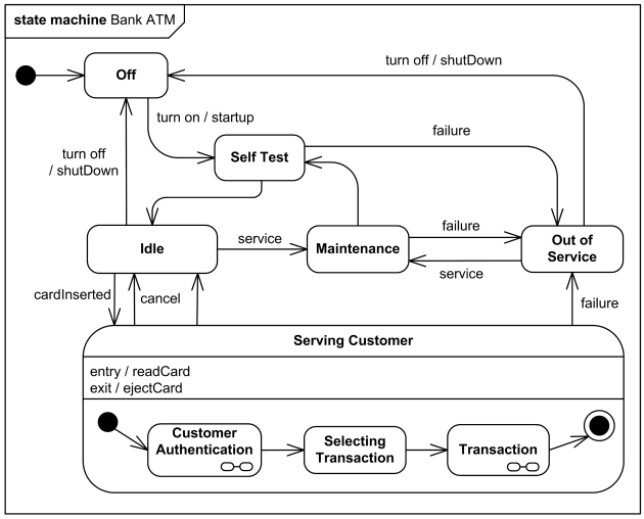


Figure 2.6: Example of a state diagram

For effective dynamic modeling, it is crucial to construct models solely for classes exhibiting substantial dynamic behavior, considering only relevant attributes. Additionally, when deciding on actions and activities, one must account for the granularity of the application and strive to reduce notational complexity.

CHAPTER 3

Alloy

3.1 Introduction

Alloy is a formal notation designed for specifying models of both systems and software. It exhibits characteristics akin to a declarative object-oriented language, underpinned by a robust mathematical foundation. To aid in the practical application of Alloy, a supporting tool is available, facilitating the simulation of specifications and enabling property verification.

Alloy has been crafted with the objective of combining the expressive power found in the Z language with the rigorous automated analysis capabilities offered by the SMW model checker. In the realm of requirement engineering, Alloy finds utility in formally describing the domain, its inherent properties, or the operations that a machine is required to provide. In software design, Alloy serves as a valuable tool for the formal modeling of components and their interdependencies.

Alloy represents a fusion of first-order logic and relational calculus. Key features of this formal language include:

- A thoughtfully curated subset of relational algebra, offering a uniform model for individuals, sets, and relations, with an absence of high-order relations.
- Minimal reliance on arithmetic operations.
- Support for modules and hierarchies to enhance organization and scalability.
- Designed for succinct, illustrative specifications.
- Employs a potent and efficient analysis tool.

3.2 Syntax

Alloy presents bounded snapshots of the world that satisfy the given specification.

It employs bounded exhaustive search to uncover counterexamples to asserted properties using SAT.

Definition. *Atoms* represent Alloy's fundamental entities, characterized by indivisibility, immutability, and a lack of interpretation.

Relations establish connections between atoms, forming sets of tuples, where tuples are sequences of atoms.

In Alloy, relations are inherently typed, with their types determined by the declaration of the relation. The fundamental Alloy relation types include:

- "none" (signifying an empty set).
- "univ" (representing the universal set).
- "iden" (denoting the identity relation).

The logical operators in Alloy encompass:

- Union "$\cup$".
- Intersection "&".
- Difference "$-$".
- Subset "in".
- Equality "=".
- Cross product "$\rightarrow$", analogous to a natural join.
- Dot join "." or "[ ]", where the final element of the first relation joins with the corresponding first elements of the second relation, followed by the removal of the combined element from the relation.

Here are the various binary closures applicable to relations in Alloy:

- Transpose "$\sim$": inverts the order of the elements in the relation.
- Transitive "$\wedge$": it signifies the transitive closure where $^\wedge r = r + r.r + r.r.r + \ldots$.
- Reflexive transitive "$*$": it represents the reflexive transitive closure, where $^* r = \text{iden} + ^\wedge r$.

The possible restrictions are:

- Domain restriction "<:", that restricts the elements on the left side to the set on the right side.
- Range restriction ":>", that is same as before, but the relations are inverted.
- Override "++", that removes the tuples on the left that are in the right relations and adds all the remaining relations of the right relation.

Alloy also includes various Boolean operators:

- Negation ("!" or "not").
- Conjunction ("&" or "and").
- Disjunction ("||" or "or").

- Implication (" $\Longrightarrow$ " or "implies").
- Alternative (",", or "else").
- Bi-Implication (" $\Longleftrightarrow$ " or "iff").

Alloy offers logic quantifiers:

- "all": holds for every element.
- "some": holds for at least one element.
- "no": holds for no elements.
- "lone": holds for at most one element.
- "one": holds for exactly one element.

For defining relations with singletons, you can use the following declaration "x: m e", where "x" is the name of the relation, "m" is the multiplicity of the element (e.g., "set", "one", "lone", or "some") and "e" is the name of the element within the relation. When the relation consists of pairs, the declaration appears as "r: Am $\rightarrow$ nB", where "r" is the relation name, "A" and "B" denote the element names with multiplicities "m" and "n", respectively.

Additionally, Alloy includes operators like "#" (counting the number of tuples in r), integers $(0, 1, \dots)$ for defining variable values, arithmetic operators ("+" and "−"), and various comparison operators ("<", "<=", "=", "=>", ">"). There is also the "sum" operator, which adds all elements within a selected tuple.

Other useful keywords are:

- "let:" this keyword is utilized to establish local variables within a formula or expression. These variables serve to store and reuse intermediate results, making it easier to simplify complex expressions.
- "enum": this keyword is employed to define enumerations. Enumerations enable the definition of a finite set of symbolic values, which can be utilized in models to represent various states, options, or categories.
- "var": this keyword is used for declaring variables within an Alloy model or specification.
- "after": this keyword is employed to specify the temporal ordering or sequence of events or states within a model. It is commonly used in temporal logic to define the sequence of events that must occur before or after a specific state or action.
- "always": this keyword is used to express a property that must remain true throughout a system's execution or under certain conditions. It is often utilized to specify invariant properties in Alloy models.
- "eventually": this keyword is used to express temporal logic properties that indicate a particular condition or event will ultimately occur during the execution of a system. It is commonly used to model and verify properties that describe what should happen at some point in the future.

- "historically": keyword is used to express temporal logic properties that describe a condition or event that has remained true for a continuous duration of time leading up to the present or to a specified point in the model's execution. It is often employed to specify properties related to the historical behavior of a system.
- "before".
- "once".

3.3 Address book example

Let's consider the task of creating a basic address book model. This address book contains a collection of addresses associated with their respective names. In this scenario, there are three primary entities:

- "Name": represents names of individuals.
- "Addr": represents addresses.
- "Book": signifies the context of the address book.

The relationship here is established through the entity "Addr", which links "Name" to "Addr" within the context of "Book". We use the "lone" keyword to indicate that each "Name" can be linked to at most one "Addr". To summarize the preceding descriptions:

```
sig Name {}
sig Addr {}
sig Book {
    addr: Name -> lone Addr
}
```

This specification creates the following relationships:

- Sets are unary relations.
- Scalars are sets consisting of a single element.
- The ternary relation involves these three predicates.

A new predicate can be defined using the "pred" keyword.

```
pred show {}
run show for 3 but exactly 1 Book
```

In the second line, it specifies that we should find a maximum of three elements for each "Book". The previously defined predicate "show" is currently empty and always returns true. We can now create a predicate with specific arguments. For instance:

```
pred show [b: Book]{
    # b.addr > 1
}
run show for 3 but exactly 1 Book
```

In the previous example, the consistent predicate imposes a restriction on the quantity of "Address" relations within a given "Book". In the following example, the consistent predicate enforces a constraint on the number of distinct "Address" entries that appear within the "Book"

```
pred show [b: Book]{
    # b.addr > 1
    # Name.(b.addr) > 1
}
run show for 3 but exactly 1 Book
```

In the subsequent example, the inconsistent predicate utilizes the "some" keyword, indicating the existence of an element. In this scenario, there is only one "Book" so the tool will report that no instances can be identified.

```
pred add [b: Book]{
    # b.addr > 1
    some n: Name | # n.(b.addr) > 1
}
run show for 3 but exactly 1 Book
```

All the preceding predicates are considered static as they do not modify the signature. In Alloy, dynamic predicates are used for dynamic analysis. For instance, we can create a predicate that introduces an "Address" and a "Name" to a "Book" as follows:

```
pred add [b,b': Book, n: Name,a: Addr]{
    b'.addr=b.addr + n -> a
}
pred showAdd [b,b': Book, n: Name,a: Addr]{
    add[b,b',n,a]
    #Name.(b'.addr) > 1
}
run showAdd
```

We can now define a predicate for the "Book" deletions.

```
pred del [b,b': Book, n: Name]{
    b'.addr=b.addr - n -> Addr
}
```

We can verify whether performing a delete operation after an add operation restores us to the initial state or not by employing an "assertion".

```
assert delRevertsAdd{
    all b1,b2,b3: Book,n: Name,a: Addr
    add[b1,b2,n,a] and del[b2,b3,n]
    implies b1.addr=b3.addr
}
```

During the assertion verification process, Alloy looks for counterexamples. In this instance, we will encounter a counterexample, causing the assertion to evaluate as false. To rectify the assertion, we should amend it as follows:

```
assert delUndoesAdd{
    all b1,b2,b3: Book,n: Name,a: Addr |
    no n.(b1.addr) and add[b1,b2,n,a] and del[b1,b2,n]
    implies b1.addr=b3.addr
}
```

In some cases, we may need to retrieve specific signatures. To accomplish this, we can leverage Alloy functions. For instance, we can define a function that searches for a particular "Book" and returns a set of "Address":

```
fun lookup[b: Book,n: Name]: set Addr{
    n.(n.addr)
}
```

3.4 Family relations example

Now, let's explore a family relationship tree. To begin, we must define a generic individual, which can be either male or female.

```
abstract sig Person {
    father: lone Man
    mother: lone Woman
}
sig Man extends Person {
```

```
    wife: lone Woman
}
sig Woman extends Person {
    husband: lone Man
}
```

We establish that each "Person" can have at most one father and one mother (indicated by the keyword "lone"), as we require a root for the family tree. The person at the root should have no parents, possibly due to them being unknown. The "Person" signature is declared as "abstract" because it needs to be specialized into one of the subsequent signatures, either "Man" or "Woman". Signatures, denoted by the "sig" keyword, represent sets of elements. Before this keyword, we can specify the number of entities required ("lone", "one", or "some").

Definition. The *fields* of a signature are relations whose domain is a subset of the signature. The keyword *extends* is used to declare a subset of signature.

To obtain the set of grandfathers of a given individual, we can define a function like this:

```
fun grandpas[p: Person]: set Person {
    p.(mother+father).father
}
pred ownGrandpa[p: Person] {
    p in p.grandpas[p]
}
```

We've additionally created a predicate that verifies if a person is within the set of grandfathers returned by the "grandpas" function. However, the issue at hand is that we haven't imposed constraints on relations. To address this, we need to define two new operators for binary relations:

- Transitive closure: $\hat{}r = r + r.r + r.r.r + \ldots$
- Reflex transitive closure: $*r = iden + \hat{}r$

We can now specify that no one can be their own father or mother:

```
fact {
    no p: Person | p in p.^(mother+father)
}
```

We must also establish a constraint that if X is the husband of Y, then Y is the wife of X:

```
fact {
    all m: Man,w: Woman | m.wife=w iff w.husband=m
}
fact {
    wife = ~husband
}
```

The two statements are equivalent, with the second being expressed using the transpose operator. A fact can encompass multiple constraints, meaning that the previous constraints can be consolidated into a single fact. It's worth noting that facts are global, while predicates need to be invoked explicitly.

CHAPTER 4

Software design

4.1 Software architecture

Definition. The *architecture* of a software system defines the system in terms of computational components and the interactions among these components.

In the context of a software system, the *software architecture* represents the set of structures necessary to reason about the system. These structures include software elements, their relationships, and properties associated with both.

There are three fundamental structures relevant to software architecture:

- Component-and-connector structures: these structures describe how the system is organized as a set of elements with runtime behavior (components) and their interactions (connectors). They allow us to study runtime properties, such as availability and performance, as well as how these structures collaborate.

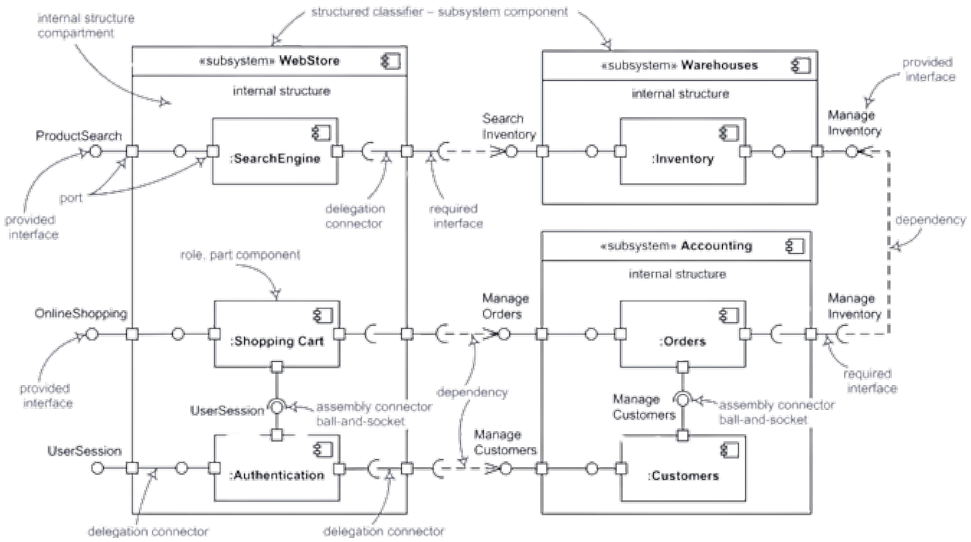


Figure 4.1: UML component diagrams for component and connector structure

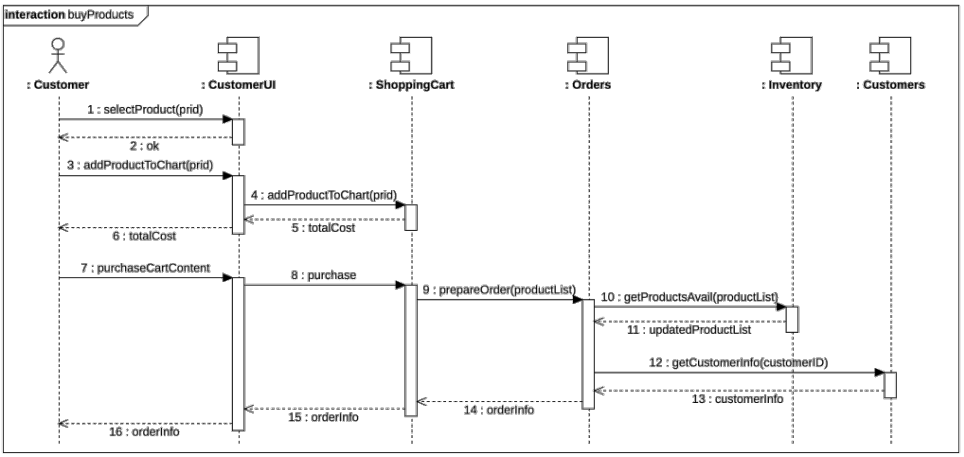


Figure 4.2: UML sequence diagrams for component and connector structure

- Module structures: illustrate how a system is structured as a set of code or data units that need to be procured or constructed, along with their relationships. Modules serve as implementation units and form the basis for work division. Typical relationships among these modules include "uses," "is-a" (generalization), and "is-part-of."

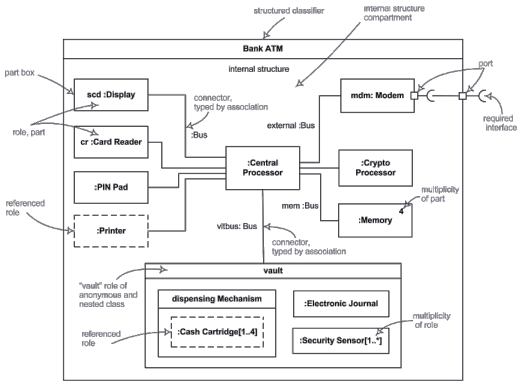


Figure 4.3: UML composite structure diagram for module structure

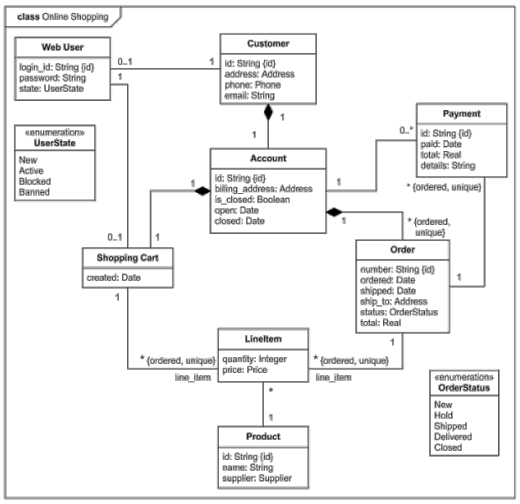


Figure 4.4: UML class diagram for module structure

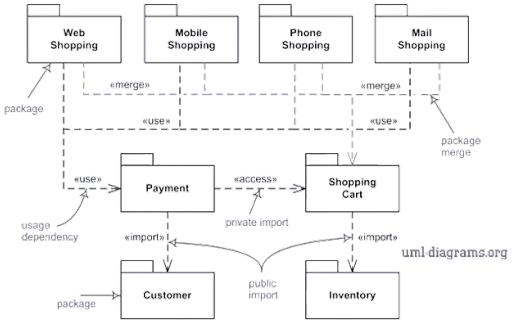


Figure 4.5: UML package diagram for module structure

- Allocation structures: these structures define how the elements from component and connector or module structures map to entities that are not software. Common allocation structures include deployment structure, implementation structure, and work assignment structure. The deployment structure captures the system's hardware topology, specifying the distribution of components and identifying performance bottlenecks.

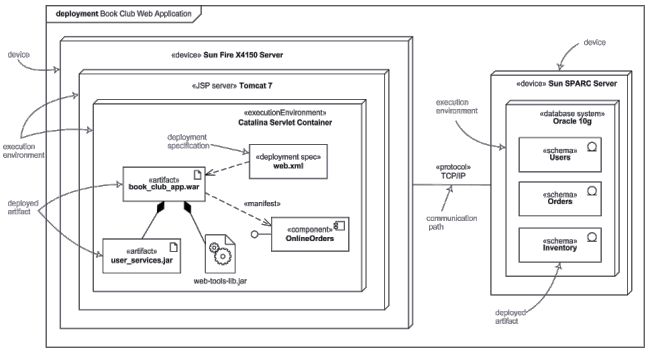


Figure 4.6: UML deployment diagrams and deployment structure

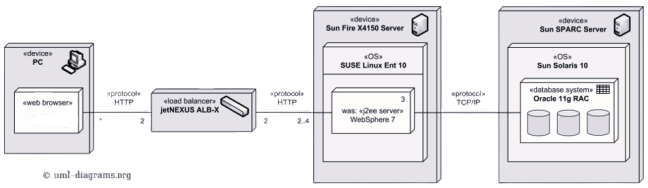


Figure 4.7: UML deployment diagrams and deployment structure

Summary

Structures	Elements	Relations	Useful for
Component diagrams	Components offering services	Provided and required interfaces	Performance analysis Robustness analysis Resource allocation Project planning
Sequence diagrams	Processes Threads	Synchronous and asynchronous messages	Analysis of resource contention Parallelism opportunities

Table 4.1: Component and connectors diagrams and usage

Structures	Elements	Relations	Useful for
Composite structures Package diagrams	Modules Packages	Is a submodule of Uses	Resource allocation Project planning Encapsulation
Class diagrams	Classes	Is-a Is part of	Object-oriented development Planning for extensions
Layered structures	-	Can use Provides abstraction	Incremental development
Data model	Data entities	One-to-one One-to-many Many-to-one Many-to-many Is-a	Engineering global data structures for consistency and performance

Table 4.2: Modular structures diagrams and usage

Structures	Elements	Relations	Useful for
Deployment diagrams	Components Hardware/software execution environment	Allocated to	Performance Security Robustness analysis
Implementation structures	Modules File structures	Stored in	Configuration control Integration Test activities
Work assignment	Modules Organizational units	Assigned to	Project management Development efficiency

Table 4.3: Allocation structures diagrams and usage

4.2 Software design description and principles

The IEEE has established two standards pertaining to architectural models:

- IEEE standard for information technology: software design descriptions.

- IEEE standard for systems and software engineering: architecture description (manner in which architectural descriptions of systems are organized and expressed).

In accordance with IEEE standards, a Software Design Description (SDD) should encompass the following elements:

- Identification of the SDD, which includes information such as the date, authors, and affiliated organization.
- Description of design stakeholders.
- Description of design concerns.
- Selection of design viewpoints.
- Design views.
- Design overlays.
- Design rationale.

Additionally, there are eleven key design principles that should be considered in the development of software systems:

1. Employ the divide and conquer approach: break down complex problems into smaller, more manageable components.
2. Maintain a high level of abstraction in designs to minimize consideration of unnecessary details and reduce complexity.
3. Maximize cohesion where possible to ensure logical and functional consistency within components.
4. Minimize coupling to reduce interdependencies between system elements.
5. Design for reusability, allowing various aspects of the system to be used in different contexts.
6. Promote the reuse of existing designs and code, complementing the design for reusability.
7. Design for flexibility by anticipating and preparing for future changes in system requirements.
8. Anticipate obsolescence by planning for technological or environmental changes to ensure software continuity or ease of adaptation.
9. Design for portability, aiming to make the software compatible with a wide range of platforms.
10. Design for testability, implementing measures that facilitate testing processes.
11. Design defensively, exercising caution when making assumptions about how others will interact with the components you are designing.

4.3 Architectural styles

Definition. 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.

Client-server

The client-server architectural style is primarily employed in distributed applications, where the client initiates requests, and the server delivers responses.

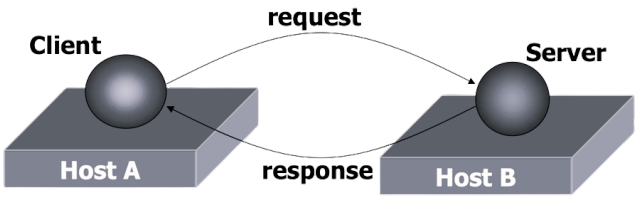
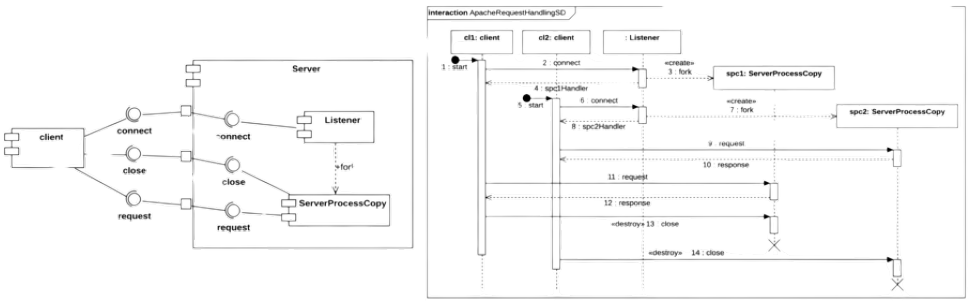


Figure 4.8: Client-server architecture general schema

This approach proves beneficial in situations where multiple users require access to a common resource, when there is a need to remotely access existing software, and when it is advantageous to structure the system around a shared functionality that multiple components utilize. Key technical considerations for this architectural style include the design and documentation of well-defined interfaces for the server and the need to ensure the server's capability to handle concurrent requests efficiently.

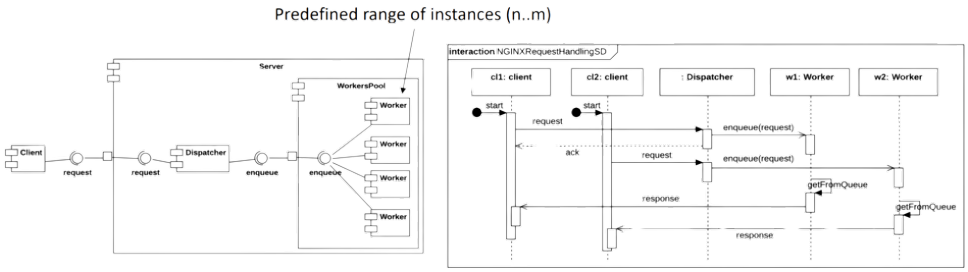
In this architectural context, the requirement is to receive and manage requests from multiple clients. Various solutions can be considered to address this challenge:

- Forking (Apache Web Server):
  - Approach: create one process per request or per client.
  - Strengths: simplicity, isolation, and protection provided by the one connection per process model. Effective for up to 2000 users.
  - Issues: limited capacity, uncertainty regarding the number of active processes at a given time, and scalability challenges due to the overhead of fork-kill operations.



- Worker pooling (NGINX Web Server):

- Approach: specifically designed for high concurrency, addressing scalability issues inherent in the forking model by introducing a new architectural tactic[1].
- Architectural Tactic: NGINX prioritizes scalability and performance at the expense of availability.
- Strengths: fixed number of workers for each pool, each worker equipped with a queue (with dropped requests for performance if the queue is full), and a dispatcher for load balancing among workers.



Three-tier architecture

Within three-tier architectures, the intermediate tier is responsible for handling the application logic. The primary advantage lies in the separation of logic from both data and presentation elements. The modular structure allows for enhanced flexibility and maintainability.

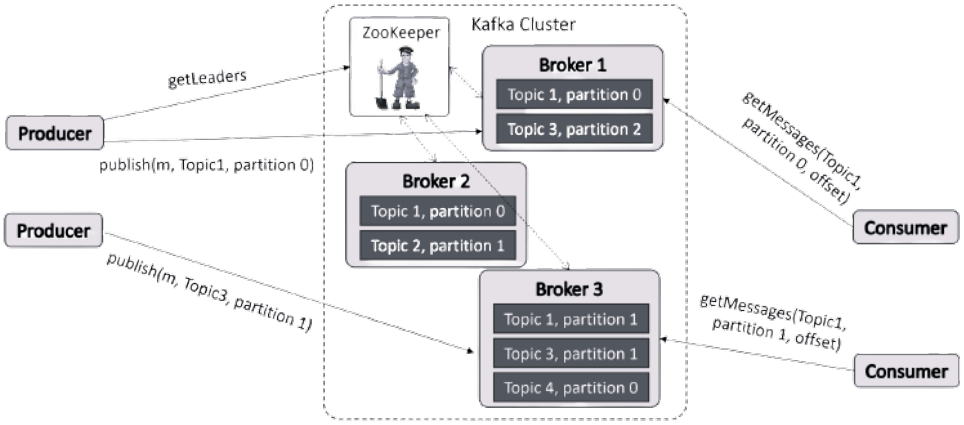
With the option of incorporating additional tiers, it becomes feasible to establish an N-tier architecture. This extended architecture enables further refinement and specialization of components, supporting more intricate and scalable system designs.

Event-driven architecture

In an event-driven architecture, components can register to both send and receive events that are broadcasted to all registered components, where neither the sender nor the receiver is known. This model, often referred to as publish-subscribe, facilitates easy addition and deletion of components and is increasingly employed in modern integration strategies. Key characteristics include asynchronous events, reactive computation, message destination determined by the receiver, loose coupling, and flexible communication means. Strengths and challenges of this architecture include its widespread adoption in modern development practices, ease of adding and removing components, potential scalability issues, and complexities in event ordering.

Apache Kafka serves as a framework for the event-driven paradigm, providing primitives for creating event producers and consumers, along with a runtime infrastructure for handling event transfer. Kafka stores events durably and reliably, allowing consumers to process events in real-time or retrospectively. These services are delivered in a distributed, highly scalable, elastic, fault-tolerant, and secure manner.

[1]Design decisions that influence the control of one or more quality attributes.



The Kafka architecture involves brokers managing designated topics and partitions, with each partition containing sets of messages related to the respective topic. The partitions operate independently and can be duplicated across multiple brokers to ensure fault tolerance.

Each partition designates one broker as the leader, while the remaining brokers hosting the same partition act as followers. Producers are aware of the leading brokers and direct their messages to them. Messages within a given topic are grouped into batches on the producer's end and transmitted to the broker once the batch size surpasses a specified threshold.

Consumers employ a pull mechanism, retrieving an entire batch of messages associated with a specific partition from a defined offset. Messages persist at the broker level for a predetermined duration and can be accessed multiple times during this period.

The leader broker keeps tabs on the in-sync followers. The correct functioning of the cluster is overseen by ZooKeeper, with all brokers regularly sending heartbeats to it. In the event of a broker failure, ZooKeeper intervenes by appointing a new leader for all partitions previously led by the failing broker. Additionally, ZooKeeper has the capability to initiate or restart brokers as needed.

Brokers finalize message commitment by storing them in their designated partition. The leader includes the message in the followers (replicas) if they are accessible. A potential challenge arises in the event of failure, where the producer may not receive a response, as illustrated by message seven. Consequently, the producer is required to resend the message.

Kafka brokers possess the ability to recognize and eliminate duplicate messages. The synchronization process with replicas can be transactional, allowing for precise control over the message flow. Achieving exactly-once semantics is feasible but comes with the trade-off of longer waiting times. Alternatively, opting for at-least-once semantics involves excluding duplicate management, and at-most-once semantics can be chosen by dispatching messages asynchronously.

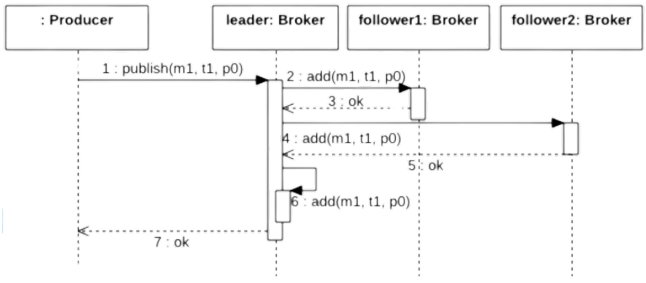


Figure 4.9: Example of a producer

Every consumer has access to a persistent log to maintain the offset, ensuring it is preserved

in the event of a failure. If a consumer encounters a failure after processing messages but before recording the new offset in the log, it will retrieve the same messages again, resulting in at-least-once semantics.

The delivery semantics can be altered by storing the new offset before processing. In this case, if a failure occurs after storing the offset, the impact of the received messages does not materialize, leading to at-most-once semantics.

Transactional management of the log introduces the capability for exactly-once semantics, providing a robust mechanism for ensuring that each message is processed only once, even in the face of failures or system disruptions.

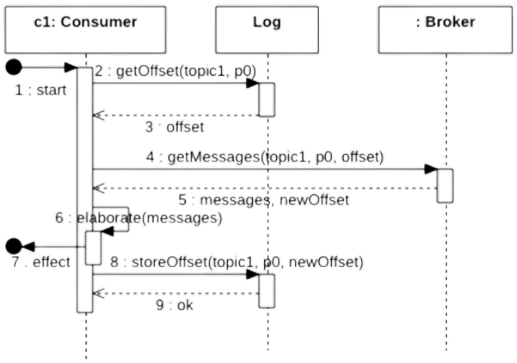


Figure 4.10: Example of a consumer

Apache Kafka's architectural tactics include scalability through multiple partitions and brokers and fault tolerance through persistent partition storage and data replication.

Microservices

In the era preceding microservices, monolithic systems delivered applications as single deployable software artifacts. The microservice architectural style emerged as an approach to developing applications by decomposing monolithic systems into small, specialized services, each running independently in its own process. These services communicate using lightweight mechanisms, often employing HTTP resource APIs.

Microservices focus on addressing a single bounded context within the target domain, offering advantages such as fine-grained scaling strategies, reduced impact of localized issues, improved resilience through decentralized functionality, and enhanced reuse and composability.

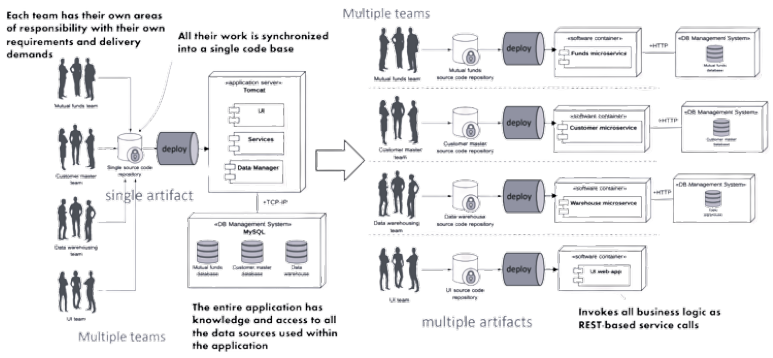


Figure 4.11: Typical microservices architecture process

The process of creating a microservices architecture involves reducing teams' synchronization overhead, organizing small development teams with well-defined responsibilities, and employing technology-neutral protocols like REST APIs for communication, resulting in smaller codebases for easier debugging and more cost-effective maintenance.

The architecture's main elements include a REST API that exposes core service operations, the application or business logic responsible for implementing core operations, and local data storage for each microservice, emphasizing limited data sharing and avoiding global databases.

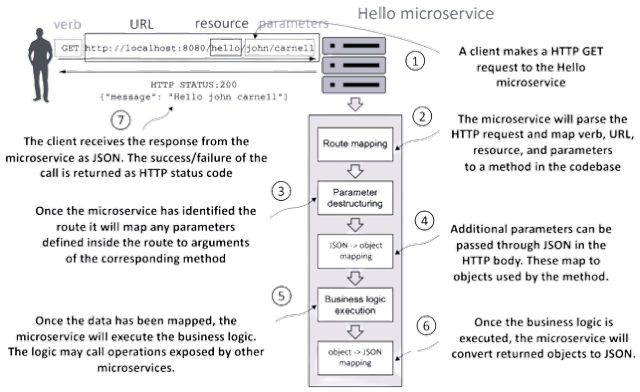


Figure 4.12: Microservices architecture's workflow

To achieve location transparency and scalability in microservices, routing patterns are essential. Service discovery, a critical element, must be highly available, load-balanced, resilient, and fault-tolerant. Resilience patterns, such as the circuit breaker pattern, aim to prevent resource waste and ripple effects by actively monitoring calls to distant services and intervening in case of potential failures.

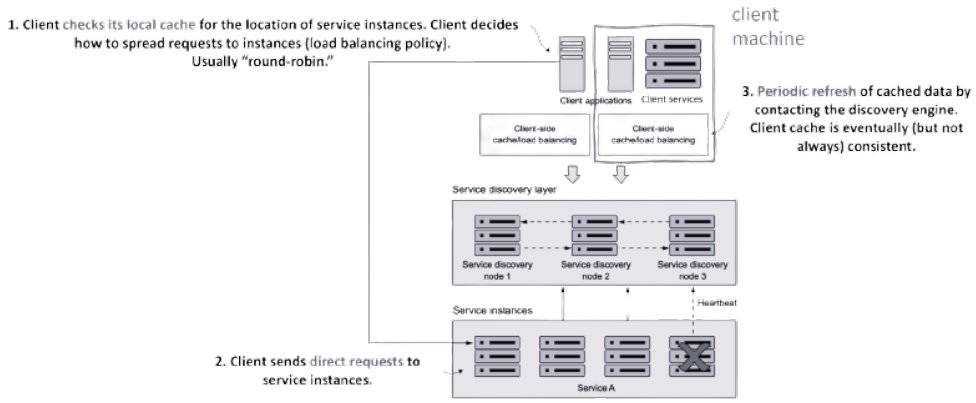


Figure 4.13: Routing pattern

To allow the microservices to be resilient we need resiliency patterns. The main goal of these patterns are to avoid useless resource consumption and ripple effects. Circuit breaker is a client-side resiliency pattern. The circuit breaker functions as a proxy for a distant service. As it invokes the remote service, the CB actively monitors the call for potential failures, which may include receiving a 5xx error or the call taking an excessively long duration, prompting the CB to terminate the call. In the event of "too many" failures, the circuit breaker intervenes by inhibiting future calls to prevent further issues.

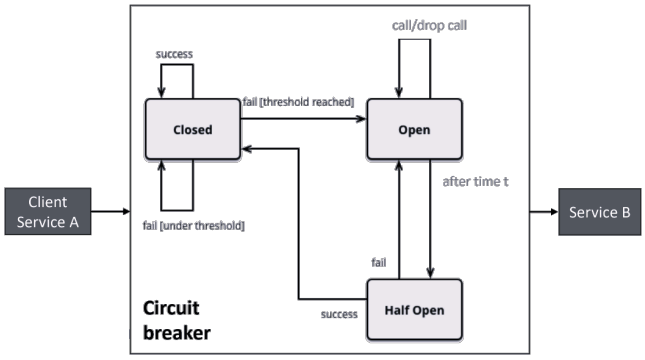


Figure 4.14: Circuit breaker pattern

Security patterns in microservices involve introducing a service or API gateway that acts as a mediator between service clients and invoked services. While this gateway simplifies the implementation of authentication and authorization mechanisms, it can become a single point of failure. This challenge is mitigated by deploying multiple instances of the gateway with a server-side load balancer.

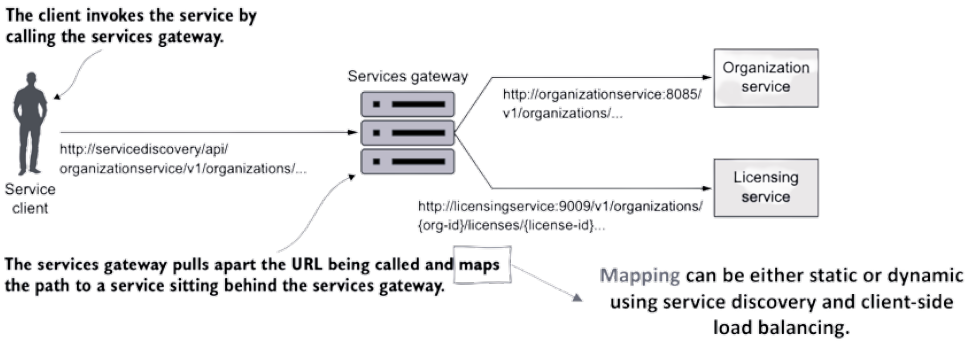


Figure 4.15: Security pattern

Communication patterns play a crucial role in achieving loose coupling in microservices. Despite their strengths in providing flexibility, scalability, and availability, they introduce higher complexity. Event-driven frameworks, supporting multiple communication styles such as notification, request/response, and publish/subscribe, are employed to decouple different parts of the system.

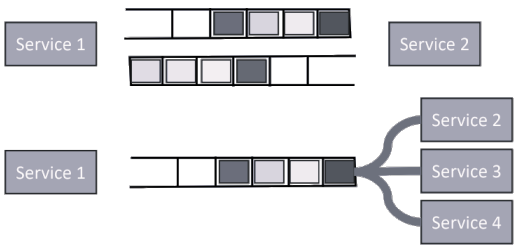


Figure 4.16: Security pattern

4.4 Interface design and documentation

Definition. An *interface* is a boundary where components interact.

The precise definition of interfaces carries significant architectural implications as it directly affects maintainability, usability, testability, performance, and integrability. Two fundamental guiding principles are information hiding and low coupling. The key aspects to consider when designing interfaces are as follows:

- Contract principle: any addition of a resource (operation or data) to an interface signifies a commitment to maintaining it.
- Least surprise principle: interfaces should behave consistently with expectations.
- Small interfaces principle: interfaces should expose the minimum necessary resources.

When designing interfaces, we have also to define:

- Interaction style:
  - Socket: after connection establishment, communication is bidirectional, and both parties must agree on the same protocol.
  - RPC/RMI: resembles a procedure/method call in a centralized setting. It requires stubs and skeletons to transform procedure/method calls into messages and vice versa.
  - REST (REpresentational State Transfer): this is a standardized architectural style for Application Programming Interfaces (APIs) that emphasizes clear separation between distributed, heterogeneous systems and their components.
- Representation and structure of exchanged data: the choice of data representation impacts expressiveness, interoperability, performance, and transparency. Common representations include XML (verbose, requires multiple parsing passes), JSON (more compact than XML, single-pass parsing), and Protocol Buffers (most compact, data passed as binary).
- Error handling: possible reactions to errors include raising an exception, returning an error code, or logging the problem.

A server can offer multiple interfaces concurrently, enabling the separation of concerns, different access rights levels, and support for interface evolution. Interfaces constitute the contract between servers and clients, and sometimes interfaces need to evolve. The strategies used to support this continuity are:

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

Representational state transfer

REST APIs are simple and standardized, alleviating developers from concerns about communication protocols (HTTP), data formatting (JSON), and request/response encoding. REST is stateless, which means it doesn't retain states across servers and clients, making it lightweight, scalable, and supportive of caching for high performance.

The available CRUD requests are:

- Create (HTTP POST): used to create a new resource.
- Read (HTTP GET): used to retrieve an existing resource.
- Update (HTTP PUT): used to modify a resource.
- Delete (HTTP DELETE): used to remove a resource.

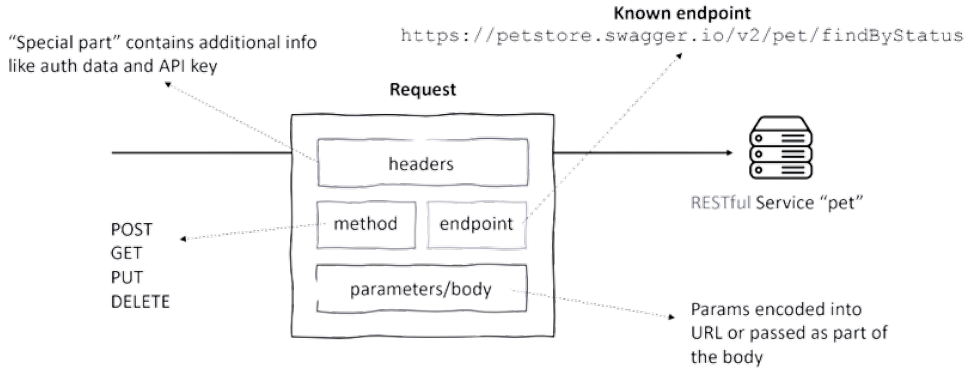


Figure 4.17: Structure of a request

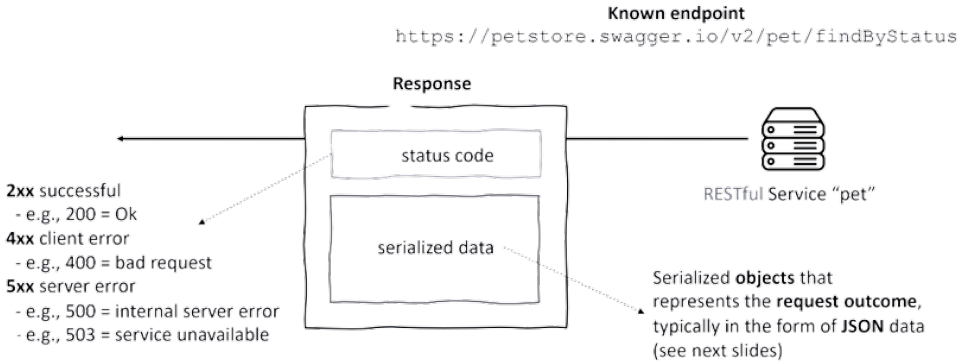


Figure 4.18: Structure of a response

Interface documentation

Interface documentation explains how to use an interface without revealing details of the component's internals. The audience for interface documentation includes developers and maintainers of both the offering and using components, quality assurance teams for system integration and testing, and software architects seeking reusable components.

To document a REST API, the OpenAPI specification is used. It describes a REST API interface through an OpenAPI definition, typically in JSON or YAML format. Benefits of this

type of documentation include its standardized format, public accessibility, and suitability for both human understanding and machine automation tasks like testing and code generation.

The OpenAPI definition covers endpoints, resources, operations, parameters (including data types), and authentication/authorization mechanisms. Support tools for OpenAPI documentation include API validators, documentation generators, and SDK generators for automated client library creation in various programming languages.

4.5 Software qualities and architectures

Numerous software qualities are directly shaped by architectural decisions, necessitating metrics to quantify these qualities and methodologies to analyze the quantitative impact of architectural choices on them.

Availability

Continuous availability of a service is imperative to ensure minimal downtime and rapid service recovery. The service's availability depends on several factors, including:

- Complexity of IT infrastructure architecture.
- Reliability of 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.

Definition. The *time of occurrence* is the time at which the user becomes aware of the failure.

The *detection time* is the time at which operators become aware of the failure.

The *response time* is the time required by operators to diagnose the issue and respond to users.

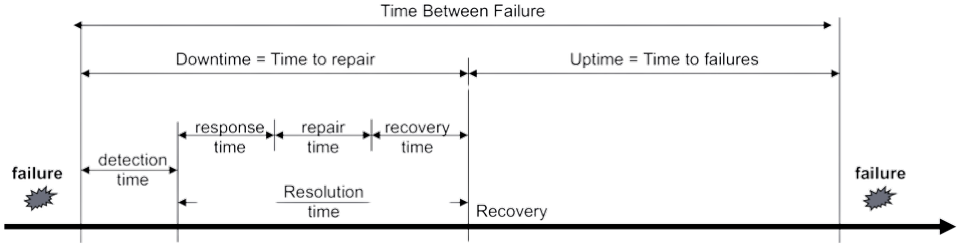
The *repair time* is the time required to fix the service or components that caused the failure.

The *recovery time* is the time required to restore the system.

The *mean time to repair* (MTTR) is the average time between the occurrence of a failure and service recovery, also known as the downtime.

The *mean time to failures* (MTTF) is the average time between the recovery from one failure and the occurrence of the next failure, also known as uptime.

The *mean time between failures* (MTBF) is the average time between the occurrences of two consecutive failures.



Definition. The *availability metric* is a probability that a component is working properly at time t, defined as:

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

Availability is typically specified in nine notation (indicating the number of consecutive nines in the percentage).

Availability is calculated by modeling the system as an interconnection of elements in series and parallel:

- Elements operating in series. If an element in the series fails, it results in the failure of the entire combination.

  $$A_{\text{series}} = \prod_{i=1}^{n} A_i$$

  A chain's strength is determined by its weakest link.

- Elements operating in parallel. In this scenario, the failure of one element prompts the other elements to assume the operations of the failed component.

  $$A_{\text{parallel}} = 1 - \prod_{i=1}^{n} (1 - A_i)$$

  Critical systems are often designed with redundant components to enhance reliability and availability.

The main techniques employed to enhance availability include replication, forward error recovery, and circuit breakers.

Replication can be done in multiple ways:

- Hot spare: one leading server with another always ready to take over.
- Warm spare: leading server periodically updates another; if the leading one fails, some time may be needed to fully update the backup.
- Cold spare: backup server is dormant and started/updated only when needed.
- Triple modular redundancy: multiple active servers, and the produced result is based on the majority.

For the forward error recovery we have that in the failure state, a recovery mechanism moves the component to the degraded state. In the degraded state, the component continues to be available even if not fully functional.

4.6 Spring framework

Definition (*Framework*)**.** A software framework is an integrated collection of components, set of applications, conventions, principles and common practices for design and development of software

Spring stands out as one of the most widely embraced Java-based frameworks for constructing distributed software systems. This framework adeptly manages the underlying infrastructure, enabling developers to concentrate on crafting application logic. Its robust support from the community adds to its popularity.

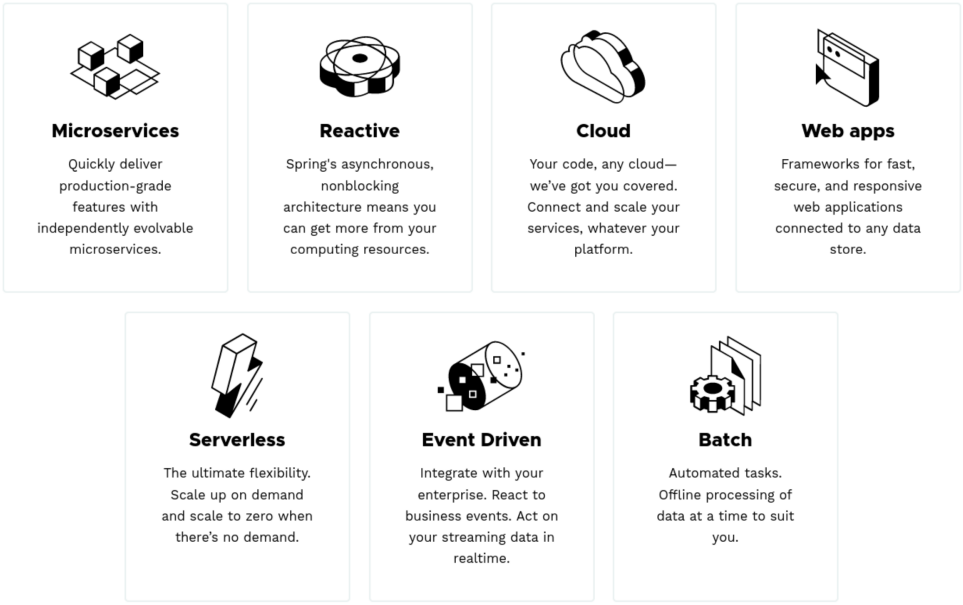


Figure 4.19: Spring framework functionalities

CHAPTER 5

Verification and validation

5.1 Quality assurance

Verification focuses on internal characteristics, ensuring that the software is built correctly, while validation concerns external characteristics, ensuring that the right software is being built.

Definition (*Quality assurance*)**.** Quality assurance establishes policies and processes to have quality in software development.

Quality encompasses various aspects:

- Elimination of defects or bugs.
- Resolution of issues hindering the fulfillment of non-functional requirements or causing degradation of software qualities.
- External qualities such as performance and availability.
- Internal qualities like maintainability.

Definition (*Failure*)**.** Failure is the termination of a product's ability to perform a required function or its incapacity to operate within predefined limits. It is an event where a system or its component fails to fulfill a specified function.

Definition (*Fault*)**.** A fault is the observable manifestation of a defect.

Definition (*Defect*)**.** A defect is an imperfection or deficiency in a program.

Definition (*Error*)**.** An error results from a human action introducing an incorrect result.

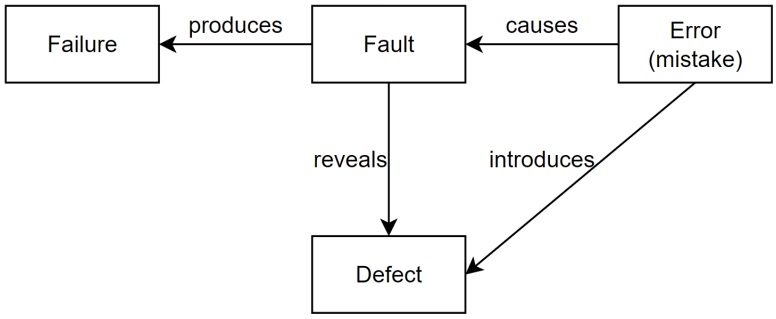
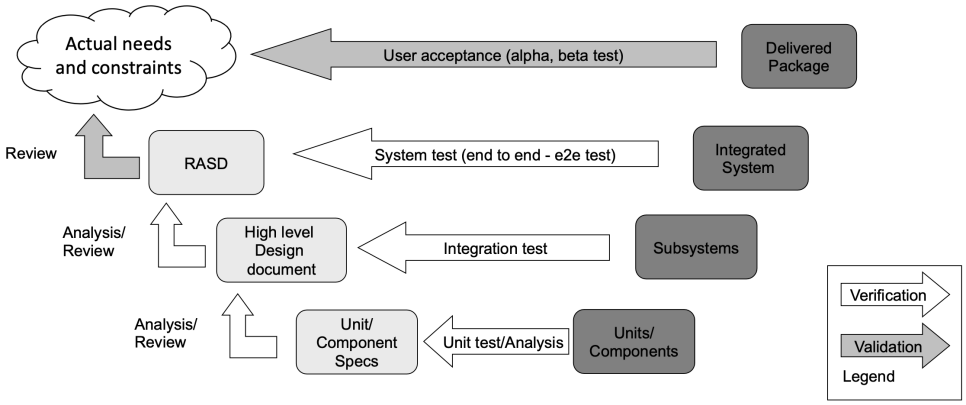


Figure 5.1: Failure, fault, defect, and errors

Achieving zero-defect software is practically impossible, necessitating careful and continuous quality assurance. Ideally, every artifact should undergo quality assurance, including verification artifacts, which must themselves be verified.

Structural engineering and software engineering In structural engineering, when tasked with constructing a bridge intended to support heavy trucks (40 tons), a straightforward approach involves subjecting the bridge to a load test with, for instance, 50 tons. Remarkably, this single test scenario effectively covers a spectrum of potential cases.
On the contrary, in software engineering, where the goal is to develop a program, the absence of continuous behavior complicates the verification process. Testing a program with a single data point provides limited insights into its performance at other points. Consequently, the choice of verification techniques varies depending on the level of implementation, as illustrated below:



Possible approaches There are two primary approaches for testing a program:

- *Static analysis*: this analysis occurs directly on the program's source code without execution. It's important to note that the properties examined are dynamic.
- *Dynamic analysis*: also known as testing, dynamic analysis involves executing the program's sources, typically through sampling. During this analysis, the actual behavior of the program is assessed in comparison to the expected behavior.

Static analysis	Dynamic analysis
At compile time	At runtime
Related to source code	Related to software behavior
Without execution of the software	While executing the software
On generic (or symbolic) inputs	On specific inputs

5.2 Static analysis

The concept behind static analysis involves examining the source code using analyzers, each tailored to assess a predetermined set of hard-coded (non-custom) properties in a fully automated manner. The output is deemed safe when no issues are detected and unsafe otherwise. The properties scrutinized typically revolve around general safety considerations, aiming to prevent conditions that could lead to errors. Commonly desired properties include:

- Absence of integer variable overflow.
- No type errors.
- Prevention of null-pointer dereferencing.
- Avoidance of out-of-bound array accesses.
- Elimination of race conditions.
- Prohibition of useless assignments.
- Prevention of the use of undefined variables.

Definition (*Program behavior*)**.** Program behavior encompasses the entire set of possible executions represented as sequences of states.

While static analysis is adept at identifying erroneous states, it tends to be pessimistic as the program behavior might not reach any of these states.

Precision and efficiency Static analysis relies on over-approximations to ensure soundness. Achieving perfect precision is often unattainable due to undecidability. The trade-off between precision and efficiency is a critical consideration, as high precision may be computationally expensive, while low precision leads to more false positives that necessitate manual verification. Designing a static analysis technique involves striking a practical balance between precision and efficiency.

5.2.1 Data-flow analysis

Control-flow graph The control-flow graph (CFG) is a directed graph that depicts potential execution paths in a program. Each node within the graph represents a program statement, and consecutive statements are connected by edges. Declarations are disregarded as they do not impact the program state.

Data-flow analysis Data-flow analysis operates on the control-flow graph (CFG) of a program to extract information about the data flow, specifically checking which values are read (used) and written (defined). The analysis of these properties involves examining the output.

Definition (*Live variable*)**.** In the context of a CFG, a variable v is considered live at the exit of a block b if there exists a path on the CFG from block b to a use of v that does not redefine v.

Live variable analysis determines, for each block, the variables that may be live at the block's exit. It's crucial to note that the phrase may be live constitutes an over-approximation. Thus, $\mathrm{LV}(k)$ represents a superset of the live variables at k:

- If $x \notin \mathrm{LV}(k)$, then x is definitely not live at k.
- If $x \in \mathrm{LV}(k)$, then x may not be live at k.

The live variable analysis is translated into an equation system, and solving this system identifies the live variables. Standard algorithms such as fixed-point or worklist can be employed to automatically compute the solution. Data-flow analysis is effective in eliminating dead assignments; if a variable is not live after being defined by an assignment, that assignment is redundant and can be safely removed without altering the program's behavior.

Reaching definition analysis Reaching definition analysis determines, for each block, the definitions that may reach the block.

Definition (*Definition*)**.** A definition (v, k) represents an assignment to variable v occurring at block k.

Definition (*Block reachability*)**.** A definition (v, k) reaches block r if there exists a path from k to r that does not redefine v.

For this type of analysis, the following equations can be defined for each block k:

$$\mathrm{RD}_{\mathrm{IN}}(k) = \bigcup_{h \to k} \mathrm{RD}_{\mathrm{OUT}}(h)$$

$$\mathrm{RD}_{\mathrm{OUT}}(k) = (\mathrm{RD}_{\mathrm{IN}}(k) - \mathrm{kill}_{\mathrm{RD}}(k)) \cup \mathrm{Gen}_{\mathrm{RD}}(k)$$

Reaching definition analysis provides information about which statements define values and which use them, making it valuable for program optimizations or error avoidance.

Definition (*Use-definition chain*)**.** A use-definition chain, denoted as UD, links a use to all definitions that may reach it:

$$\mathrm{UD}(v, k) = \{q | \text{"}q : v := E\text{" and def\_clear}(v, q, k)\} \cup \{?|\mathrm{def\_clear}(v, ?, k)\}$$

Here, "q:v:=E" is an assignment for v at line q. $\mathrm{def\_clear}(v, q, k)$ holds if and only if there is a definition-clear path from q to k (i.e., no block between q and k redefines v).

Definition (*Use-definition path*)**.** A use-definition path is a path from q to k such that k uses some v and $q \in \mathrm{UD}(v, k)$.

The set $\mathrm{UD}(v, k)$ can be computed from reaching definitions analysis as follows:

$$\begin{cases} \{q | (v, q) \in \mathrm{RD}_{\mathrm{IN}}(k)\} \text{ if } v \text{ uses } d \text{ in block } k \\ \{\ \} \text{ otherwise} \end{cases}$$

Definition (*Definition-use chain*)**.** A definition-use chain, denoted as DU, links a definition to all use such that the definition may reach them:

$$\mathrm{DU}(v, k) = \{q | q \text{ uses } v \text{ and } (v, k) \text{ reaches } q\}$$

Definition (*Definition-use path*)**.** A definition-use path is a path from k to q such that k defines some v and $q \in \mathrm{DU}(v, k)$.

The set $\mathrm{DU}(v, k)$ can be computed as the inverse of use definition:

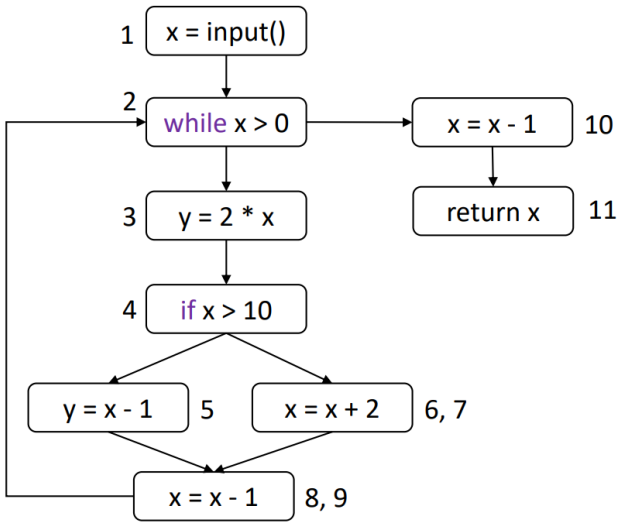
$$\mathrm{DU}(v, k) = \{q | k \in \mathrm{UD}(v, q)\}$$

Definition (*Definition-use pair*)**.** Given $\mathrm{DU}(v, k)$, if some $q \in \mathrm{DU}(v, k)$, we say that $\langle k, q \rangle$ is a definition-use pair for v.

Example:
Let's examine the given code snippet:

```
int foo() {
    x = input();
    while (x > 0) {
        y = 2 * x;
        if (x > 10)
            y = x - 1;
        else
            x = x + 2;
            x = x - 1;
    }
    x = x - 1;
    return x;
}
```

We initiate the analysis by constructing the control flow graph (CFG) of the program:



Utilizing the CFG, we conduct live variable analysis, resulting in:

- $\mathrm{LV}(1) = \{x\}$.
- $\mathrm{LV}(2) = \{x\}$.
- $\mathrm{LV}(3) = \{x\}$, there is a problem since y is defined here and not live.
- $\mathrm{LV}(4) = \{x\}$.
- $\mathrm{LV}(5) = \{x\}$, there is a problem since y is defined here and not live.
- $\mathrm{LV}(6,7) = \mathrm{LV}(8,9) = \mathrm{LV}(10) = \{x\}$.
- $\mathrm{LV}(11) = \{\,\}$.

Using the CFG, we also establish definition-use pairs:

- $\mathrm{RD}_{\mathrm{IN}}(1) = \{(x,?),(y,?)\}$.
- $\mathrm{RD}_{\mathrm{IN}}(2) = \mathrm{RD}_{\mathrm{OUT}}(1) \cup \mathrm{RD}_{\mathrm{OUT}}(8,9) = \{(x,1),(y,?),(x,8),(y,5),(y,3)\}$.
- $\mathrm{RD}_{\mathrm{IN}}(3) = \mathrm{RD}_{\mathrm{OUT}}(2) = \{(x,1),(y,?),(x,8),(y,5),(y,3)\}$.
- $\mathrm{RD}_{\mathrm{IN}}(4) = \mathrm{RD}_{\mathrm{OUT}}(3) = \{(x,1),(x,8),(y,3)\}$.
- $\mathrm{RD}_{\mathrm{IN}}(5) = \mathrm{RD}_{\mathrm{OUT}}(4) = \{(x,1),(x,8),(y,3)\}$.
- $\mathrm{RD}_{\mathrm{IN}}(6,7) = \mathrm{RD}_{\mathrm{OUT}}(4) = \{(x,1),(x,8),(y,3)\}$.
- $\mathrm{RD}_{\mathrm{IN}}(8,9) = \mathrm{RD}_{\mathrm{OUT}}(5) \cup \mathrm{RD}_{\mathrm{OUT}}(6,7) = \{(x,1),(y,5),(x,7),(x,8),(y,3)\}$.
- $\mathrm{RD}_{\mathrm{IN}}(10) = \mathrm{RD}_{\mathrm{OUT}}(2) = \{(x,1),(y,?),(x,8),(y,5),(y,3)\}$.
- $\mathrm{RD}_{\mathrm{IN}}(11) = \mathrm{RD}_{\mathrm{OUT}}(10) = \{(x,10),(y,?),(y,5),(y,3)\}$.

Based on these definitions, we establish the used definition chains and the definition-use pairs. We observe that x is defined in 1, 7, 8, 10 and used in 2, 3, 4, 5, 7, 8, 10, 11, and y is defined in 3, 5 but not used. The use definition chains are defined as follows:

1. $\text{UD}(x, 2) = UD(x, 3) = UD(x, 4) = UD(x, 5) = UD(x, 7) = UD(x, 10) = \{1, 8\}$
2. $\text{UD}(x, 8) = \{1, 7, 8\}$
3. $\text{UD}(x, 11) = \{10\}$

Consequently, the definition-use pairs for x are:

1. $\langle 1, 2\rangle$, $\langle 1, 3\rangle$, $\langle 1, 4\rangle$, $\langle 1, 5\rangle$, $\langle 1, 7\rangle$, $\langle 1, 8\rangle$, and $\langle 1, 10\rangle$.
2. $\langle 7, 8\rangle$.
3. $\langle 8, 2\rangle$, $\langle 8, 3\rangle$, $\langle 8, 4\rangle$, $\langle 8, 5\rangle$, $\langle 8, 7\rangle$, $\langle 8, 8\rangle$, and $\langle 8, 10\rangle$.
4. $\langle 10, 11\rangle$.

5.2.2 Symbolic execution

Symbolic execution is an approach that scrutinizes source code to assess specific properties. The key properties examined include:

- *Reachability*: this involves determining whether a particular location l in the code S is reachable in any execution. The method aims to verify that l is unreachable in some executions or identifies the conditions under which l becomes reachable.
- *Path feasibility*: this property involves checking whether a specific path p in the code S is entirely reachable. The objective is to confirm that the execution of p is not possible, or alternatively, identify the conditions under which p can be executed.

Symbolic execution can be employed for automatic test case generation. However, it may fall short of identifying all possible paths. The process involves executing programs with symbolic values, and symbolic states keep track of the symbolic values of variables. The potential outcomes of symbolic execution are:

- SAT exit (π is satisfiable): indicates that there exists a satisfying assignment to variables in π and such an assignment represents an input that satisfies the specified property in a concrete execution.
- UNSAT exit (π is not satisfiable): denotes that the given property cannot be satisfied by any concrete execution.

Execution paths can be organized into an execution tree, with final states marked as SAT or UNSAT.

Example:
Consider the provided code snippet:

```
int foo(int x, int y) {
    int z := x
    if (z < y)
        z := z*2
    if (x < y && z >= y)
        print(z)
}
```

In line zero, two variables, x and y, are defined. Consequently, we initialize a table with these variables having symbolic values X and Y, respectively. Line zero is added to the path as it is always executed.

Symbolic state	x	y	$\langle 0 \rangle$
Symbolic values	X	Y	

In line one, a new variable z is defined with the same value as x. We add a state z with a value of X and include line one in the path as it is always reachable.

Symbolic state	x	y	z	$\langle 0, 1 \rangle$
Symbolic values	X	Y	X	

In line two, a conditional expression is encountered, which is always checked. Thus, we add 2 to the path and introduce a state π to represent the condition. The table becomes duplicated based on the condition:

Symbolic state	x	y	z	π	$\langle 0, 1, 2 \rangle$
Symbolic values	X	Y	X	$X < Y$	

Symbolic state	x	y	z	π	$\langle 0, 1, 2 \rangle$
Symbolic values	X	Y	X	$X \geq Y$	

Since symbolic execution prioritizes feasible paths, we select the first table. In line three, the value of z is redefined, and the path now reaches line three:

Symbolic state	x	y	z	π	$\langle 0, 1, 2, 3 \rangle$
Symbolic values	X	Y	$2X$	$X < Y$	

Another condition on the π column is added, resulting in two tables:

Symbolic state	x	y	z	π	$\langle 0, 1, 2, 3, 4 \rangle$
Symbolic values	X	Y	$2X$	$X < Y$ $X < Y \wedge 2X \geq Y$	

Symbolic state	x	y	z	π	$\langle 0, 1, 2, 3, 4 \rangle$
Symbolic values	X	Y	$2X$	$X < Y$ $X < Y \wedge 2X \geq Y$	

We again select the path that satisfies the condition, resulting in:

Symbolic state	x	y	z	π	$\langle 0, 1, 2, 3, 4, 5 \rangle$
Symbolic values	X	Y	$2X$	$X < Y$ $X < Y \wedge 2X \geq Y$	

We observe that location five is reachable by assigning the appropriate values to the variables ($X = 2$, $Y = 3$). Additionally, the path $\langle 0, 1, 2, 4, 5 \rangle$ is not feasible since there is no satisfying assignment.

Execution paths can be organized into an execution tree, with conclusive states marked as SAT (satisfiable) or UNSAT (unsatisfiable). While symbolic execution offers a promising approach for verifying program correctness, certain challenges exist:

- *Complex path conditions*: path conditions generated during symbolic execution may become overly intricate for constraint solvers. While solvers excel at handling linear constraints, their proficiency diminishes when faced with non-linear arithmetic, bit-wise operations, or string manipulation.
- *Handling large path spaces*: symbolic execution encounters difficulties when the number of paths to explore becomes substantial. Unbounded loops, in particular, introduce infinite sets of paths. Even if the path set is finite, examining all loop iterations can be impractical. A common strategy is to approximate the analysis by considering 0, 1, and 2 iterations as a practical rule of thumb.
- *External code and unavailability of sources*: the presence of external code, especially when the source code is not accessible, introduces uncertainty in the solver's understanding of behavior. This can pose challenges as the solver may lack crucial information to reason about the program accurately.

5.3 Dynamic analysis

Testing serves the purpose of analyzing the behavior of programs. In this context, properties are translated into executable oracles, serving as representations of anticipated outputs and desired conditions. The primary limitation of testing lies in its reliance on a finite set of test cases, rendering this form of verification non-exhaustive. Failures manifest with specific inputs that act as triggers, and the execution process is automated. It is crucial to recognize that the primary objective of testing is to uncover program failures. Other common goals include:

- Exercising different segments of a program to enhance coverage.
- Verifying the seamless interaction between components (integration testing).
- Facilitating fault localization and error removal (debugging).
- Ensuring that previously introduced bugs do not resurface (regression testing).

Test cases A test case comprises a collection of inputs, execution conditions, and a criterion for determining pass or fail outcomes. The typical sequence for executing a test case involves:

- *Setup*: establish the initial state of the program to meet specified execution conditions.
- *Execution*: execute the program using the provided inputs.
- *Teardown*: document the output, final state, and any failures identified based on the pass/fail criterion.

A testing suite, encompassing numerous test cases, can be employed for comprehensive evaluation. The specification of a test case outlines the requirements to be met by one or more actual test cases.

Unit testing Developers perform unit testing, focusing on examining isolated, small sections (units) of code. The definition of a unit is often influenced by the programming language in use. Unit testing serves to identify issues at an early stage, provides guidance in design, and enhances overall code coverage. However, a challenge in testing isolation arises due to potential dependencies between units. To address this, a viable solution involves simulating the absent units.

5.3.1 Integration testing

Integration testing is focused on validating the interaction and interfaces among components. The potential faults uncovered during integration testing include:

- Inconsistent interpretation of parameters.
- Violations of assumptions about domains.
- Side effects on parameters or resources.
- Nonfunctional properties.

Integration and testing plans are typically outlined in the Design Document. The build plan dictates the implementation order, while the test plan details the execution of integration testing. Ensuring the test plan aligns with the build plan is crucial. The most commonly employed strategies for integration testing are:

- *Big bang*: testing occurs only after integrating all modules together. This strategy has the advantage of not requiring stubs and needing fewer drivers and oracles. However, it has drawbacks such as minimal observability, challenging fault localization, reduced efficacy, and delayed feedback. The cost of repairing faults found in later stages is also high.
- *Iterative and incremental strategies*: Ttsts are run as soon as components are released. This approach is structured around the hierarchical system design, leading to the following techniques:
  - *Top-down strategy*: progressing from the top level in terms of use or include relation. Drivers use top-level interfaces, but the need for stubs at each step can be a challenge. As modules are ready (following the build plan) more functionality is testable. As modules become available following the build plan, more functionality becomes testable, with stubs gradually replaced.
  - *Bottom-up strategy*: starting from the leaves of the uses hierarchy, this strategy doesn't require stubs. However, more drivers are typically needed (one for each module, similar to unit testing). It may result in multiple working subsystems that are eventually integrated into the final one.
  - *Threads strategy*: integrating portions of several modules that collectively offer a user-visible function. This approach minimizes the need for drivers and stubs but introduces complexity to the integration plan.
  - *Critical modules strategy*: commencing with modules carrying the highest risk, this strategy may resemble a threaded process with specific priorities. The focus is on a risk-oriented process, treating integration and testing as proactive measures to identify issues early.

Strategy selection Structural strategies, such as bottom-up and top-down, offer simplicity. In contrast, thread and critical modules strategies enhance external visibility of progress, especially in complex systems. It is viable to combine different strategies based on the specific requirements. Top-down and bottom-up strategies are suitable for relatively small components and subsystems. On the other hand, combinations of thread and critical modules integration testing are often preferred when dealing with larger subsystems. The selection of strategies depends on the nature and scale of the system under consideration.

5.3.2 System testing

End-to-end testing of the entire integrated system is carried out by independent teams, with the testing environment closely resembling the production setup. This testing encompasses both functional and non-functional aspects. Key strategies employed in system end-to-end testing include:

- *Functional testing*: verifying if the software aligns with functional requirements, this involves using the software according to use cases outlined in the RASD and checking for fulfillment of requirements.
- *Performance testing*: identifying bottlenecks affecting response time, utilization, and throughput. It detects inefficient algorithms, hardware, or network issues and suggests optimizations. The system is loaded with the expected workload to ensure the measured performance aligns with expectations.
- *Load testing*: exposing issues like memory leaks, memory mismanagement, and buffer overflows, load testing determines the upper limits of system components and compares alternative architectural options. The system is tested under increasing workloads until it reaches its capacity, and the load is sustained over an extended period.
- *Stress testing*: ensuring graceful recovery after failure, stress testing aims to break the system under test by overwhelming its resources or by reducing available resources.

5.3.3 Test case generation

Our objective is to establish high-quality test sets, characterized by:

- Demonstrating a high probability of error detection.
- Covering an acceptable range of cases.
- Maintaining sustainability with a limited number of tests.

Test cases can be formulated manually or automatically generated through automated testing techniques, including:

- *Combinatorial testing*: enumerating all possible inputs based on a specified policy.
- *Concolic execution*: pseudo-random generation of inputs guided by symbolic path properties.
- *Fuzz testing (fuzzing)*: pseudo-random generation of inputs, encompassing invalid and unexpected inputs.
- *Search-based testing*: exploring the space of valid inputs to identify those improving specific metrics.

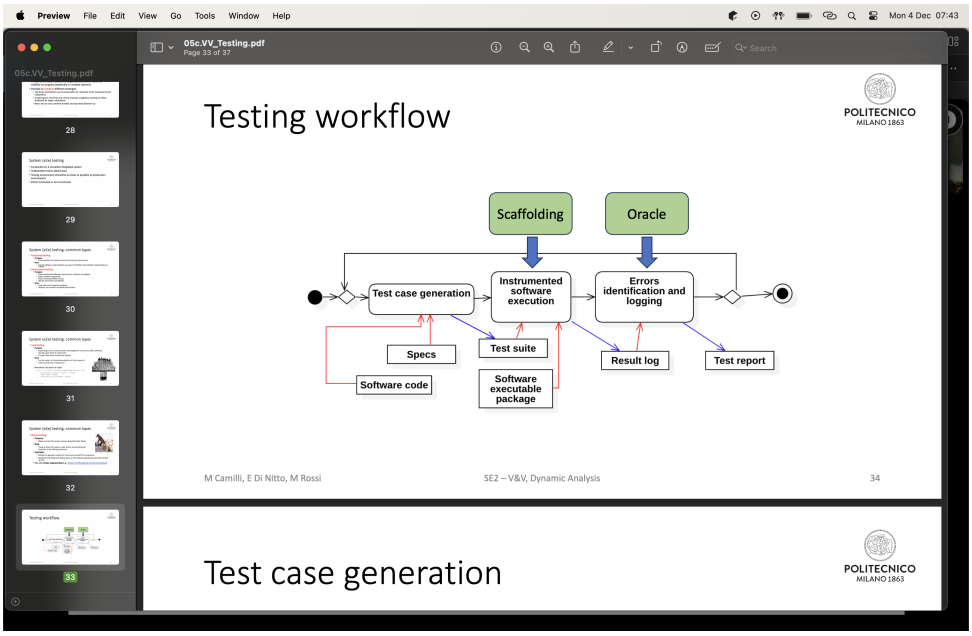


Figure 5.2: Testing workflow

Concrete symbolic execution Concolic execution involves performing symbolic execution alongside a concrete one, where the state combines a symbolic part and a concrete part. These components are used interchangeably to advance exploration. The technique involves two steps:

1. *Concrete to symbolic*: deriving conditions to explore new paths.
2. *Symbolic to concrete*: simplifying conditions to generate concrete inputs.

Example:

```
void m(int x, int y) {
    int z := 2 * y
    if (z == x) {
        z := y + 10
    if (x <= z)
        print(z)
    }
}
```

We initiate by considering line zero, assigning random variables to x and y:

Symbolic state	x	y	
Symbolic values	X	Y	$\langle 0 \rangle$
Example values	22	7	

Applying this method until the function returns, we obtain:

Symbolic state	x	y	z	π	
Symbolic values	X	Y	$2Y$	$2Y \neq X$	$\langle 0, 1, 2, 6, 7 \rangle$
Example values	22	7	14		

To explore another path, negate the path condition, obtaining the constraint $2Y = X$.

Reapplying the procedure with values satisfying the constraint, we obtain:

Symbolic state	x	y	z	π	
Symbolic values	X	Y	$2Y$	$2Y = X$ $X \leq Y + 10$	$\langle 0, 1, 2, 3, 4, 5, 6, 7 \rangle$
Example values	1	2	2		

To explore another path, we negate the second constraint while maintaining the preceding one. The final table in this case is:

Symbolic state	x	y	z	π	
Symbolic values	X	Y	$2Y$	$2Y = X$ $X > Y + 10$	$\langle 0, 1, 2, 3, 4, 6, 7 \rangle$
Example values	30	15	25		

The final result is that we explored three different paths with three sets of values:

- $\langle 0, 1, 2, 6, 7 \rangle$ with values $\{x = 22, y = 7\}$.
- $\langle 0, 1, 2, 3, 4, 5, 6, 7 \rangle$ with values $\{x = 2, y = 1\}$.
- $\langle 0, 1, 2, 3, 4, 6, 7 \rangle$ with values $\{x = 30, y = 15\}$.

The advantages of concolic execution are:

- *Handling black-box functions*: can handle black-box functions within path conditions, which is not achievable with pure symbolic execution.
- *Automatic test case generation*: capable of automatically generating concrete test cases based on a specified code coverage criterion.

The limitations are:

- *Fault occurrence limited to specific inputs*: faults may be associated with specific inputs, making it challenging to identify them if they are rare events. Concolic execution might not effectively highlight such issues.
- *Path explosion*: the number of paths tends to explode, especially in the presence of complex nested conditions, creating a vast search space. This can lead to challenges in scalability and efficiency.
- *Exploration guidance*: lacks a mechanism to guide the exploration systematically; it explores possible paths one by one within the allocated budget, potentially missing efficient exploration strategies.

Fuzzing Functional testing and fuzzing are complementary approaches in testing software components or systems, addressing not only correctness but also external qualities such as reliability and security. Fuzzing, a technique that operates at the component or system level, is particularly effective in uncovering defects that may elude other testing methods due to its use of random and often unexpected inputs. The core concept of fuzzing involves generating random inputs to observe their impact on the system.

Example:
Let's assume we want to fuzz an existing program, such as `bc`. `bc` is a UNIX utility serving as a basic calculator, dependent on a character stream denoting mathematical expressions for input. We assess the resilience of `bc` by applying an unpredictable input stream through the following steps:

- Develop a fuzzer: a program designed to produce a random character stream.
- Employ the fuzzer to test `bc` with the aim of identifying vulnerabilities or breaking points.

Definition (*Fuzz input*)**.** A fuzz input s characterized by its randomness and lack of structure.

Testing involves creating a substantial input file and assessing the program's behavior based on error messages, crashes, and the presence of illegal characters. Common errors detected through fuzzing include buffer overflows, missing error checks, and unexpected behavior caused by unconventional input values. To enhance the effectiveness of fuzzing, it is advisable to integrate it with runtime memory checks. Instrumentation at runtime monitors every memory operation, identifying potential violations such as out-of-bounds accesses, use-after-free, and double-free conditions. While this combination may introduce some performance overhead, it streamlines the developer's efforts in identifying and addressing issues.

However, a challenge arises when many programs expect inputs in specific formats. Random inputs might have a low probability of exploring deep paths within the program. To address this, mutational fuzzing can be employed, where existing valid inputs are mutated rather than generating entirely random inputs. This approach, guided by coverage information, focuses on evolving successful test cases. The fuzzer maintains a population of successful inputs, retaining and mutating those that lead to new paths, thereby improving the exploration of the program's functionality.

5.3.4 Search-based software testing

Search-Based Software Testing (SBST) complements existing test case generation methods and operates at both the component and system levels. Unlike traditional methods, SBST guides test case generation toward specific testing objectives, making it a goal-oriented approach. In contrast to fuzzing, SBST typically integrates domain-specific knowledge to produce more meaningful and context-aware test cases, addressing both functional and non-functional aspects such as reliability and safety. The underlying concept of SBST involves recasting testing as an optimization problem, aiming to generate test cases that fulfill defined testing objectives. The process involves iteratively improving test cases to achieve the desired objective. The key steps in SBST are as follows:

1. *Identify the objective*: clearly define the testing objective that needs to be achieved.
2. *Define distance measurement*: specify how to measure the distance of the current execution from the objective. This distance serves as the fitness metric, evaluating how well the current execution aligns with the testing goal. The current execution is represented by the inputs, i.e., the test case.
3. *Instrument code for fitness computation*: modify the code to compute the fitness (distance) of the current execution (test case) concerning the defined objective.

4. *Select random inputs*: randomly choose some inputs to run the program, effectively identifying test cases.

5. *Execute test case*: run the selected test case and compute its fitness with respect to the testing objective.

6. *Check fitness threshold*: if the fitness is insufficient, return to step four and select new inputs.

7. *Objective achieved*: if the fitness is satisfactory, the testing objective is considered achieved, and the process concludes.

CHAPTER 6

Project management

CHAPTER 7

Documentation's structure

7.1 General quality of documentation

A well-crafted documentation should possess the following qualities:

- *Completeness*: for the goals we need that all requirements must satisfy the goals within specified domain assumptions. For the input we want that the software behavior should be specified for all possible inputs. We also need to check for structural completenss.
- *Pertinence*: each requirement or domain assumption should be necessary for achieving a goal. Each goal should be genuinely needed by the stakeholders. The documentation should not contain items unrelated to requirement definitions.
- *Consistency*: there should be no contradictions in the formulation of goals, requirements, and assumptions.
- *Unambiguity*: clear and well-defined vocabular, unambiguous assertions, and verifiability of requirements. It must also define a clear delineation of responsibilities between the software and its environment.
- *Feasibility*: the goals and requirements must be achievable within the allocated budget and schedules.
- *Comprehensibility*: the documentation should be easily understandable by the target audience.
- *Good structuring*: every item must be defined before it is used.
- *Modifiability*: the document should be adaptable, and the impact of modifications should be assessable.
- *Traceability*: indication of the sources of goals, requirements, and assumptions, and the link between requirements and assumptions to underlying goals. This property facilitate referencing of requirements in future documentation.

7.2 Requirement Analysis and Specifications Document

The Requirements and Specifications Document (RASD) serves several purposes:

- *Communication*: it conveys an understanding of the requirements, encompassing the application domain and the system under development.
- *Contractual*: it can be legally binding, serving as a formal agreement between stakeholders.
- *Baseline for project planning and estimation*: it provides a foundation for project planning and estimation, covering aspects like size, cost, and schedule.
- *Baseline for software evaluation*: it supports system testing, verification, and validation activities. It contains the information necessary to verify if the delivered system aligns with the requirements.
- *Baseline for change control*: it establishes a foundation for managing changes in requirements as the software evolves.

The RASD document is utilized by various stakeholders, including:

- *Customers and users*: they are interested in a high-level description of system functionalities and requirements.
- *System analyst and requirement analysts*: these individuals use the RASD to specify how the system interacts with other systems.
- *Developers and programmers*: they refer to the RASD for implementation details.
- *Testers*: they use the RASD to check if the system meets its requirements.
- *Project managers*: they rely on the RASD to control the development process.

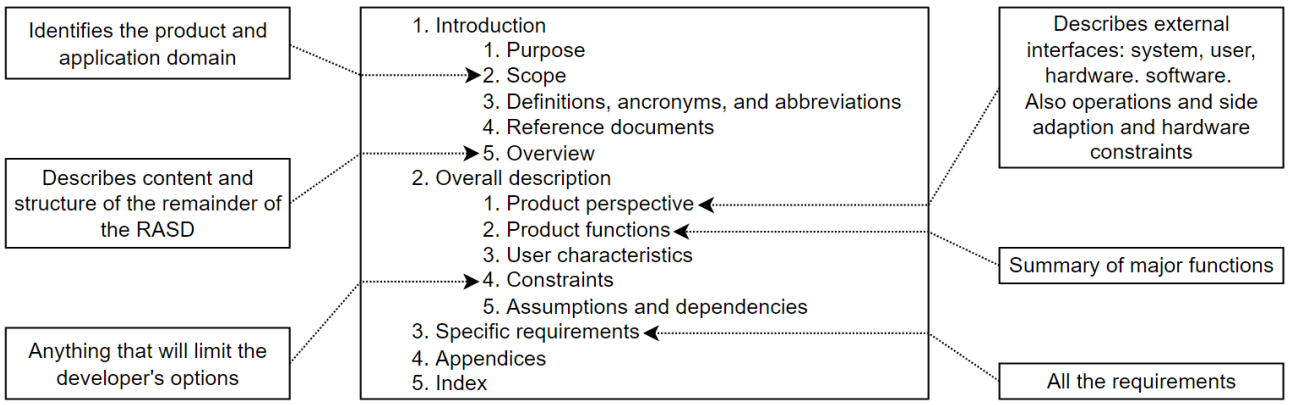


Figure 7.1: IEEE standard for RASD

7.3 Design Document

The Design Document (DD) serves various essential purposes within the software development process:

- *Communication*: it acts as a communication tool facilitating interaction among requirement analysts, architects, and developers.
- *Refinement of plan and estimations*: it allows for a more accurate assessment of the resources, time, and effort required for implementation.
- *Baseline for implementation activities*: it outlines the design choices, architectural decisions, and key components that developers will build upon during the coding phase.
- *Traceability*: this ensures that each requirement has a clear association with the components responsible for its implementation, aiding in project management and quality assurance.
- *Baseline for integration and quality assurance*: in preparation for integration and quality assurance activities, the DD plays a crucial role by:
  - Identifying the order of implementation, allowing for a structured and phased development approach.
  - Defining the integration strategy, outlining how different components will be integrated into the complete system.
  - Supporting verification and validation processes, ensuring that the developed system meets the specified requirements and quality standards.

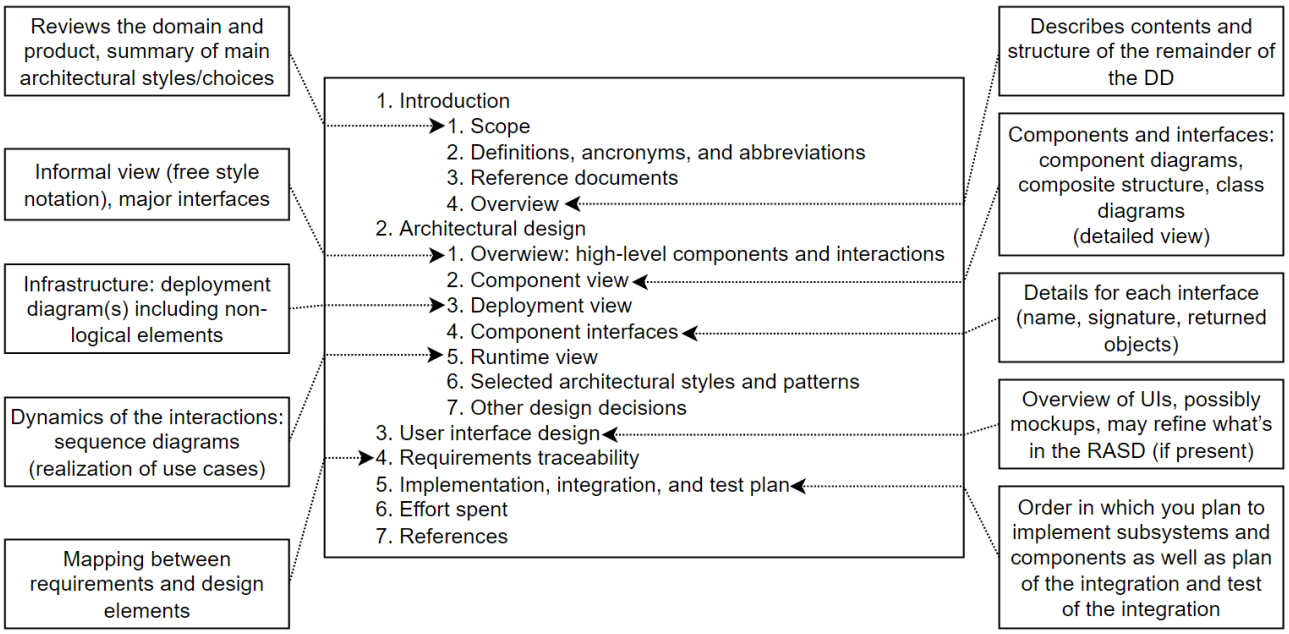


Figure 7.2: IEEE standard for DD