Software Engineering for HPC - Notes

260236

March 2024

Preface

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

- None

About:

GitHub repository

Contents

1 Introduction

1.1 The importance of software engineering

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

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

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

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

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

  More info is here.

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

  More info is here: accident tech report and video.

1.2 Software engineering: definition

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

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

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

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

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

- **Identifies** *requirements* and develops *specifications*;
- **Designs** a component to be combined with other components, developed, maintained, and used by others; component can become part of several systems;
- **Works in a team**.

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

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

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

1.3 The software product and the process

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

- **Development technology**;
- **Process quality**;
- **People quality**;
- **Cost, time and schedule**.

1.3.1 ISO/IEC 25010

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

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

Figure 1: ISO/IEC 25010

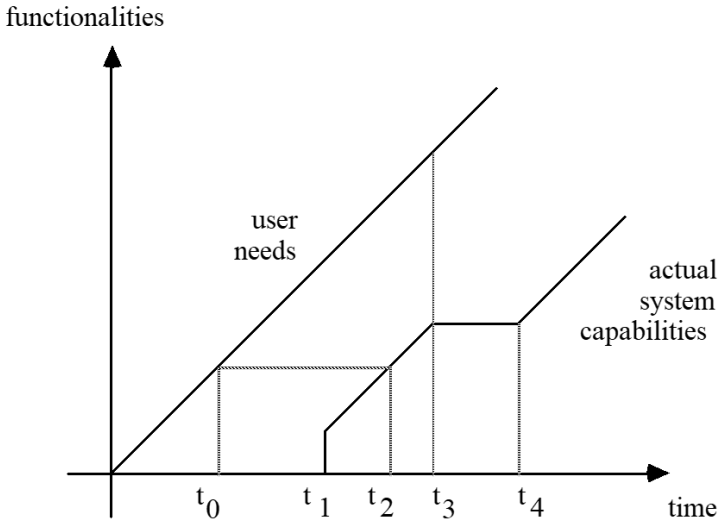
1.3.2 Productivity

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

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

1.3.3 Timeliness

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



As you can see by the graph, the "*user needs*" is a linear function (and sometimes can be exponential!). A software engineer should be able to respond to the client's requests as soon as possible. As the graph shows, a request made on time t_0 is completed on time t_2; but another request can be made at that time, and so on. The actual system capabilities can't grow up always because sometimes there are "brainstorming times" to increase product quality (ISO/IEC 25010).

1.4 Software Lifecycle

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

1.4.1 Waterfall model

The **waterfall model** is a **breakdown of development activities into linear sequential phases**, meaning they are passed down onto each other, where each phase depends on the deliverables of the previous one and corresponds to a specialization of tasks. [1] Its organization is the following:

- High phases:
  - **Feasibility Study**: this is a **cost-benefit analysis**.
    The ***main goal*** is determining whether the project should be started (e.g. buy or make), possible alternatives, and needed resources.
    The ***outcome*** is a **feasibility study document**. This paper provides:
    * A preliminary problem description;
    * Some scenarios describing possible solutions;
    * Costs and schedules for the different alternatives.
  - **Requirements Analysis and Specification**: this is an **analysis of the domain in which the application takes place**.
    The ***main goal*** is to **identify requirements** and **derive specifications for the software**. Note these specifics require a (continuous) interaction with the user and an understanding of the properties of the domain.
    The ***outcome*** is a particular document called **Requirements Analysis and Specification Document** (RASD).
  - **Design**: this is the **definition of the software architecture**. There, the definition of components (modules) and the relations/interactions among these components.
    The ***main goal*** is to **support the concurrent development of separate responsibilities**.
    The ***outcome*** is a summary of this info in a **design document**.
- Low phases:
  - **Coding and Unit Test**: each module is **implemented using the chosen programming language**. Furthermore, each module is **tested in isolation** by the module developer. Also, the programs should include their documentation.
  - **Integration and System Test**: the modules are **integrated into (sub)systems**. The integrated (sub)systems are **tested**. Follows an incremental implementation scheme. A complete system test is needed to verify the overall properties. Note that sometimes we have alpha test and beta test.

- **Deployment**: is the process used to conceive, specify, design, program, document, test, and bug fix to create and maintain applications, frameworks, or other software components.
- **Maintenance**: the maintenance is divided into **two types**:
  - **Corrective** deals with the **repair of faults or defects found**.
  - **Evolution** is also divided into **three types**:
    - ***Adaptive*** maintenance: consists of **adapting software to changes in the environment** (the hardware or the operating system, business rules, government policies).
    - ***Perfective*** maintenance: mainly deals with accommodating **new or changed user requirements**.
    - ***Preventive*** maintenance: concerns activities aimed at **increasing the system's maintainability**.

⚠ Problems derived from correction and evolution

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

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

✔ Solutions - Best practices

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

✔ Flexible processes

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

2 HPC Software, Relevant Qualities and Systems Engineering Methods

2.1 High Performance Computing Software

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

Definition 1

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

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

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

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

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

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

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

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

There are **two categories** of HPC software:

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

2.2 Relevant Qualities

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

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

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

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

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

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

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

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

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

- **Hardware Faults**.

  ⚠ Description of the problem

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

  ✔ Possible solutions

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

- **Software Faults**.

  ⚠ Description of the problem

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

  ✔ Possible solutions

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

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

2.3 Systems Engineering Methods

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

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

3 Requirement Engineering

3.1 Definition

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

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

Definition 1

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

The questions derived from requirements engineering are:

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

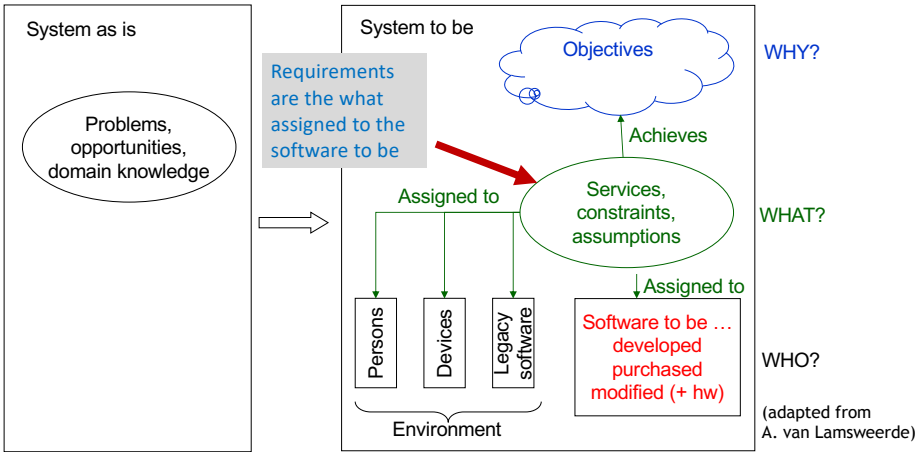


Figure 2: Analyzing the system as is and the system to be.

3.2 Studying the interplay between the world and the machine

Example 1: ambulance dispatching system

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

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

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

And more in general:

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

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

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

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

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

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

Requirements models are models of the world!

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

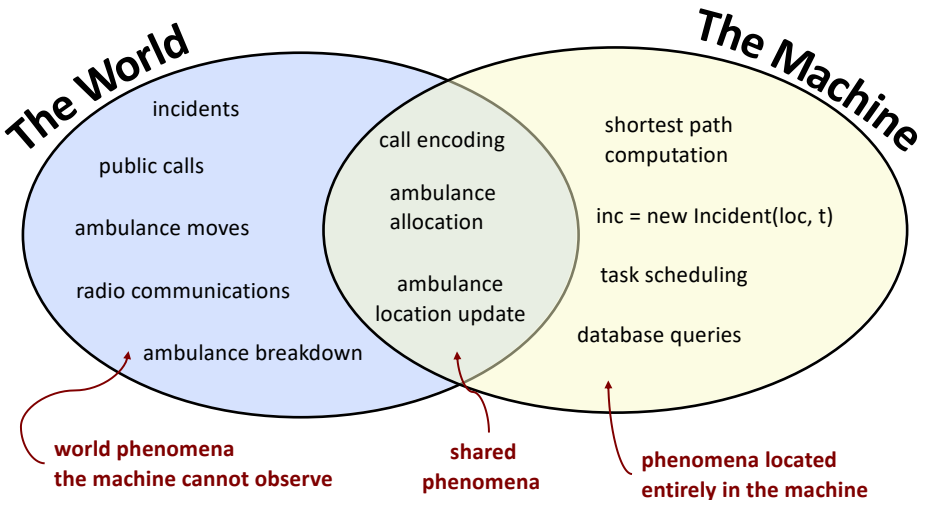


Figure 3: The world and the machine sets, with reference to example on page 16.

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

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

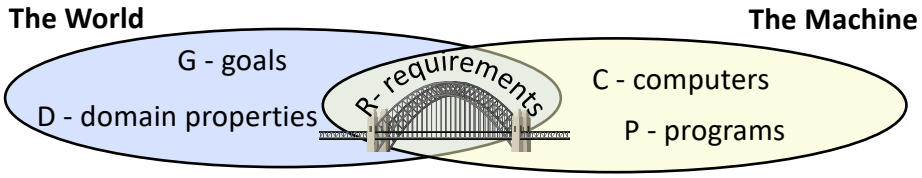


Figure 4: Goals, domain properties, requirements, computers and programs.

We explain more detailed these value inside the two sets:

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

Using the **example** on the page 16, we can identify the goal, the domain assumptions and the requirement as follows:

- **Goal**: *For every urgent call reporting an incident, an ambulance should arrive at the incident scene within 14 minutes.*
- **Domain assumptions**:
  - *For every urgent call, details about the incident are correctly encoded.*
  - *When an ambulance is mobilized, it will reach the incident location in the shortest possible time.*
  - *Accurate ambulances' location are known by GPS.*
  - *Ambulance crews correctly signal ambulance availability through mobile data terminals on board of ambulances.*
- **Requirement**: *When a call reporting a new incident is encoded, the Automated Dispatching Software should mobilize the nearest available ambulance according to information available from the ambulances' GPS and mobile data terminals.*

3.2.1 Completeness of Requirements

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

Definition 2

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

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

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

  We can make an analogy with program correctness. A program P running on a particular computer C is correct if it satisfies the requirement R: P and C $\models$ R.
- **G** captures all the **stakeholders' needs**.
- **D** represents **valid properties/assumptions about the world**.

3.3 Formulating and classifying requirements

The requirements can be of three types:

- **Functional requirements**: describe the interactions between the system and its environment (independent from implementation). In other words, are the **main** (functional) **goals the software has to realize**.

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

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

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

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

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

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

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

Example 2: are these requirements?

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

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

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

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

Example 3: types of requirements

Example of **functional requirements**:

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

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

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

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

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

Example of **technical requirements**:

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

Example 4: bad requirements

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

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

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

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

3.4 Eliciting requirements

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

Definition 3

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

Example 5: warehouse on fire

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

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

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

Alice received the acknowledgment and the ETA.

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

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

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

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

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

Definition 4: Use Cases Schema

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

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

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

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

Example 6: bad use case

Example of a bad use case referring to the ambulance dispatching example on page 16:

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

The errors are as follows:

- In the *use case name* field, the **word is a noun**. It's better to use a verb that indicates what the user is trying to achieve.
- The Dispatcher actor is **not declared** in the *Participating Actors* field, but is mentioned in the *Flow of Events* field.
- There are two main errors in the *Flow of Events* section: the first is in the sentence "*An ambulance is dispatched*". But **who sends it?** The second is in the third sentence, because **who notifies the Dispatcher?**

References

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

Index