Chapter 1

Introduction

1.1 Why is Analysis and Design Essential?

Software systems are increasing becoming a critical part of our day-to-day life. To understand the extent to which we depend on software systems today, let us look into a typical morning of John Smith and Jane Doe’s life. They are both undergraduate students at some university. To wake up for his class at 7:30 AM, John needs an alarm. His alarm uses his music player to wake him up with his favorite tune. At the same time, the programmable thermostat turns the temperature of his apt to 70 degrees to provide a comfortable environment in the apartment. John sets his thermostat to a lower temperature, while he is sleeping, to save resources. The coffee machine is also programmed to start brewing at the same time so that fresh coffee is brewed by the time John is ready for breakfast. After getting out of the bed, John heads straight to the bathroom where a motion sensor detects his presence and turns on the light and heat. After getting ready, he calls Jane on her cell-phone to ask whether she is coming to the class. In a hurry, he warms up his breakfast in the microwave oven, quickly eats it and leaves for the class by 9:00 AM. Before leaving, he activates the intruder’s alarm.

Let us analyze John’s morning and the electronic devices he used in a short-duration:
alarm clock, music player, programmable thermostat, coffee machine, bathroom motion sensor, cell phone, microwave oven, intruder alarm, car control, etc. All these devices have software parts, mostly to control their functionality. These devices and many more, controlled by software parts, have become almost indispensable for us. Today, software plays crucial role in cars, airplanes, trains, telecommunication systems, life support systems, national security and administration infrastructure, etc. It might not be life threatening for the alarm clock or music player to fail, however, faulty cars, airplanes, etc, can lead to inconveniences, financial loss, and/or loss of life.

Between 1985 and 1987, the Therac-25 medical electron accelerator was involved in six massive radiation overdoses killing three patients and injuring three others. In an investigative report about the accidents Nancy Leveson and Clark Turner [24] wrote that a lesson to be learned from the Therac-25 story is that focusing on particular software bugs is not the way to make a safe system. Virtually all complex software can be made to behave in an unexpected fashion under certain conditions. The basic mistakes here involved poor software-engineering practices and building a machine that relies on the software for safe operation. Furthermore, the particular coding error is not as important as the general unsafe design of the software overall. Examining the part of the code blamed for the Tyler accidents is instructive, however, in showing the overall software design flaws.

**Figure 1.1: The Therac-25 Accident**

In the past, malfunction of some software systems have caused significant damages. For example, between June 1985 and January 1987, the Therac-25 medical electron accel-
erator was involved in six massive radiation overdoses. As a result, several people died and others were seriously injured [24]. See Figure 1.1. On June 4, 1996, Ariane Flight 501 tore itself apart few seconds after launch because of a malfunction in the control software, making the fault one of the most expensive computer bugs in history [1]. The accident caused a loss of DM 1200 Million. On August 14, 2003, due to a bug in the Unix-based General Electric Energy’s XA/21 system, a massive power outage affected parts of the northeastern United States and eastern Canada. This outage resulted in financial loss of $6 billion and affected around 50 million people in United States and Canada [10]. There are other such incidents where incorrect software has caused major financial losses, inconveniences, and fatal injuries.

Impact of software failures are huge and sometime life threatening, therefore it is incumbent upon us as software engineers, to build it right. Most often, these errors are the result of an incorrect understanding of the desired functionality of the software system. This is where analysis of software systems helps. Other culprit is incorrect understanding of the functionality of the parts of a software system, of how they are to interact with each other, and of what they are supposed to assume about each other. This is where design of software system helps.

1.2 Analysis and Design

What is analysis? Some linguists define it as a study of composite, its components and their interactions.

*From Merriam-Webster Dictionary: analysis (noun) - an examination of a complex, its elements, and their relations, a statement of such an analysis.*
For example, a chemical analysis is the study of a sample to understand its chemical composition, i.e. the components of the chemical, its structure and function because of interactions between components. In software engineering, the term analysis is used similarly, but with a twist. During development of a software system, we analyze the problem that it is going to solve, not the system itself. In other words, we analyze the requirements of the software system. Analysis and requirement analysis are often used interchangeably in the context of software systems.

The objective of any software systems is to solve some problem. We can think of a software system as a machine that when put in a world, makes it better in some way. For example, a problem in the world might be that writing and printing documents is difficult. A word processor is a software system (machine), which transforms the world into a better place where users can afford better writing and typesetting functionalities. It is this problem world, and the better world that is the subject of analysis.

![Figure 1.2: The World and the Machine](image)

### 1.2.1 Requirements and Specification

Informally, requirement describes the world with the problem, and how it can be transformed into a better world. Jackson defines the requirements as:

> The requirement is an explicit description of the behavior and properties that we want the world to have as a result of its interaction with the machine.

---

1.2. **Analysis and Design**

It is a description in the optative mood that is, it expresses what we would like to be true. It is a description over the phenomena of the world that are of interest to the customer of the development: it captures the purpose for which the machine is to be built and installed. [16, pp. 47]

A specification on the other hand is the description of the machine and its functionality. In Jackson’s view the specification describes the behavior and properties that we want the machine to have at its interface with the world. It is an optative description. It is a description over the shared phenomena at the interface, consistent with the properties of the world and satisfiable by appropriate action of the machine [16, pp. 47].

A major purpose of specification is to establish a set of criteria against which the software system will be evaluated. A design is a high-level description of the parts of the machine and a description of how these parts will interact to satisfy the specification. To make the distinction more concrete let us reconsider the text editing scenario. To make the world without text editing programs a better place, the requirements would describe what it means for a text to be formatted, if a text and some formatting directions are available. This broad requirement is shown in Figure 1.3.

1. If a user has some English text
2. and formatting directions
3. they would receive a document such that the text in the document matches the supplied text and format matches the supplied formatting directions.

**Figure 1.3: The Requirements of the Text Editor**

Note that many different mechanisms can fulfill this requirement. A secretary for example can also take the text and the formatting directions and produce a document. In this case, secretary is the machine that makes the world a better place. One could then write a simplified specification of such a machine as shown in Figure 1.4.
This specification states that there will be two inputs to the machine: some text in English and formatting instructions. The machine will output the formatted text.

### 1.2.2 Formal and Informal Specifications

Informal specifications like that in Figure 1.4 are easy to write, however, they are often subject to multiple interpretations. It is also very difficult to check if informal specifications are complete and correct. For example, the informal specification in Figure 1.4 does not specify the relation of the output formatted text with the input text and formatting instructions. This is example of an *incomplete specification*. Informally, an incomplete specification does not provide sufficient information about the machine to be able to unambiguously understand and/or create such machine. To complete this specification, one may change the sentence describing the output to: “Formatted text, such that it matches the input text and format matches the input formatting instructions”. Specifications can also be *incorrect*, in which case the machine described may be different from that required to solve the problem in the world.

To avoid ambiguity and make it easier to check specifications for correctness and completeness automatically, often, critical software systems are specified mathematically or formally. For example, the specification in Figure 1.5 describes that the input text will consist of zero or more letters in both upper and lower case and ten digits and their corresponding locations. To make the description simple, we are not considering words and
1. input: A text in English $t$, where
   $t \in T = \{(l,p)^*: l \in L \land p \in P\}$
   $L = \{A-Z,a-z,0-9\}$
   $P = \{\text{integers} \geq 0\}$

2. input: Formatting instructions $i$, where
   $F = \{\text{bold, italic, underline}\}$
   $i \in I = \{(p,f)^*: p \in P, f \in F\}$

3. output: Formatted text corresponding to original text $(ft,t)$, where
   $ft \in \{(l,p,f)^*: l \in L, p \in P, \land f \in F\}$

Figure 1.5: A Simple Mathematical Specification of the Machine

punctuations. Furthermore, it specifies that a formatting instruction will consist of zero or
more \{(position (p), format (f))\}-tuples, where position is a positive integer and format can
be either bold, italic or underline. It also says that the machine will output a formatted text.
This formatted text will be of the form [letter, position, format]. This specification is an
example of formally defined specification. In many cases, it is not possible to define a for-
mal specification or it is very time consuming. In such situations, a specification is usually
an informal text such as that in Figure 1.4, however, they are more error-prone compared
to formal specification so extra care should be exercised while writing such artifacts.

A specification is the basis of the rest of the software de-
velopment activities, such as high-level design, detailed de-
sign, implementation and verification. Therefore, an error in
specification process is costlier compared to errors in later
part of the software development such as design as the impact
of the error propagates to the other software development ac-
tivities downstream. For example, an incorrect specification
will lead to incorrect design, incorrect implementation, incor-
correct test cases, etc. The cost of fixing this error will be greater
compared to the cost of fixing an error in say implementation.
1.2.3 Refinement of System Descriptions

We can build many different machines from a single specification. In other words, more than one machine can satisfy a given specification. The design further refines the details of the machine. For example, the design shown in Figure 1.6 will satisfy the specification in Figure 1.4.

This particular design of the machine splits the functionality into three parts represented by three rectangular boxes in the figure. The first part of this machine reads the text and the format as an input. The second part of this machine formats the text according to the formatting directions. The third part of this machine outputs the formatted text. The design also shows using arrows from start to end that these three parts will do their respective functions serially. Each of these parts is dependent upon the previous part in the chain for providing the right input and responsible to provide the right output to the later part.

Like specification, we can build several different software systems that fit a design. In other words, more than one software system can satisfy a design. However, this number is smaller compared to those that satisfy the specification. The types of machines that satisfy the specification are in turn smaller compared to those that will satisfy the requirements. In each case, we have refined the previous description gradually making it more specific.

Let us look at an alternative design derived from the same specification. This design is shown in Figure 1.7. In this design, the functionality of the software system is split into four different parts: input, output, formatter, and the control. The input and the output
1.2. **Analysis and Design**

parts work like the previous design, reading the text and the formatting directions from the user and outputting the formatted text. The formatter processes the text and the formatting instructions to produce the formatted text. The control part co-ordinates the action of other parts in the system, invoking them as necessary.