Chapter 1

Introduction

This book is about analysis and design of software-intensive systems. As a reader, you might ask why study analysis and design at all? In basic programming and other similar courses, you have learned programming. You have probably written some if not large amount of code in X, where X is your favorite language such as Java, C#, C++, etc. Why is the knowledge of programming not sufficient to build significant software systems? Some of you may be apprehensive about this topic, wondering whether design is really the responsibility of the software architects. May be you think design is all about writing those lengthy repetitive documents. May be you have also come across the myth: junior software engineers are never given the chance to design! So, why bother studying it. This book will try to address all these apprehensions and myths. Let us first talk about software-intensive systems, and study some software intensive systems of modern age.

1.1 Software-Intensive Systems

Just for fun, let us look into a morning of John and Jane’s life. They are both undergraduate students at the world-class 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 alarm.

Let us analyze John’s morning and the software systems 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, are around us as integral parts of our day-to-day life. 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 be faulty, however, faulty cars, airplanes, etc, can lead to inconveniences, financial loss, and/or loss of life.

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 accelerator was involved in six massive radiation overdoses. As a result, several people died and others were seriously injured [22]. 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 designers, 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 Requirement, Specification, and Design

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

1 (See Figure 1.1). For example, a problem in the world might be that writing and

\[1\] This view is inspired from Michael Jackson’s book [14] and his paper [15].
printing documents is difficult. A word processor is a machine, which transforms the world into a better place where users can afford better writing and typesetting functionalities.

A requirement is a description of the world, and how it can be transformed into a better world. A specification on the other hand is the description of the machine and its functionality. 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 editor scenario. To make the world without text editing programs a better place, we would have to help users who have text and formatting directions so that they can produce a document that matches the supplied text in the given format. This broad requirement is shown in Figure 1.2.

1. A user has this English text
2. and some formatting directions
3. and would like to receive a document that has the text formatted using the directions.

Note that many different machines 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 1.3.

This specification says that there will be two inputs to the machine. A text in English and formatting instructions. The 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 punctuations. A
1. input: A text in English $t$, where 
   $$ t \in T = \{(l, p)^* : l \in L \land p \in P\} \quad L = \{A - Z, a - z, 0 - 9\} $$
   $$ P = \{\text{integers} \geq 0\} $$

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 1.3: The Simplified Specification of the Machine

formatting instruction will consist of zero or more $[(\text{position} (p), \text{format} (f))]$-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 $[\text{letter, position, format}]$.

This specification is an example of formally defined specification. In many cases, it is not possible to define a formal specification or it is very time consuming. In such situations, a specification is usually an informal text. An informal specification is usually error-prone so extra care should be exercised while writing such artifacts. A specification is also the basis of the rest of the software development activities. An error in specification process is thus costlier than errors in later part of the software development as its impact are greater.

We can build many different machines from this specification. The design further refines the details of the machine. For example, one can produce a design as shown 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 takes 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.

Like before, we can build several different type of machines that fit this design, however, the number of different types of machine that satisfy this description are 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.
1.2. REQUIREMENT, SPECIFICATION, AND DESIGN

Figure 1.4: The High-Level Design of the Machine

making it more specific.

Exercises

1. Is the specification of the secretary in Figure 1.3 correct? Is it complete? If you answered yes to any of the previous two questions please correct or complete the specification.

2. Some details of the specification in Figure 1.3 are not reflected in the design in Figure 1.4. Please complete the design by writing the details of each part of the machine in the following form:

Part: part name
Input: formal description of the input to this part
Output: formal description of the output of this part