Notes on Software Design
Part II: Software development techniques

PJA de Villiers

Chapter 1

Introduction

The development of complex systems that are useful to mankind sooner or later give rise to mature engineering disciplines. Many universities have engineering faculties where new techniques are being developed and where students are trained who then join industry to apply what they have learnt. But the development of software cannot yet be called a mature engineering discipline. Perhaps because it is still so new and exiting, the software industry has attracted many people who do not understand the principles of software development. In fact, after learning a popular programming language, some people consider themselves qualified programmers and sell their services to anyone willing to hire them. Such people are easily misled by untested ideas that promise wonderful results.

Considering the lack of appropriate training of many programmers in industry, it is hardly surprising that so many large software projects are completed long after their promised deadlines and that budgets are often exceeded.

It is perhaps natural to think that the process of producing quality software should be similar to the production of other complex systems such as motor vehicles or electronic devices, but this is not so. The production of complex software systems seems to be fundamentally difficult and is expected to remain so. There is no shortage of advice, though. As a quick search of Amazon.com will confirm, there are many books on the market about so-called "software engineering"—currently a collection of methods that sometimes involve little more than common sense. Topics covered include project management, requirements analysis and various design methodologies. But there is no single, widely accepted method that will always work. In fact, in the article "No Silver Bullet: Essence and accidents of software engineering" [1] it is argued that no single method can be expected to revolutionise software development any more; the problems that were simple to solve have been solved. The extensive experience of the author, Brookes (jr), who also wrote the celebrated book on software development"The Mythical Man-Month" [2], makes it difficult to simply brush this disturbing argument aside.

The process of producing software is normally divided into several phases,

although it does not mean that each phase must be completed before starting the next:

1. The requirements of the product are documented informally.

2. Transformation of the informal requirements to produce a specification that can be used by developers. Clients (the people for whom the software will be developed) are not expected to understand this documentation, although every effort should be made to ensure that the (informal) requirements are met.

3. Design of the major structures involved in the system. Different strategies are compared and decisions are made about how things should be done. Design involves making compromises. It is seldom possible, or necessary, to make everything equally efficient, or to always use the least possible storage. The goal should be to devise a solution that meets the requirements while being an acceptable balance of various considerations such as execution speed and memory usage.

4. Writing code. This phase is too often seen as the main activity of the software production process because the code directly represents the end product. However, if coding is started before describing precisely *what* is to be done and *how* to do it, much time (and money) can be wasted because code may have to be thrown away.

5. Testing of individual modules. Since large software systems are too complex for one person to understand, there are many opportunities for making errors. Testing is the traditional method of detecting such errors. (The most effective way of *preventing* errors, however, is to develop clear specifications. Most errors in practice can be traced back to incomplete, ambiguous or erroneous specifications. Even worse, too much software in industry is being developed without any specification whatsoever! This alone is the reason for the failure of many projects.)

6. Integration of individual modules into a working system. This phase is more than simply putting everything together. Since software is produced by several programmers who work individually on different components, there are usually some surprises. For example, in one module height above sea level may be represented in feet and in another in meters. Such an error cannot be detected by a compiler, since both quantities may be represented as integers. It is therefore essential to check the transmission of values between system modules carefully during the integration phase.

7. Finally, successful systems often need to be changed. This is so because users quickly dream up new application areas which could be handled if some feature is added to the system, or if something could work in some new way. Last, but not least, defects may be discovered and those have to be corrected. This process of constant change of an existing system is

called maintenance and it can represent up to 70% of the cost involved in software products.

In 1956 the first compiler was completed—it took 18 man-years! In those days it was a huge project that involved several brilliant people. On the other hand, the compiler project described in Part I can be completed by a few students with limited programming experience in one semester. It is still not a trivial exercise, but the difference in productivity, if compared to the 1956 project, is striking. The reason is that most problems associated with compilers have been solved over the years and that we can simply reuse those solutions.

This principle—to develop techniques that can be applied to solve classes of problems—underlies engineering in general. It is the task of scientists to develop *new ideas*. The task of engineers, on the other hand, is to transform promising ideas into *techniques that can be applied routinely in industry to solve technological problems*.

In this part of the notes we reflect on compiler development as an exercise in software development. The various development stages are also encountered during the development of other types of software.

The first task encountered during compiler development is to write a *specification*. For the compiler, this meant writing an EBNF definition of the source language and writing an interpreter for the target machine. This described precisely *what* was needed: a program that will translate any source text into a functionally equivalent target text. A similar phase should be part of every software project although EBNF may not be the right specification language to use. There are several other formalisms that can be used for specification and we will study some of them.

The next step, for the compiler, involved working out the general principles of *how* to accomplish the compilation task. This was the goal of the *design* phase. Design is, again, essential for any kind of non-trivial software. Compromises are always necessary and it is wise to explore various alternatives in principle before any code is developed.

Students who register for this course should already be experienced coders. However, programmers keep learning all the time and some guidelines may help to improve your skills.

Finally, we will study the fundamental ideas of systematic testing. Although testing can never guarantee the absence of defects, any defect detected during development certainly improves the quality of the final product.

Chapter 2

Requirements and Specification

The first step to success is to understand precisely what is needed. This may sound obvious, but when a project is sufficiently complex, it is all too easy to misunderstand some of the requirements. This usually means that work will have to be redone—a costly exercise in most cases. Another important issue is to concentrate on *what* is required and to avoid getting distracted by the details of *how* the various problems could be solved.

2.1 User requirements document

Successful projects start by writing a user requirements document. The goal is to describe as precisely and as completely as possible what the user (or the client) expects to see by the completion date. There are especially two areas to be addressed: (1) the functionality of the system must be described and (2) several constraints on the final product (such as the programming language used or the response time to user requests) must be described.

Sometimes the client is an expert and knows exactly what he or she wants. More often, however, this is not the case and the client has only a vague idea of what is needed. This lack of precise knowledge about the requirements of a system—for whose development the client is prepared to pay a substantial amount of money—should not be too surprising: since the system does not exist yet, it is hard to envisage every detail about its intended behaviour or how it will be used. It is therefore important for the key people involved in the project to discuss every issue and to document decisions.

The resulting document—the user requirements document—is an *informal* one, since the client is usually not an expert in software development and he or she must be able to understand everything. However, any means of eliminating ambiguity should be used. This may include diagrams, tables, screen layouts, lists of requests to be processed by the system, and the expected results for

specially selected test data. Questions about future extensions of the system may seem out of place before development starts, but some extensions may influence the design. It is therefore essential to ask the clients about this and to agree about constraints on response time and storage space before development starts. It is also important to be on the lookout for inconsistent requirements, even at this early stage.

The user requirements document is needed to prepare a tender document to develop the system. It is therefore necessary to include enough detail to make a cost estimation possible. This is often little more than guess work, since a decision must be made before development starts. A good strategy is to divide a complex project into several stages. This approach helps because the first stage will usually involve that part of the system where information is most freely available. Completion of the first stage will uproot some problems that might have been overlooked and this will affect the second stage. In this way progress is made in a manageable way.

If a software product is not developed for a specific client, but is meant to be marketed to customers—something like a new computer game—it is obviously necessary to do some market research to determine which features will attract the largest number of buyers. If there is a competing product, it is necessary to include something special that will make potential buyers decide that it is worthwhile to try something else. If the product is completely new, potential buyers should be kept in mind to decide which features to include. Whatever the situation, a user requirements document is always needed.

2.2 Specification techniques

EBNF notation is indispensable during development of a compiler although other applications usually ask for different formal notations. Many such notations have been developed and some have found their way into industry. Many formal notations for specification are based on logic, set theory and automata theory—theoretical fields that are fundamental to Computer Science.

2.3 Propositional logic

A *proposition* is a statement that is either true or false. Something like "the value of `x` is less than 10". We represent a proposition in a specification in a similar way to variables in programming languages—by an identifier. For example, the identifier `temphi` represents the proposition "the temperature is too high" and it may have one of the values `true` or `false`. (What is meant by "too high" should be specified somewhere.)

We can combine propositions by using operators. These include the following:

- $\neg a$ ("not a")

- $a \wedge b$ (" a and b")
- $a \vee b$ ("a or b")
- $a \Rightarrow b$ ("a implies b")
- $a \equiv b$ ("a is equivalent to b")

It is possible to capture the essence of some systems by writing down sets of logical formulas. For example, the formula

$$(temphi \wedge pressurehi) \Rightarrow alarm$$

means that if both the temperature and the pressure are too high, the alarm should be sounded. This is a requirement to be met by the system and therefore something the designer(s) of the system should take into account. The specification $a \Rightarrow b$ therefore should be interpreted as "if `a` is true, then `b` must be true. The leftmost operator `a` is a clearly a condition, but since programmers are used to interpret the "then" clause of an `if` command as something to be "done", a warning is necessary: the right side of an implication is also a condition; it describes a property that should hold. In the example above, this is the property that the alarm is sounding; it is not the action of switching on the alarm. In other words, a specification tells us *what properties should hold*, and not *how to do it*. This difference is important and something that is often confusing to beginners.

Logical operators are evaluated in a specific order. Negation ($\neg$) is evaluated first, then and ($\wedge$), or ($\vee$), and finally $\Rightarrow$ and $\equiv$. Brackets should always be used to prevent uncertainty. Certain operators are basic in the sense that we cannot do without them. Other operators can be defined in terms of the basic operators. For example, $a \Rightarrow b$ can be defined as $\neg a \vee b$ and $a \equiv b$ as $(a \Rightarrow b) \wedge (b \Rightarrow a)$.

Some expressions are always true no matter what values are substituted for their propositional variables. Such expressions are called *tautologies*. Other expressions can never be true no matter what values are substituted and these are called *contradictions*. An example of a tautology is the expression $a \vee \neg a$. It is true whether a is true or false. The expression $a \wedge \neg a$ is a contradiction. For simple expressions, it is easy to see whether they are tautologies or not, but for more complex expressions such as $(p \Rightarrow q) \Rightarrow ((q \Rightarrow r) \Rightarrow (p \Rightarrow r))$ we can use any one of several methods:

1. Use a truth table. Basically we assign all possible combinations of truth values to all variables and evaluate the expression's value in every case. If the formula evaluates to "true" in every case, it is a tautology. Although this method is easy to understand, it is cumbersome to use for larger formulas.

2. Simplify the expression by substituting subformulas by equivalent subformulas. If the end result is the value "true" the original formula is a tautology. This method can be more effective than truth tables, but some practice is needed to identify the best subformulas to replace.

3. Assume the expression to be "false" and try to derive a contradiction. If this is possible, the assumption was incorrect and the expression is a tautology. If a contradiction cannot be derived, we have determined a set of values that makes the formula false and it therefore is not a tautology. This method is often the most effective.

Exercise Use all three methods to determine which of the following expressions are tautologies:

1. $p \vee \neg p$
2. $p \wedge q$
3. $p \Rightarrow (q \Rightarrow (p \wedge q))$
4. $(p \Rightarrow q) \Rightarrow (\neg q \Rightarrow \neg p)$
5. $(p \Rightarrow q) \Rightarrow (\neg p \Rightarrow \neg q)$

2.4 Simplification of logical expressions

Simplification of propositional expressions is carried out in the same way algebraic expressions are simplified. For example, the statement $z = x * 1$ can be simplified to $z = x$ because if we multiply an integer value x by 1 the result is simply x. This is an example of a simple *rule of inference* that can be stated formally as

$$\frac{x * 1}{x}$$

The part above the line is a proposition that must be true for the rule to be applied. The rule states that we can infer that the proposition below the line holds.

To simplify expressions in propositional logic we need several inference rules, some of which have generally accepted names. Here are a number of rules that are useful in general for simplification of propositional expressions:

1. $\frac{p \wedge q}{p}$ (If p and q is true we infer that p is true.)
2. $\frac{p \wedge true}{p}$
3. $\frac{p \wedge false}{false}$
4. $\frac{\neg\neg p}{p}$ [*double negation*]
5. $\frac{p \vee \neg p}{true}$
6. $\frac{\neg(p \vee q)}{\neg p \wedge \neg q}$ [*De Morgan*]

7. $\frac{p \wedge \neg p}{false}$ [*contradiction*]
8. $\frac{p \Rightarrow q,\ \neg q}{\neg p}$
9. $\frac{p,\ p \Rightarrow q}{q}$
10. $\frac{p \vee (q \wedge r)}{(p \vee q) \wedge (p \vee r)}$ [*distributive rule*]

When complex expressions are simplified, it helps to indicate explicitly which rules of inference are being used. For this we need a formal notation. We write one expression per line and assign a unique number to each line. These numbers are useful when it is necessary to refer to previous expressions. In addition, we indicate the inference rule used to derive each line as shown in the example below.

1. $(open \vee \neg open) \wedge open$
2. $true \wedge open$ [1, $\frac{p \vee \neg p}{true}$]
3. $open$ [2, $\frac{true \wedge p}{p}$]

Although in each line of the simple example above we refer to the previous line, this is not always the case. In a more complex derivation it is sometimes necessary to refer to more than one line, depending on the rule applied. The notation supports this as will be shown in what follows.

2.5 Analysing specifications

Specifications, if they are to be of any use, should not contain errors. A formal notation helps by making the description precise. However, with specifications of practical size and complexity it is difficult to be sure that nothing has been left out or that some statements are not contradictory. We must be sure that all user requirements are satisfied. For this, we need to analyse the specification.

Are specific properties satisfied?

To determine whether a specification satisfies specific user requirements, we can use standard techniques from logic. Assume that a specification contains the expressions S_1, S_2 and S_3 and that we want to show that the user requirement C is satisfied. We can proceed to show that $S_1 \wedge S_2 \wedge S_3 \Rightarrow C$ by simplifying the expression, showing that it is true. Unfortunately this is impractical for all but the simplest specifications because the expressions are too complex to handle.

Another approach is to use the method of indirect proof or proof by contradiction. The first step is to negate what we want to prove. For the example above, this means assuming that $\neg C$ is true. We now proceed to manipulate expressions, hoping to generate a contradiction. If we are successful, we know

that our original assumption (that the property C we wanted to prove is false) was wrong. Therefore, the property C is true.

To illustrate the principle, assume that a specification contains the following formulas:

$S1\ p \Rightarrow q$

$S2\ q \Rightarrow r$

If we want to prove that user requirement $C : p \Rightarrow r$ is satisfied by the specification, we assume $\neg C$ to be true. Now we can write down the following lines, each of which is true, based on the assumption that $\neg C$ is true:

1. $\neg(\neg p \vee r)$ $[assumption, def\ impl]$
2. $p \wedge \neg r$ $[1, DeM]$
3. p $[2, and]$
4. $\neg r$ $[2, and]$
5. q $[3, S_1, def\ impl]$
6. r $[5, S_2, def\ impl]$
7. $false$ $[4, 6, contr]$

Since we have shown that a contradiction can be derived, we know that the property we wanted to prove is true.

Inconsistencies

It is essential to check specifications for inconsistencies since an inconsistent specification cannot be implemented—it describes requirements that cannot be satisfied. Again, the technique of indirect proof is useful. First we need a workable definition of what is meant by an inconsistent specification. A list of expressions is inconsistent if and only if there does not exist a set of values that can be assigned too the variables contained in the expressions that will make them all true. That is, at least one of the expressions will be false, no matter what values we assign to the variables.

Assume we are given a specification that contains expressions $(S_1, S_2, \ldots S_n)\$$. The specification is inconsistent if we can show that $S_1 \wedge S_2 \wedge S_3 \ldots \wedge S_n$ is false. Again, we can use indirect proof to accomplish this.

Exercise

Determine whether the following specification is inconsistent:

$Pr1 : HighPressure \Rightarrow ValveOpen$

$Pr2 : ValveOpen \Rightarrow Bell$

$Pr3 : \neg Bell \vee Reset$

$Pr4 : \neg HighPressure \Rightarrow Reset$

$Pr5 : \neg Reset$

2.6 Predicate Logic

Propositional logic is fundamental to specification of system properties, but it is not expressive enough. First, we need to refine the concept of a proposition. We now assume that a proposition can be any expression which yields a value of true or false. What we gain is the ability to refer directly to variables that are relevant to the system we want to describe. For example, we can now write an expression $temp > 100$ instead of the less specific $temp_h i$ allowed by propositional logic. This improves our ability to state precisely what we mean. Programmers like to think of predicates as Boolean functions. Second, it is often necessary to describe properties of entire classes of objects. For this we need quantifiers—operators on classes of objects. The most useful of these quantifiers are the operators $\forall$ which states that *all* objects in a certain class have a certain property, and $\exists$ which states that *some* of the objects in a certain class have a certain property.

Simple examples

- The fact that there exists an integer x whose value is at least 1 and at most 10 such that it has the value 25 when multiplied by itself can be expressed concisely as follows: $\exists\, x : 1..10 \bullet x^2 = 25$
- The requirement that all files older than 21 days should have a backup is described by: $\forall f : File \bullet Age(f) > 21 \Rightarrow Backup(f)$

Note that the variable x in the first example above could be replaced by any other variable. We say that variable x is *bound* to the quantifier $\exists$. The variable x is within the scope of the quantifier and in a more complex example it may be necessary to use brackets to delineate the intended scope. The truth value of the formula $\exists\, i : 1..10 \bullet i < j$ depends on the value of j. The variable j is clearly not under control of the quantifier and is called a *free* variable. Its value must be defined for the formula to be meaningful.

2.7 Exercise

Formalise the following requirements by using predicate logic:

- All files marked for archiving must be written to tape and deleted from disk
- Some communication lines are not in use.
- No communication line can be in use without having a status of 0.

2.8 Nested quantifiers

Quantifiers can be nested to express more complex requirements as illustrated by the following examples:

- There is at least one computer whose communication lines are all active: $(\exists\, c : \mathit{Computer} \bullet (\forall\, l : \mathit{Line} \bullet \mathit{Active}(c, l)))$
- Some users do not have access to all programs:
- $(\exists\, u : \mathit{User} \bullet \neg(\forall\, p : \mathit{Program} \bullet \mathit{Access}(u, p)))$

2.9 Exercise

Explain the difference between the following requirements:

1. $\neg(\forall\, u : \mathit{User} \bullet (\forall\, p : \mathit{Program} \bullet \mathit{Access}(u, p)))$
2. $(\forall\, u : \mathit{User} \bullet (\exists\, p : \mathit{Program} \bullet \neg \mathit{Access}(u, p)))$
3. $(\forall\, u : \mathit{User} \bullet \neg(\exists\, p : \mathit{Program} \bullet \neg \mathit{Access}(u, p)))$

2.10 Rules of inference

Here are some important inference rules for the predicate logic which allow us to simplify expressions:

1. $\frac{\forall\, x{:}S \bullet P(x),\ \forall\, x{:}S \bullet Q(x)}{\forall\, x{:}S \bullet P(x) \wedge Q(x)}$
2. $\frac{\forall\, x{:}S \bullet P(x)}{\exists\, x{:}S \bullet P(x)}$
3. $\frac{\forall\, x{:}S \bullet P(x)}{P(a)}$ if $a \in S$
4. $\frac{P(a)}{\exists\, x{:}S \bullet P(x)}$ if $a \in S$
5. $\frac{\neg(\forall\, x{:}S \bullet P(x))}{\exists\, x{:}S \bullet \neg P(x)}$
6. $\frac{\neg(\exists\, x{:}S \bullet P(x)}{\forall\, x{:}S \bullet \neg P(x)}$

2.11 Specifications involving arrays

Assume that array X has n elements of type integer and that indexes start at 0. We can use quantifiers to state different requirements:

1. All elements of array X are 0:
   $(\forall\, i : 0..n \bullet X[i] = 0)$

2. The elements of array X are in ascending order:
   $(\forall\, i : 0..n-1 \bullet X[i] < X[i+1])$
3. All elements of array X are unique:
   $(\forall\, i, j : 0..n \bullet i \neq j \Rightarrow X[i] \neq X[j])$

Note that requirement 2 above is stronger than requirement 3. If the elements of X are in strict ascending order, they must all be different and thus unique. However, if we know that all elements of X are unique, it does not mean they must be in ascending order; they can be in any order whatsoever as long as they are all different. Weaker specifications usually leave more options for implementation and are preferable in general.

2.12 General format of specifications

Good specifications capture the essence of a system and should be easy to understand. Usually, the following information is required:

1. A concise description of the *state* of the system;
   - The main *data structures*
2. All *operations* defined on the main data structures
   - Precondition (condition that must hold *before* execution of the operation)
   - Postcondition (condition that must hold *after* execution of the operation)
3. The initial values of the main data structures

2.13 Example

To specify the functionality of a procedure to compute fuel consumption, we need to think about the input data required and the result of the computation. We need to know the distance travelled and the amount of fuel used as input. A single result is computed from these data. Even for this simple procedure there are a number of choices that can influence the specification (and the implementation). First, we can measure distance in miles or kilometres and the programmer must know what is required. Second, the result can be expressed as litres per 100 kilometres or as kilometres per litre. Although the names of variables could be selected in any sensible way, there is a convention to use primed names for quantities after the operation has been completed and names without primes for quantities before the operation has been started. Here is a specification that states clearly what is required.

- Precondition: $l \geq 0 \wedge km > 0$
- Postcondition: $res' = l * 100/km$

Note that we require the distance travelled to be a positive number because the computation involves division by this number. The precondition could be seen as a restriction placed on input values. If these restrictions are not met, the computation is not guaranteed to be correct. The implementation could check that these restrictions are always met or the checking could be done by the environment of the operation. The postcondition describes the result which is indicated by the primed variable res'. In this simple case, the specification provides all information needed for the implementation, but this need not be the case. For example, if we specify a procedure to sort an array of numbers, it is only necessary to specify what we mean by a sorted array; whether to use bubble sort, quicksort or some other algorithm is left to the programmer to decide.

It is possible to write specifications for most computations by using only predicate logic, but to specify data structures we need the notation of set theory—the topic of the next chapter.

Bibliography

[1] Brookes, FP, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[2] Brookes, FP, Jr. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1999.