

Flexibility and Maintainability

Learning Objectives

One of the main ambitions of this book, as evident from the word *flexible* in the title, is to demonstrate techniques to design and develop software that can easily accommodate change. “Change is the only constant”, originally formulated by Greek philosopher Heraclitus, is a quote that has been adopted by the agile community as a truth about software development. Once customers and users begin using a software system they get new ideas for improving it. And the software that handles today’s requirements will need to be adopted and changed to handle those of tomorrow. Thus, software must be designed and developed to make it “easy to change.”

It turns out that there are many properties that contribute to make software either easy or difficult to change. The learning objective of this chapter is to establish a solid terminology of the different qualities or characteristics that influence ease of change, allowing us to discuss techniques, like design patterns and frameworks, at a more precise level. Specifically, I will introduce the ISO 9126 standard definition of maintainability and the sub qualities it is composed of.

3.1 Maintainability

Let me start by a small code fragment. The code below is correct, it compiles, and it defines a well-known, everyday concept. The main method demonstrates its operations.

Listing: chapter/maintainability/example1/X.java


```
public class X {  
    private int y;  
    public X() {  
        y = 0;  
    }  
    public int z() {
```

Copyrighted Material. Do Not Distribute.

```

    return y;
}
public void z1(int z0) {
    y += z0;
}
public static void main(String[] args) {
    X y = new X();
    y.z1(200);
    y.z1(3400);
    System.out.println( "Result is " + y.z() );
}
}

```

 What abstraction does the class represent? What behavior does each method represent?

Can you guess what abstraction it is¹? The key point here is that the way you name abstractions in source code has profound influence on how easy it is to understand. And code that is hard to understand is next to impossible to change, enhance, and correct.

Next consider the same class X but subject to a novel indentation style.

Listing: chapter/maintainability/example2/X.java

```

public class X{private int y;public X(){y = 0;}public int z(){
return y;}public void z1(int z0){y += z0;}public static void main(
String[] args){X y=new X();y.z1(200);y.z1(3400);System.out.println
("Result is " + y.z());}}

```

From the point of view of the compiler, this latter version is identical to the first—it is just some white space that has been removed. For a developer, however, it has become incomprehensible. Understanding software requires understanding structure, and proper indentation is a key technique.

Both versions of class X have some qualities and lack other qualities. For instance, the last version of X only require 231 bytes of my hard disk space while the first version takes 314 bytes. This is a substantial difference of 35%. Thus the second version is the “best” when it comes to buying storage. The question is whether this “storage” quality is important or not.

When we discuss software quality we have to be as objective as possible rather than fall into arguments about what is “good” or “bad.” The last chapter introduced the ISO-9126 definition for reliability, and I will again take my starting point in this internationally accepted quality model that allows software engineers to more precisely define aspects of quality in their software products.

Definition: Maintainability (ISO 9126)

The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

¹You can find an identical class but with proper naming on the website in the same folder as class X.

Basically, maintainability is a *cost measure*. Any software system can be modified (in the extreme case by throwing all the code away and write new software from scratch) but the question is at what cost? Thus the interpretation is that *maintainable software* is software where the *cost of modifying it in order to add/remove/enhance features is low*. Consider the X class above that is difficult to change—it is less maintainable.

Maintainability cannot by nature be a global quality of a given software system in the sense that all parts of the software system can easily be modified for all types of changed requirements. You always have to state in what respect it is maintainable. For instance, an accounting system's source code may have been designed and implemented in anticipation that at some later date it may have to handle different currencies and taxation systems (thus easily modified for use in a new country) but have a fixed graphical user interface (thus difficult to modify for a new operating system).

Exercise 3.1: Give examples of some software systems where maintainability is very important. Give examples where maintainability is not an issue.

3.2 Sub Qualities of Maintainability

A highly maintainable software unit must possess quite a few different qualities and the ISO standard acknowledge this by stating that maintainability is composed of several, finer grained, qualities, as illustrated in Figure 3.1.

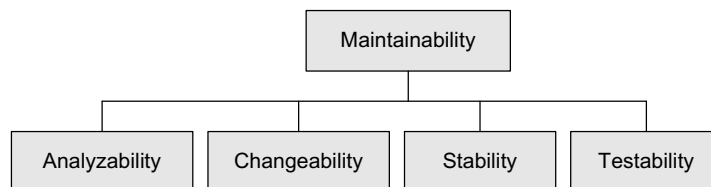


Figure 3.1: Maintainability and its sub qualities.

3.2.1 Analyzability

Definition: Analyzability (ISO 9126)

The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.

Analyzability is basically the ability to *understand* software. If I cannot analyze a piece of software, how am I supposed to modify it? Correct it? The problems with class X in the previous section are basically analyzability problems: the methods names `z()` and `z1()` do not hint at what behavior they encapsulate and the class name has the

same problem. In the second version I cannot even overview the method structure and the code is simply not analyzable.

The ability to read and understand the source code and the structure of our software designs is central to analyzability and I will touch upon it in many places. It will in particular be discussed in Chapter 7, *Deriving Strategy Pattern*, and Chapter 18, *Design Patterns – Part II*.

🔍 If you would like to see some real nightmares in analyzability, search the web for *The International Obfuscated C Code Contest*. Here people compete for fun to produce the most unreadable but correct programs in C.

3.2.2 Changeability

Definition: Changeability (ISO 9126)

The capability of the software product to enable a specified modification to be implemented.

Changeability expresses the quality that I can add, modify, or enhance a feature in my system at a reasonable cost. Let me again illustrate this by a simple example. Consider a program to generate and display a rectangular maze. The developers have decided for a 80 column times 25 row maze and have used these constants throughout their program.

Listing: chapter/maintainability/example3/Maze.java

```
public class Maze {
    private boolean[] isWall = new boolean[2000];
    public void print() {
        for (int c = 0; c < 80; c++) {
            for (int r = 0; r < 25; r++) {
                char toPrint = (isWall[r*80+c] ? '#' : ' ');
                System.out.print(toPrint);
            }
            System.out.println();
        }
    }
    public void generate() {
        // generate the maze
    }
}
```

Now consider that the users wants 160×80 mazes instead? Then all occurrences of the constant 80 has to be analyzed in order to evaluate whether it relates to a maze dimension or not (remember, there can be other uses of the constant 80 that are unrelated to the maze size!) and next change those that are related. Also some occurrences are masked and thus difficult to spot, like the constant 2000 in the declaration above that is really 80×25 . Of course, in a small example as above it is possible to do, but consider having a 3D graphics engine based on the maze with thousands of lines of code where the constants 80, 25, and 2000 appear in many places. *The changeability of such code is low* as it does not enable a size modification to be implemented reliably.

It is a well-known golden rule in programming never to use *magic numbers* but use named constants. Named constants makes it much easier to change the software to new requirements, like

```
private static final int MAZE_WIDTH = 80;
private static final int MAZE_HEIGHT = 80;
private boolean[] isWall =
    new boolean[MAZE_WIDTH * MAZE_HEIGHT];
```

Changeability basically comes in two flavors. Those aspects of the software that can be changed at compile-time, by modifying the source code, recompile it, and release it; and those aspects that can be changed at run-time, by changing parameters while it is executing. The maze size problem is of course a compile-time change while an example of run-time changeability is your favorite web browser whose “options” dialog contains numerous ways to change its behavior.

Changeability is a central quality in successful software and in particular for software that can be reused. Almost all design patterns are geared towards increasing designs’ changeability and I will in particular focus on the quality in Chapter 32, *Framework Theory*.

3.2.3 Stability

Definition: Stability (ISO 9126)

The capability of the software product to avoid unexpected effects from modifications of the system.

As I will emphasize many times, any change to existing software carries a risk of introducing defects: software that has run flawless for years may suddenly stop working properly because some apparently minor change has been made. Software units are often interconnected in subtle ways and changing one unit may lead to failures in other units. As an example, read the war story in sidebar 3.1. Of course we want to minimize this as much as possible, that is, keep the software stability high.

Stability is also a recurring theme in this book and I will advocate the practice to avoid modifying existing code but preferably add features or modify existing ones by other means. Chapters 7, *Deriving Strategy Pattern*, 8, *Refactoring and Integration Testing*, and , *Compositional Design Principles*, are in particular focused on stability.

3.2.4 Testability

Definition: Testability (ISO 9126)

The capability of the software product to enable a modified system to be validated.

Sidebar 3.1: Person Identification in ABC

I once participated in a research project, *Activity Based Computing* (Bardram and Christensen 2007), in which we developed a novel computing platform to support clinicians' complex ways of working in a hospital. In the first design of the platform, we equipped clinicians with RFID tags (small electronic tags that can be sensed by a RFID reader when about half a meter away) that were automatically scanned when they were near a computer thus allowing the computer to quickly log them in and bring up their personal list of ongoing activities. In our design, we had a database where persons were identified by the unique RFID identity string of the tag that they wore. Thus when a tag was scanned it was very easy to look up the person it belonged to.

Later we worked on other aspects of the system and one of our student programmers found that the identity strings used in the database were really weird, so he replaced them with a 10 digit unique person identity number. As the part of the systems he was developing was unrelated to the RFID functionality, and he never used nor tested this part of the system, he never noticed that this seemingly trivial change had serious consequences.

However, we did—during a demonstration for the clinicians! Suddenly none of the fancy functionality to automatically detect and login the physicians and nurses worked. It was not the best time to exclaim: “What? It worked the last time I tried it???” The system design had little stability concerning person identification...

You may wonder how you can build software you cannot test but it is actually quite easy as the following small example shows. Consider that you are required to build a software system to control chemical processes in a chemical plant. One requirement is that if the temperature measured in a process exceeds 100 degrees Celsius then an alarm must be sounded and some action taken. Consider the following code fragment for this requirement:

Fragment: chapter/maintainability/example4/Monitor.java

```
public class Monitor {
    private AeroDynTemperatureSensor sensor =
        new AeroDynTemperatureSensor();
    public void controlProcess() {
        while(true) {
            if ( sensor.measure() > 1000.0 ) {
                soundAlarm();
                shutDownProcess();
            }
            // wait 10 seconds
        }
    }
    // rest omitted
}
```

and consider that class `AeroDynTemperatureSensor` is the correct implementation that communicate with a temperature sensor from company AeroDyn. The question is how you would test the `controlProcess` method? If you review the code you will notice that it has a serious defect but how can you demonstrate the failure it will produce at run-time? As the code above is hard wired to the actual sensor, there is no

other way than go heating the sensor until it is above 100 degrees Celsius and then notice that the alarm has not been sounded. This is cumbersome and not something you would like to do as part of everyday development. The source code is not very testable.

A major theme of this book is how to make reliable software so techniques to test software and make software testable are central and I will in particular look into it in Chapter 5, *Test-Driven Development*, and Chapter 12, *Test Doubles*.

3.3 Flexibility

A main theme of this book is *flexible software* so I will start by defining what I mean by this term, give a small example, and next consider how it relates to the ISO characteristics.

Definition: Flexibility

The capability of the software product to support added/enhanced functionality purely by adding software units and specifically not by modifying existing software units.

What does this mean? Well, much of this book is about what this really means in a deep way, but to convey a first idea of it, consider your computer system. If for instance you experience that the graphics performance of it is not adequate for the next generation of computer games, then you may buy a better graphics card, open your computer, take out the old graphics card and replace it with the new. You do not do it by cutting wires and start soldering new circuits onto your motherboard. The same goes if you want to add functionality for your computer like being able to digitize and store video: you buy a digital video card and plug it into your machine. Thus, you enhance functionality by adding/replacing units and not by modifying the existing ones. By flexible software, I define software with the same merits—software systems are flexible if they allow adding and/or replacing units of functionality without rewriting and recompiling the existing code.

As a software example, consider that I have developed a point of sales system for customers in the states of California and Nevada. As these two states have different sales taxes, I have programmed a method `calculateSalesTax` to calculate the tax (2009 values, source: Wikipedia. Local taxes are not considered) to add to the base price of the item, as shown in this code fragment:

Listing: chapter/maintainability/example5/PointOfSale.java

```
public class PointOfSale {
    private State state;
    public double calculateSalesTax(double price) {
        switch (state) {
            case CALIFORNIA: return price * 8.25 / 100.0;
            case NEVADA: return price * 8.10 / 100.0;
            default:
                throw new RuntimeException("Unknown state");
        }
    }
}
```



```
}  
public enum State {  
    CALIFORNIA, NEVADA }  
    // rest of functionality omitted  
}
```

Next consider that we want to sell the point of sales system in a third state. While the above design for handling sales tax is analyzable, testable, and changeable, it is not flexible. The only way I can add a correct sales tax calculation for a new state is by modifying the existing code, recompile and test it, and finally deploy an updated system to the customer. If the sales tax percentage had been read from a configuration file at system start up instead, then the system had been flexible with respect to this requirement as I could have shipped the existing system to the new customer and just provided a new configuration file with the proper sales tax percentage for the state in question. If a state changes its sales tax then this is also much easier for the point of sales administrators as they just can change the configuration file locally instead of waiting for a new release of the system.

If you require that software can adapt to changing requirements without modifying the production code then you need to employ a special set of design and programming techniques as well as adopt a special mindset. The techniques covered in this book like compositional design, design patterns, and frameworks, are all techniques that focus on the flexibility quality, and treated in particular in Chapters 7, *Deriving Strategy Pattern*, , *Compositional Design Principles*, and 32, *Framework Theory*.

Returning to the ISO standard, flexibility can be said to be yet another sub quality of maintainability, and most closely related to changeability. However, changeability does not take a stand point with regards to the way “a specified modification” is implemented—it may be by adding or replacing a new software unit or it may be by modifying an existing one. In contrast *flexibility* does take this stand point and require that no modifications are made.

3.4 Summary of Key Concepts

Maintainability is an important quality aspect of most software products and is defined as the *ability to be modified*. Maintainability is basically a quality that deals with the cost of introducing change: maintainable software means software that is cheap to change, that takes less hours to modify.

Maintainability has sub qualities like *analyzability* (ability to understand), *changeability* (ability to implement specified modification), *stability* (ability to avoid unexpected effects from modifications), and finally *testability* (ability to validate modifications.)

How to write maintainable software is a key focus point in this book. A special case of changeability is a quality that I denote *flexibility*, namely the ability to support modifications purely by adding or replacing software units, in contrast to modifying software units.

Maintainability is just one of the qualities defined in the ISO model. The full model is defined in the ISO/IEC International Standard (2001).

3.5 Selected Solutions

Discussion of Exercise 3.1:

As maintainability is basically a “cost of change” metric, then maintainability is of little relevance for systems that are not required or expected to be changed. Sometimes I write a small program to change some data I have from one format to another. Once the data is converted, then the program is irrelevant, and maintainability is therefore of little importance. Large software systems that live on the market for a long time and where the development is made by many software developers have high requirements to maintainability.

3.6 Review Questions

How is *maintainability* defined? *Analyzability*? *Changeability*? *Stability*? *Testability*? Mention some software systems that do have or do not have these qualities. Argue why they have (or do not have) these qualities. Define *flexibility* and argue for the difference between this quality characteristics and the changeability quality.

3.7 Further Exercises

Exercise 3.2:

Review programs of some complexity that you have written earlier in your career.

1. Analyze each program and argue for the absence or presence of the following capabilities: Analyzability? Changeability? Stability? Testability? Flexibility?
2. Evaluate its maintainability based upon the assessment above.
3. List some program changes that would enhance maintainability. Argue why.

Exercise 3.3. Source code directory:

`exercise/maintainability/iceblox`

Iceblox is a small game in the tradition of early 1980 platform games like Pacman. It is an old Java applet meaning, so to run it, you will need to run it using “appletviewer iceblox.html”.

1. Analyze Iceblox and argue for the absence or presence of the following capabilities: Analyzability? Changeability? Stability? Testability? Flexibility?
2. Evaluate its maintainability based upon the assessment above.
3. List some program changes that would enhance maintainability. Argue why.

