Software Engineering Principles

Mark Bush

Sep 24, 2021

CONTENTS

This documentation contains a number of opinions of mine about Software Engineering. I make no claim as to anything being *right* or *wrong*. Opinions are, after all, just the ideas of a person. However, my opinions are based on more than 40 years of writing software and, more importantly, of reading other people's code. I provide this information purely in the hope that it will be useful to others.

Much of the information here can be found in numerous places around the Internet. Unfortunately, it is often difficult to determine the accuracy of information gathered this way. Also, much of it is not written well, or is not designed to be easily understood by someone new to the various concepts. I fully recognise that the information here might not be easy to understand, either, however I am more than happy to receive criticism that will allow me to improve the content.

There are many code examples here. I have chosen to use Java for these. It should not be too challenging to convert these examples to other languages.

CHAPTER

ONE

SOFTWARE ENGINEERING GUIDELINES

I have a personal list of *Rules of Software Engineering* as follows:

1. There are no rules!

Rules suggest very strongly that they should never be broken. By sticking blindly to rules, making decisions about application development based on a series of logical choices, can easily result in very complex code that is hard to understand and hard to maintain.

Instead, I think in terms of *guidelines*. These are principles which guide me, but don't control me. I can step back, look at the big picture, and choose when to follow them and when to ignore them (or, perhaps, adapt them). I usually require myself to have a pretty good reason for not following a particular guideline, though. Good software development is not anarchic!

1.1 Top-Down or Bottom-Up?

It seems that software engineers are pretty split when it comes to top-down or bottom-up development.

I plan to convince you that a top-down approach is far superior, however let's have a look at what bottom-up can achieve first.

1.1.1 Bottom-Up Development

A bottom-up approach tends to occur in one of two scenarios. The first of these is in pure research. In this area, you don't necessarily have a goal in mind, or even have an idea where your work is likely to lead you.

You start off with a number of known building blocks. You try various ways of combining these to see what the results are. Sometimes, something useful comes out of your endeavours. More often, however, the results are meaningless. Regardless, the outcomes allow you to focus progress.

Eventually, you build up a whole framework of concepts and tools for future research.

Another area is solving problems. You need a solution and it is not clear how to attain the goal, so you look at what solutions are already available to you and see how you can combine them to solve your original problem.

If you rely on existing building blocks like this, then the solution will be determined by what is already available. If there are a small number of components, then you will have very little freedom in how you can solve the problem (if, indeed, it is solvable with what is available). You will often find yourself using components in awkward ways to get them to help with the solution. This approach can also be overwhelming as you try and keep everything in your head so that you can work out how you are putting things together and how all the components will relate to each other.

If you have a rich and varied set of components, then you have a much better chance of finding ones that exactly suit your purpose, though this is not guaranteed. You are also likely to end up with plenty of components that were not necessary. If you spent a lot of time creating them, then that time will be wasted.

There is also the matter of testing. All these components need testing. The tests for the unused components are more unnecessary work and you are testing components without knowing yet what you need the components for. How can you be sure that your tests are appropriate?

If you just need an "adequate" solution to a problem to get something done as a one-off and can discard everything afterwards, then this approach can work out to be fairly useful.

In software engineering, if you build an application in this way by creating the low level components first, then you are restricting the way the higher levels of the application can work. By the time you try to compose the building blocks together, you'll often find yourself making compromises over how to plumb things together. This is not a good route to understandable and maintainable code.

It is very common, and natural, for software engineers to work in this way, though. After all, the high level code is just connecting things together. It is the small components where the interesting coding lies. When solving problems, we are often reminded of previous tasks (or even of *Design Patterns* that we are familiar with) and are sure that the techniques used will help in solving the current problem. We leap at the chance to get straight in to writing that low-level code.

I urge you not to get drawn into this approach. While working on these small components can feel very worthwhile, and can often make us feel great when we have solved a tricky little bit of coding, it is distracting from the real goal: that of creating a working, useful application for *someone else*. Software engineering is not about *you*.

1.1.2 Top-Down Development

The alternative is top-down development. This is the traditional *divide and conquer* approach which I like to refer to as *lazy programming*.

The idea here is that most problems to solve and applications to create are fairly large and too unwieldy for you to keep all in your head at once. You can easily feel overwhelmed by it. Instead of rushing in trying to work out the details, try to step back and just think in terms of what needs to be achieved.

If you need to process a collection of data, then a loop over the data calling a processing function on each item is all you need. The details of that function can be left until later:

```java
public void doProcessing(List<Thing> things) {
  for (Thing thing : things) {
    process(thing);
  }
}
```

If you need to achieve a number of tasks sequentially (say, A then B then C), then these just need to be method calls for those tasks:

```java
public void doProcessing() {
  doA();
  doB();
  doC();
}
```

If there are a lot of things that need to be achieved, then you should be able to group them. You then just need to achieve those phases, each of which will now be a small list of tasks.

With this approach, you are only creating components that you really need. Also, the interfaces to those components exactly fit how the higher level code wants to be able to call in to them.

This also lends well to good testing. Since you are defining what the lower level functionality needs to achieve before writing it, then you know exactly how it should behave. You can write tests before writing the code to ensure that it meets those requirements. This is the heart of Test Driven Development and means you are creating *exactly* the components you need and have confidence in them providing correct functionality.

Eventually, you will certainly have components that do actual work. Following this process through, however, will mean those functioning components will be very small. You'll usually find they write themselves since, by that point, there is usually only one way they could be written.

This may not seem very satisfying - code writing itself and, seemingly, no space for your creativity to flourish. Remember, though, that your goal is to provide a working solution for *someone else* that can easily be understood and maintained by others.

If people look at your code and say "wow, that's amazing, I could never create that" then they are not going to be able to easily maintain it. Instead, you want them to be saying "well, obviously that's what you'd do". That doesn't give your ego the boost that the previous comment would, but it tells you that they understand your code and could maintain it.

1.2 Lazy Programming

I have already suggested that *lazy programming* is the same as top-down development, however it is so much more. Most of us have a certain understanding of what top-down development means, but the name doesn't really help us understand exactly how to go about it. Or, rather, how to do it well.

The name "lazy programming" embodies a mindset. It represents a cultural shift in how you approach the whole concept of software engineering. It also has implications beyond mere "coding".

In one sense, being lazy is about doing the least possible work. However, it is also about efficiency. That is, using the least effort to achieve the required results. This is not just about short term goals. This is also about efficiency in the long term. Whenever you are working, there are always at least two participants. There is you, right there, performing the task. But there is also your future self looking back over your shoulder. There is no point doing something quick now if it results in them looking back at you in dismay.

To embrace laziness, you need to balance simplicity now without creating more work for yourself later. There are a number of ways in which you can do this. The most important of which is not to overthink what you are doing. Each function or method you write should contain the least amount of code necessary to convey the *intent*. It should read like a tiny story. Good choices of variable and function names will achieve this goal.

I have often been in the situation of having someone show me some code and immediately feel the need to explain it. This always suggests that the code is more complex than it needs to be. The same follows for code that someone has felt the need to comment significantly. If the code was more descriptive in terms of *what* it is supposed to be achieving rather than coding *how* it will produce the results, that both of these situations can be avoided for all but the lowest levels of the code. In fact, the remaining code in those lowest levels will then be small enough that extra comments or explanations will still not be necessary.

1.2.1 Code Performance

One argument that often crops up for writing complex code with larger functions is the need for performance. I think there are always three questions that must be asked by anyone taking this approach:

1. Is the code being written guaranteed to still be a part of the final version of the application?
2. Is the complex code guaranteed to perform better than more structured code?
3. Are there specific performance requirements for the piece of code that can only be met by manually optimising it?

The first question is important because the content of an application changes over time. You can't be certain of exactly what will be needed until the application develops and the intended users give feedback. It is not uncommon for ideas about what is needed or how that will be implemented to change and for some code to be thrown away. What a waste if you have spent weeks trying to get that code to run as fast as possible!

The second is more subtle. It is not always clear how a pice of code will run and simple tests often don't take into account the actual environment in which the application will run. For example, if a piece of code will run many times, the JVM can update on the fly by inlining suitable code which will increase performance over time without needing to restructure the code. This process that the JVM goes through as it gathers runtime statistics to be used for restructuring is called "warming up." There is no point manually inlining code that the runtime would inline for you automatically. If the code will not run much during production, then it is less likely that manual optimisation will make much difference to execution time. Other languages also have similar functionality where code can be automatically inlines either at run time or compile time.

It is very rare with modern compilers that humans can do a better job at increasing performance by tweaking the code. In practice, such improvements are more likely seen through better choices of data structures than actual code. The more code that is produced which is purely functional (that is, where the result only depends on the function arguments), the easier it will be for the compiler or runtime to inline code and increase performance automatically for you.

In general, the best approach here is to write code for humans as a priority. People (often including yourself) are going to be returning to the code in the future in order to fix bugs or extend the functionality and their task will be easier if the existing code is well structured and obvious in its purpose.

Once an application has been developed and is working correctly, then analysis tools can be brought to bear on it in order to highlight any areas of concern. Manual optimisation efforts can then be focused on just those areas where the analysis has shown that automatic processes cannot meet the required performance requirements.

1.2.2 Getting "In the Zone"

It's very common to hear people talk about being "in the zone" while working on some code. This is often accompanied by stories of ending up working through lunch or late into the night. The feeling is often that when you are "in the zone" then you want to make the most of it and keep going.

Most people consider this to be a good thing and that it means they are being productive. While it may be true that they are writing lots of code, it doesn't follow that they must be writing *good* code. It usually means that the person is currently juggling lots of aspects of an intricate piece of code and doesn't want to take their attention away in case they lose track of what they are doing. That suggests they are very much thinking in terms of *how* the code will work, rather than *what* it is intended to achieve.

In my experience, the results are usually complex pieces of code that are not obvious in what they do. The argument that it will be refactored later is a very poor one. As we all know, work has many pressures and the first thing to suffer under the strain is time to review and clean up code.

It is much better to be creating clean code right from the start. Think about *what* you want your code to achieve rather than *how* it will achieve it. When you are writing a function, either it will be immediately obvious what the content needs to be, or you will be able to describe what it needs to do. In the latter case, you just need to create code that describes what you want to achieve and you are done. Make each part of the description the name of another function. You'll have successfully split the code into smaller chunks that are easier to think about.

The most important aspect of all of this, in my opinion, is that when you concentrate more on the *what* rather than the *how*, then you are more able to take breaks. Having breaks from coding sessions is extremely important. Furthermore, you are much less likely to be building technical debt for the future.

CHAPTER

TWO

DESIGN PATTERNS

2.1 What are Design Patterns?

Whenever we find ourselves solving problems, we notice that some types of problem occur again and again. Over time, we begin to recognise these repeating *patterns*. We can then either directly reuse the solution we remember, or adapt it slightly for our needs. These common patterns have become known as *design patterns*.

Some patterns are so simple and obvious that we don't think of them as such. For example, naming variables according to the *role* they will play. Others, we might think of as simple *tips and tricks* that we have picked up over time. Many more lie as yet undiscovered. Over time, they will be discovered and move from the unknown, to the known. If used often enough, they may even be remembered!

There are a couple of key things to remember about design patterns.

Firstly, never force a pattern onto a problem. Even if the problem looks identical to the description of the pattern. Check that the solution makes sense in context. That it will be easy for someone to come along later and understand. When used inappropriately, they become *anti-patterns*, making code less understandable and harder to maintain.

Secondly, using patterns is a key to lazy programming (a most desirable thing). Whenever you need to solve a problem, your first thought should be "is it likely that something like this has been solved before?" If so, then there is likely to be either a language feature, framework, application component, or pattern that will provide your solution for you. Reuse existing code or ideas where possible.

I will not list all design patterns you might come across here. I intend to stick to ones that usually have challenging descriptions or where it is not clear exactly how they might be related to typical situations.

2.2 Inversion of Control

A typical piece of code might have the general structure:

```
public void doTheThings() {
  doThingOne();
  if (thingTwoRequired()) {
    doThingTwo();
  }
  doThingThree();
}
```

The code knows exactly what needs to happen and where that code is. This piece of code is in control of the whole process.

For precise requirements, this is perfectly fine, which is why most code looks like this.

If requirements start changing, then we can look for patterns in the changes. For example, suppose that the above code has to become this under certain situations:

```
public void doTheThings() {
  if (thingOneRequired()) {
    doThingOne();
  }
  if (thingTwoRequired()) {
    doThingTwo();
  }
  if (thingThreeRequired()) {
    doThingThree();
  }
}
```

This is starting to look a mess! We can see a pattern emerging, though. Clearly, there are a number of steps that need to be achieved and mechanisms for determining which ones are needed. If this code continues to be in control, the problem will escalate. Instead, we can view the steps as components in recipes. The conditions are selecting which components form the specific recipe we need to follow in a given situation.

We can rewrite this as:

```
public void doTheThings(Recipe recipe) {
  for (Step step : recipe.steps()) {
    step.perform();
  }
}
```

We are no longer in control of which steps to perform. That has been given to the layer above which is, presumably, responsible for either creating the recipe, or selecting one from a known collection of them.

This code is still going to perform the same function. It no longer controls precisely what or when. Control has been *inverted*. The result is that it is much less likely that this code will ever need to change again. Changes will be related to new recipes. We could imagine some managed mapping of situations to recipes which we might be able to externalise, turning into application configuration.

This is a specific example of how we can use inversion of control in a very specific situation. (If we found ourselves doing this on several occasions, we could consider it a pattern and call it the *Recipe Pattern*.) There are a number of well known design patterns which use inversion of control and we'll discuss some of these individually here:

- *Dependency Injection*
- The *Factory Pattern*
- The *Strategy Pattern*
- The *Template Pattern*

2.2.1 Dependency Injection

The whole purpose of object-oriented programming is to have a population of objects that communicate with each other as they collaborate to perform a task. At some point, those objects have to be created. This can lead to code like this:

```
public class ThingSupport {
  public void doTheThing(Thing thing) {
    ThingHandler handler = new ThingHandler();
    handler.process(thing);
  }
}
```

When the objects being created, and the requirements of them, are simple, then this is perfectly fine. It is not long, however, before we decide that we need to use an object of a different, but related, class to do the work. The thing is, we don't actually *care* what does the work, as long as it gets done.

At the application level, we might only have one option available to us, so it might seem reasonable to reference it directly like this. If we want to replace the handler here, it is not so difficult. But what if this handler is used in several places? We have created very tight coupling between our code and the class that can do what we need. If we change our mind, then we could have a number of places where we need to make changes.

A first step in changing this would be to provide the object in question as an argument:

```
public class ThingSupport {
  public void doTheThing(Thing thing, Handler handler) {
    handler.process(thing);
  }
}
```

This code is no longer in control of determining exactly what class of object to use. It can be reused with multiple handlers as necessary. The specific handler has been *injected* into the function. (This is a very common option when the value to be injected is a function!)

If it makes sense for the caller to know what specific handler to use, then this works well. But what if the caller has no idea about handlers? Then all we have done is move the coupling.

In this situation, it is usually the case that there is only one choice of handler anyway. But it may need to change later. For example, suppose the handler is currently interfacing with files, but later will use a database. Eventually, it might use a web service. Each of these implementations would probably have class names describing their nature - we wouldn't be likely to just replace the internal implementation of the single handler class.

What we need is some global configuration that says *when we need a handler anywhere,* **this** *is the current class we are using*. There are two common ways of implementing this. One is very similar to the *Factory Pattern*:

```
public class HandlerProvider implements Provider {
  public ThingHandler thingHandler() { return new FileThingHandler(); }
}

public class ThingSupport {
  public void doTheThing(Thing thing) {
    ThingHandler handler = handlerProvider.thingHandler();
    handler.process(thing);
  }
}
```

The difference to a factory is that there is no provided information to specify which object to build and return. The selection is done via distinct methods for each outcome.

The `HandlerProvider` is a configuration file written in code. We can easily change implementations used by the rest of the application. We should be careful to ensure that this configuration can easily be replaced for testing, so that we can control specific objects to be used by code under test.

Another common way is to use automatic injection. We can do this because this is such a common scenario that the Java language has built in features to support it. For example, we can declare that we need a handler, without needing any reference to *how* we obtain it as follows:

```
import javax.inject.Inject;

public class ThingSupport {
  @Inject private ThingHandler handler;
  public void doTheThing(Thing thing) {
    handler.process(thing);
  }
}
```

This mechanism is only suitable if we really don't care about the item being injected and just want to use it as some opaque service provider. Otherwise, the code becomes obfuscated and we feel we need to keep looking at the implementation of things in order to understand what is going on.

Several frameworks exist to manage this for us and they may have their own annotations to ensure successful implementation (such as `@Autowired` in Spring). These frameworks often allow us to give extra hints in case we need different implementations in different places of our code.

We can often configure these frameworks at run time using an external file. This has the advantage that we can run the same application in multiple contexts where different classes would be needed to provide the same functionality. The disadvantage is that the configuration is now inaccessible to the compiler, so we no longer have compile-time checks that we are specifying things correctly. A typo in the configuration will result in a runtime error which might occur much later and in a way that is hard to debug.

2.2.2 The Factory Pattern

We often find ourselves writing code like the following:

```java
public Response process(Thing thing) {
  if (thing.isOne()) {
    return doOneThing(thing.getOneValue());
  } else if (thing.isTwo()) {
    return doTwoThing(thing.getType(), thing.getLocation());
  } else {
    doOtherThing();
    return new Response();
  }
}
```

This function has clearly got things in hand. It is even delegating responsibility for getting things done to other functions. Plenty of boxes being ticked. And yet looking at it raises the hackles. You get a sinking feeling in the pit of your stomach as you realise this is going to end up a maintenance nightmare. You can feel your future self looking over your shoulder with dismay.

Well, firstly, we can recognise that the things to do are each a separate responsibility. They each deserve to be encapsulated in their own class. That will clean up the rest of this class, but not the above method. What we need to do is delegate the *structure* of the method. A perfect way to achieve this is with a factory.

A factory is something which, as its name suggests, creates new things. It is given specific requests and knows exactly how to create the right thing for that request. Note that a factory doesn't actually *do* anything (other than create things). It doesn't make any assumptions about what you want the things for. It just builds them for you.

For small applications, or when the factory is used at the top level to control the entire operation, it is common for the factory to be implemented as a singleton. In larger applications or where the factory may be required in lower level code, it is much better for the factory to be replaceable to help with testing. We can then use *injection* to make it available.

A typical factory contains conditionals that make use of the input to determine which object to create and return like this:

```java
public class ThingHandlerFactory implements ThingFactory {
  public ThingHandler thingFor(ThingEvent event) {
    switch (event.type()) {
      case ThingType.ONE:
        return new ThingOneHandler(event);
        break;
      case ThingType.TWO:
        if (event.isA()) {
          return new ThingTwoAHandler(event);
        } else {
          return new ThingTwoBHandler(event);
        }
        break;
      case ThingType.THREE:
        return new ThingThreeHandler(event);
```

(continues on next page)

(continued from previous page)

```
        break;
      default:
        throw new UnmatchedThingException(event);
        break;
    }
  }
}
```

Note that it is fine for the factory to make specialised things if the situation merits it. The factory should not need extra information for this. If it did, that would suggest the distinction should be the responsibility of another layer. The output of a factory should only be determined by the input.

The original code would now look something like this:

```
public Response process(ThingEvent thingEvent) {
  ThingHandler thingHandler = thingHandlerFactory.thingFor(thingEvent);
  return thingHandler.process();
}
```

The code is no longer in control of what is done or how. That control has been delegated to the factory and the handlers. It is just responsible for ensuring that something *is* done. Your future self is looking much happier. Is that a thumbs-up you're getting?

As for the factory, now that the selection control is encapsulated, you can choose other ways to implement it, if appropriate. For example, you could have the mapping from event type to class in some configuration file.

2.2.3 The Strategy Pattern

Often, we have a problem that is very similar to a generic one where there is a reasonably known way of solving it. We just need to adapt that solution a little.

We then discover that we have a second situation that needs a slightly different adaptation. Our solution might start looking like this:

```
public class ThingProcessor implements Processor {
  public void process(Thing thing, ProcessController controller) {
    doPreparation(thing);
    if (controller.isTypeA()) {
      doPostPrepA(thing);
    } else {
      doPostPrep(thing);
    }
    for (Step step : thing.steps()) {
      step.perform();
      if (!controller.isTypeA()) {
        doExtraStep(step);
      }
    }
    if (controller.isTypeA()) {
      finishProcessing(thing);
    }
  }
}
```

Yuk! The class is definitely in control, but it is micro-managing. This is certainly not scalable. Fortunately, we can see a pattern emerging. This time, we can keep the core of the structure in this class, and implement the adaptations in subclasses. We just need to add callouts at all places where we need to adapt:

```
public abstract class ThingProcessor implements Processor {
  abstract protected void doPostPrep(Thing thing);
```

(continues on next page)

(continued from previous page)

```
  abstract protected void doExtraStep(Step step);
  abstract protected void finishProcessing(Thing thing);
  public void process(Thing thing) {
    doPreparation(thing);
    doPostPrep(thing);
    for (Step step : thing.steps()) {
      step.perform();
      doExtraStep(step);
    }
    finishProcessing(thing);
  }
  protected void doPreparation(Thing thing) {
  }
}
public class GeneralThingProcessor extends ThingProcessor {
  @Override protected void doPostPrep(Thing thing) {
  }
  @Override abstract protected void doExtraStep(Step step) {
  }
  @Override abstract protected void finishProcessing(Thing thing) {
  }
}
public class ThingAProcessor extends ThingProcessor {
  @Override protected void doPostPrep(Thing thing) {
  }
  @Override abstract protected void doExtraStep(Step step) {
  }
  @Override abstract protected void finishProcessing(Thing thing) {
  }
}
```

The controller is no longer required (in this scenario). It is used by the caller to determine which of the specific processors to use to process the item.

Each of the subclasses is a different *strategy* for solving the problem. Sometimes, we can determine precisely which one to use in a given situation, as above. There are times, however, when we might want to run several strategies to see which is best before doing something with the results.

For the Strategy Pattern to be meaningful, the implementations of all of the abstract methods need to be relatable in some way. A particular class encapsulates all of the information about a particular way of adapting the solution. If the implementations of these methods are independent of each other, then we may end up with the number of strategies growing exponentially. In that case, we would be better off using the *Template Pattern*.

2.2.4 The Template Pattern

With the *Strategy Pattern*, we saw that we can easily delegate components of a process to subclasses if those components are related to each other. Suppose, however, that we need to do something like the following where the processing at the start and end are independent of each other:

```
public abstract class ThingProcessor implements Processor {
  abstract protected void doPreProcessing(Thing thing);
  abstract protected void doPostProcessing(Thing thing);
  public void process(Thing thing) {
    doPreProcessing(thing);
    for (Step step : thing.steps()) {
      step.perform();
    }
    doPostProcessing(thing);
  }
}
```

If we created a whole load of subclasses for all the available combinations of options, then we could end up with an explosion of classes and an enormous amount of repetition. We couldn't factor it all out in a hierarchy because we might need all possible combinations. The code selecting the subclasses would also become unwieldy, too.

Since the calling code would need to be selecting which subclass to use, it knows something about the pre-processing and post-processing necessary. It can then select appropriate implementations and pass them in like this:

```
public class ThingProcessor implements Processor {
  private final PreProcessor preProcessor;
  private final PostProcessor postProcessor;
  public ThingProcessor(PreProcessor preProcessor, PostProcessor postProcessor) {
    this.preProcessor = preProcessor;
    this.postProcessor = postProcessor;
  }
  public void process(Thing thing) {
    preProcessor.process(thing);
    for (Step step : thing.steps()) {
      step.perform();
    }
    postProcessor.process(thing);
  }
}
```

Now, each type of processing only needs to be implemented once and we can mix and match as necessary. The items to fill in are holes in a *template* which are filled in by anyone wanting a concrete version of the control flow.

This technique is common when the components are functions being passed in. This is the heart of functional programming!

CHAPTER

THREE

MANAGING SOURCE CODE

Whether you are working in a team on a project, or just creating a simple application for personal use on your own, you will benefit from good source code management.

3.1 Source Code Repositories

The key reasons for using source code repositories are:

- protection from accidental loss;
- easily sharing the latest versions of code;
- tracking changes (including *why* things changed);
- being able to refer back to earlier versions of code.

The first of these covers the situations where you might lose a laptop or your computer might become damaged somehow (even if that is just a disk drive corruption). It also covers the situation where you inadvertently delete files you didn't mean to. For this to be meaningful, the repository should be on a different computer system and it should be secure and backed up.

If you are working with a team, then the repository will need to be available on some sort of central system that is readily accessible to everyone. If everyone works for the same company, then this can easily be an on-site server. Otherwise, it may need to be "in the cloud" for maximum availability.

I have seen many applications where almost the entire project has been completed and then added to a repository in one go with the log message "initial commit". This doesn't help anyone to understand the parts of the system and means that everything was at risk of loss while it was being developed. Similarly, if code is only ever updated in the repository when there are significant changes (for example, the implementation of an entire feature), then log messages will not really say anything about when actual code changes have occurred.

In general, it is best to be committing code often and regularly to a repository. When you write a test, commit it. When you write a function, commit it (even if the test is still not working). If you feel that you no longer need a piece of code, ensure there is a commit of the existing code, then delete the lines and immediately commit. At least then you can easily track the change down and recover just the deleted lines if you realise you did need them.

The result of all these little commits will be a log of changes that are actually meaningful. You will easily be able to refer back to any change and understand exactly what is was and why it happened.

There are (at least) three good reasons for wanting to refer back to previous code. Firstly, to remind yourself how you implemented something that you may want to repeat; secondly to make use of old code that you thought was not going to be needed; and thirdly, to remind yourself of something you tried that didn't work. This last reason should not be overlooked. It is not uncommon to forget a "mistake". Indeed, those new to a codebase will not know about things that went wrong in the past. Having a recorded history of things that have been tried (with comments about why they were removed) will help avoid trying the same things again.

There is a distinction between committing and pushing code. When you commit, you give a log message about what has changed and save that change locally on your own machine. When you push, you copy any local changes to the central repository so that they are secure and everyone else has visibility. Committing often gives you the granularity of meaningful log messages and recovery points and pushing regularly gives you the security of protection against problems.

I would recommend pushing code to the central repository at least once a day at the end of the day. Common times for problems to occur to a laptop are during journeys to and from your workplace. Having your latest code secure in the repository every day also ensures that everyone else has access to the latest code if something happens to you, such as an illness.

It is easy to forget to push your changes if you are distracted by meetings and lunch breaks, so I would also recommend getting into the habit of pushing before lunch. In fact, the end of any coding session would be a good time to push so that you don't forget.

CHAPTER

FOUR

TESTING

When creating code, it is clearly important that what we create is correct. Testing is the process by which we ensure this. There are two expressions which always raise alarm bells with me. These are "testing code" and "test coverage". This might be the opposite of what most people would think, so let me explain what I mean.

Both of these expressions tend to imply that the code comes first. That is, that there is existing code and that there needs to be tests to confirm it is performing as expected. This is especially true of "test coverage". The result is that we end up assuming the code works and we write the test to fit that assumption. If the test fails, then it is the test that is wrong and needs to be updated to fit what the code is doing.

While it is very much the case that a test can be wrong (and should, therefore, be changed to comply with expectations), if the code existed first, then the test is not providing any value. Sure, it might tell us if something changes in the future, but right now, it is not actually telling us if the code is correct. It just affirms the assumed truth of the code.

When striving for "test coverage", we usually end up with all sorts of contrived tests just to ensure that every line of actual code has a way of affecting some test case. This produces very fragile tests since the implementation details of the code may change and either the tests will break because the assumptions about how to exercise those specific lines of code have changed, or some of the tests will now be superfluous since they are now no longer testing aything in addition to other tests.

A much better approach is Test Driven Development. This is a process where we write the tests in advance of the code under test. It is very much misunderstood and I will try to explain the idea and purpose here.

Sometimes, we are faced with an application which is already completed. Or, at least, large parts of it are. For systems which have been created with time pressures, it is common for testing to have taken a back seat. There is then a scramble to create tests in order to have confidence in what was created. As I have argued, tests written in this way do not really provide this confidence if they are only being written to say they exist or to claim a certain level of coverage.

Since this is so common, I will also describe how we can approach solving this in a way which actually does give confidence in what we are doing.

4.1 Test Driven Development

4.2 Testing Legacy Code

INDEX