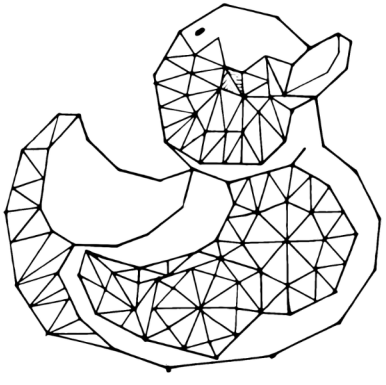
JAVA OOP
DONE RIGHT

Create object oriented code you can
be proud of with modern Java



Alan Mellor

Java OOP Done Right

Create object oriented code you can be proud of with modern Java

Alan Mellor

This book is for sale at http://leanpub.com/javaoopdoneright

This version was published on 2021-04-05

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To Reverend Mike Cavanagh
You've inspired me in more ways than you know.

You once said I would make a good consultant, "but I didn't have the stories yet".

It's taken a while, but here they are.

Contents

Preface

Why another book on OOP in Java?

I've been a writer on Quora for a few years now. I really enjoy it. I get to interact with people from all over the world about a subject I love - the craft of designing good software. I started learning this craft in 1981. I haven't finished yet.

This book came about after seeing a common thread amongst new Java developers on Quora. Many of them thought OOP in Java was 'verbose' or 'complex' or 'outdated'. And that it needed 'getters and setters' everywhere.

Yet we've had great books on Java, as well as classics on OOP.

So what changed?

Here's my theory. All the great books were written in the 1990s - before some of our modern Java developers were even born. The reason nobody knows the lessons in them is simple. We don't teach them anymore.

This book is my distillation of those classic ideas plus a twist of experience. Each chapter contains stuff I actually use, day to day, to get results. It's full of hard-won wisdom from 25 years at the code face. It's practical. And very simple. Bugs hide in complexity. I like to give bugs no place to hide.

My hope is this book gets you past 'getter and setter' coding and gets you into high gear using objects as they were intended to be used. It might be your first insight into how OOP fits together in the real world. Java has a reputation for being verbose. I hope this book shows you how to fix that. I want you to take away the techniques of crafting clean, powerful, readable OOP code.

This book is not an introduction to Java. It should be suitable for beginners who can write Java "Hello World" and understand the basic syntax for variables, conditionals and classes. Examples use Java 11 syntax. Many work in Java 1.

Alan Mellor
Rock Cottage
February 2021

Optimise for Clarity

Before we start, here's the big theme of this book:

OPTIMISE FOR CLARITY

Programming is all about explaining to another human what the computer happens to be doing at any given moment. As a by-product, it also tells the computer what to do.

The reason this is so important is that what the computer *actually* runs is almost incomprehensible to anyone.

As an example, if I write this:

```
1  clearScreen();
```

you can understand that I want the computer to clear everything off the screen, leaving it blank.

The computer can't though. It's as dumb as rocks. Literally. Silicon chips are made of Silicon Dioxide - Sand.

We need to pass this code through some kind of translator to turn this into the language of the computer. One that looks nothing like English.

One of my old computers would need something like this:

```
1    LD HL, 4000H  00100001 00000000 01000000
2    LD DE, 4001H  00010001 00000001 01000000
3    LD BC, 6143   00000001 11111111 00010111
4    LD (HL), 0    00110110 00000000
5    LDIR          11101101 10110000
```

Where the assembly language on the left was *still* a human-readable form. The binary on the right was what that computer ran on its Z80A microprocessor.

I know which one I like better.

The art of programming is making programs clear. Readable. Easy to skim read. No tricks.

It is the difference between having to read all that binary and work out that it meant ‘clear the screen’, compared to reading ‘clearScreen()’.

In a working professional team, nobody has the time, money, or desire to do that. Ever.

This book is all about helping you create code that both you and your colleagues can understand.

What is an object, anyway?

The best way to think about objects is as people at work. Each one has a specific job to do. They know how to do their job. They have all the knowledge, tools and supplies they need to do it.

Think about how a handwritten letter gets sent through the post.

We have a writer. They choose what words go in that letter, then write them down. This letter is then handed to a postman. The postman drives the letter to a sorting office.

The sorting office has inside it a big map of postcodes. On that map, each postcode has a little pin with the another postman's name, who is responsible for delivering the letters.

The sorting office hands over the letter to the correct postman, who delivers it to a reader.

So far, so dull. Just an everyday story about an outdated way of sending mail. But it contains the two key insights into Object Oriented Programming.

Notice how the writer *asks* the postman to deliver the letter. The writer does not tell the postman *how* to deliver the letter. That's not the writer's job. The writer writes. The postman delivers. They each do their own job without interference from the other.

The writer also doesn't ask the postman anything at all about how the delivery will be done. The postman is free to do that however they see fit. They can even change how they do the delivery later. The writer will not be affected at all.

This is the essence of OOP. Instead of people, we have objects. Each object knows how to do its own job.

You'll notice that this description is quite independent of programming languages. OOP itself is a design approach that can be done in any language - even assembler or C.

Languages like Java are called OOP languages because they provide direct support for writing code that reads like this.

What's all this got to do with Java?

As an object oriented language, Java was designed to reflect the real world, like in our example above. It allows us to write code that speaks about Postman and Letter and deliver() and so on. It allows us to write computer code that reads like an English description of our problem. Not like code.

We can write short pieces of code that can hold secrets, and present behaviours. Those short pieces of code can be asked to do things. The code representing what a postman knows and does can be asked to deliver a letter.

The way Java does this is in two parts.

First, it provides a 'traditional' computer language. It has variables, conditionals, loops and all the other good stuff we think of as code.

Then it gives us a way to package up that code to represent real world concepts.

These packages of code and data are called 'objects'. Each object has a certain type - like "this object represents a postman", or "this one is a user". We call these Classes in Java. All objects belong to a class, which tells you what they represent. You can have any number of objects of the same class. You can have any number of classes.

A Class represents an idea. It can create multiple objects. Objects can store their own data. They provide methods, which are small chunks of code that can be called by a program. These methods represent the behaviours we talked about. Things like 'deliver a letter', or 'check spelling' on a word.

Methods can describe absolutely any behaviour we can think of. This is the super-power of object oriented programming.

Confusing? Yes. It is at first. There's a lot to learn.

The easiest way to understand it all, I think, is by the 'User' example in the next section. But let's introduce some terminology and Java syntax before we look at that.

Classes, methods, constructors, objects, references

Object An object is a small bundles of methods and data, used to represent a single, specific thing: a person, a train, a word, a product.

Class In Java, all objects belong to a class. A class allows one or more objects to be created. It is a blueprint or a template for the common features of each object.

Method A method is used to describe the behaviours our object gives us. It is rather like a function that is specific to a class.

New Keyword new takes a class name, then creates a specific object. It returns a reference to that object that we can store. This lets us use that object later in our code.

Constructor A special method that new will call. It is responsible for setting up the object as it is being created. We can supply initial values of things here.

Fields These are variables that are unique to each object. To help our behaviour methods do their work, we can store data in our object in fields.

Object reference keyword new will create an object, set it up with a constructor, then return a 'reference'. We can store this reference in a local variable, or in an object field. We can then call methods on that object, using the reference and the dot operator.

this Inside a method, keyword 'this' is an object reference that refers to the 'current object'. It is the same idea as when I talk to you and say 'me'. I am referring to myself.

private A keyword to mark a field or method as being usable only inside the Class. Outside, it is invisible. It is used to mark 'secrets'.

public A keyword to mark a method as being usable from outside the Class in calling code. Can be used on fields, but that is rare.

A simple example: Greeting users

To show what we mean, let's make a simple object: a User.

Most systems have users. Hopefully, we'll have loads of users - if our marketing works right. Our job is to show a personalised greeting to every one of them.

The first design decision is to represent each user as an object. And to do that, we'll need to code up a 'blueprint' for what all user objects have in common. A Java 'User' class:

```
1  public class User {
2    private String name ;  // 3. private data
3
4    public User( String name ){. // 2. constructor
5      this.name = name ;
6    }
7
8    public void greet() { // 1. greet method
9      System.out.println( "Hello, " + name );
10   }
11 }
```

Before we can do anything with them, we need to create a simple test application. We'll use the standard bit of Java boilerplate code to create an Application class with a 'static main()' - That's a 'magic method' that tells the operating system where to start our Java program.

```
1  public class GreetingsApplication {
2    public static void main( String[] commandLineArgs ) {
3      User u1 = new User( "Jake" );
4      User u2 = new User( "Katy" );
5
6      u1.greet();
7      u2.greet();
8    }
9  }
```

Running this program shows us 'Hello, Jake' and 'Hello, Katy'. It does this by creating two, separate user objects and asking them to greet the user they represent.

This code might look underwhelming, but it contains the most important basics of OOP. Let's go through the key pieces.

Methods - Making objects do stuff

The most important part about each object is what you can ask it to do: the public methods.

Class User has a single method on it called 'greet()' (see 1).

The method name tells us *what* the method does, not *how*. When we write the calling code, this name describes what this method will do for us. It also insulates us from having to care about how this gets done.

This is important because it lets us change the internals of method greet(). When we do, there will be no change at all to the calling code.

Our test app creates two user objects and calls the greet() method on each of them. Notice how close to plain English the test app reads. OOP is all about designing customised objects that do exactly the right thing for our app. We can name methods the same way we talk in English about the problem we're solving.

When we design objects, we start with behaviour methods. We add data and logic later, if it is needed to make those methods do their job.

Secrets - Specialist knowledge for our objects

Just like the people in our example know things and have tools to help them work, objects have their secrets, too.

Objects typically have two secrets:

- Data - which is unique to each object
- Algorithms - the logic of how work is done

Our greet method has both kinds of secrets.

The logic secret of greet() is simple. We use Java's System.out.println() library method to write text to the console.

This is the *how* - how our method greets a user. Because it is hidden behind the greet method's signature, we are free to change how we do this without affecting

the calling code. This is important. This stops changes from 'rippling out' through the system. That makes the code easier to understand and safer to change.

This is also why the calling code looks so simple. It is not concerned with deep technical details. It just asks for what it needs doing - user.greet(). It leaves it up to the object to get the work done.

*This is called the **Tell Don't Ask** principle*

We tell the object what we would like it to do for us. We don't ask it for any of its secrets and try to do its job ourselves in the calling code. It is a huge advantage in simplifying our code.

Our object also has the other secret - data.

For our greet() method to work, it will need to know the user's name.

We decided upfront that each User object will represent one individual user in the real world. That user has a name. So our user objects make the perfect place to store their name. We do this in the private field called 'name' (see 3)

Having just one String field can be confusing to OOP beginners. How do you store the names of all the users without an array or something?

Non-OOP code might well have an array or dictionary to store all the names of the users. OOP code splits this problem up a different way. As we have one object per user and only store one name for each user, our object only needs to store one name.

Instead of one array with many names, we have *many objects* each with only one name. It's less to think about and a lot more obvious. Where can we find a user's name? In their object.

This idea of self-contained objects makes Object Oriented programs simpler to understand.

Constructors - Getting objects ready to use

As we create one object per user, we want to make sure that the object is ready to use after creation. This avoids many problems of 'uninitialised data' that plague other approaches.

We have a special method to do this: the Constructor.

The big idea of a constructor is that it creates an object, loads it up and makes it ready to use.

For our User objects, the private 'name' field (see 3) is set inside the constructor. We pass each individual name into the constructor as a parameter.

Constructors help keep our private data secret. They provide a way for the calling code to set up an object without knowing anything about its secrets.

The big idea: calling code is simple

Now we understand how the User class has been coded inside, let's look at the big win of OOP: the public interface. The part that the calling code sees.

It's really simple.

```
1  User u = new User("Alan");
2  u.greet();
```

To write a program that knows how to greet a user, all we need to know - and call - are the two public pieces. We call the constructor to create the object and get it ready for use. Then we call greet(). It really couldn't be any easier.

Now, imagine how this helps us as we grow bigger programs. We might have hundreds or thousands of classes. Maybe tens of millions of lines of code. But with OOP, we only need to understand what our object's public interface can do. It insulates us from the details.

This is the key of OOP done right: *the calling code is simple*

And if it isn't ... we're doing it wrong.

Object Oriented Design is Behaviour Driven Design

The key to doing OOP right is to let behaviours drive the design.

The first question is always ‘what does this object need to do?’. We represent that as a method, using the name to describe the behaviour.

Then, we can add supporting code inside that method to make it work. We might add calls to private methods to break the work into small chunks. We might need stored data as fields in the object. We would need a constructor in that case.

Clean Code

Whilst small, the code in our previous example was also 'clean'. Clean Code simply means code that is easy to read, easy to work with and safe. Safe here means 'no gotchas' about using the code.

This means happier days, faster development, fewer bugs. If ever there was a secret sauce to development, then clean code is it.

There are several techniques used here that we should always use in code.

Good Names

Names are absolutely critical in software. Let me give you an example. What does the class below do:

```
1  class Z2 {
2      private String aa;
3
4      Z2(String aaa) {
5          aa = aaa;
6      }
7
8      void q() {
9          System.out.println("Hello, " + aa)
10     }
11 }
```

That's right - it is our User class, with terrible names.

Hard to understand, wasn't it?

Did you find yourself reading the code, "playing computers in your head" and reverse-engineering the meaning out of it? Did you have thoughts like "Ok, this is a

Z2 class. It has a string called aa which gets set, and we can call the q method, which … hang on … says "Hello" … is this like a greeting or something?"

The only text here that gave us a clue was "Hello". In our better named User class, we had the words user, greet, name and hello which helped us understand what that class was about. We didn't need to understand every detail of the code.

Choosing *intention revealing names* massively speeds up your team's ability to read your code. Not a little bit. Massively. And it helps you - when you come back in six months to code you've long forgotten about.

Method names: Tell me the outcome

Effective method names explain why I should call them - what will have happened after that method completes. They should not explain what is inside the method. Only what that method can do for callers.

GOOD: Users findNewUsersSince(Date date)
BAD: Users runDatabaseQueryToSortUsersByJoiningDateThenMap(Date date)

The method name *should be the comment* for a block of code. The code itself explains how the method works. The method name serves as a comment explaining what the method does if I call it.

Oh yes; don't use comments for this. Turn comments into method names.

Variable names: Tell me the contents

Effective variable names tell me what their contents are, not what they are made of.

GOOD: String surname // if I need a surname, I can use this
BAD: String theString // yes, yes, I can see that! Why is it here?

Follow these two rules **methods say what they do**, **variables say what they hold** and straight away, most of your code will be easier to understand.

Design methods around behaviours, not data

A lot of texts suggest that the important thing about OOP is the data that a class has. This confuses a lot of people and leads to some brittle designs. The critical thing

is ‘what behaviours should the class have?’. Answer that before deciding what data those behaviours might need.

In the past, I have viewed an object as being a stucture of data with some functions wrapped around that. I have learned that designing behaviours first, without any knowledge of how they will work internally, leads to more stable OO designs. I recommend you do the same.

Designing around behaviours is key to doing OOP right.

Hidden data - No getters, no setters

Speaking of data, our greet() method will need some. It needs the name of the user to do its job. So we provide a private ‘name’ field for that purpose.

But *only* greet() needs to know that. Nothing else does.

There is no access to the name from outside the class.

No getName(). No setName().

The User class simply has no need for that. We set the name in the constructor. We use the name as part of how greet() works, but it has no more importance. It exists for that one reason only.

This is what truly encapsulated fields look like.

Later on, you might find that a getter *is* appropriate. Just start with behaviours, start with fully encapsulated data and go from there.

That way, you won’t risk other programmers from writing code in other classes that lock us into private data.

Aggregates: More than one

Our last example followed in the footsteps of all great software by having just one User. Even Facebook started from there.

But what happens when we have more than one thing? After all, it is pretty much what computers were invented for.

Java provides many ways of doing this. Most of them result in hard to read, verbose code. So, over time, I've gravitated to just one key idea. It is from Domain Driven Design: Aggregate objects.

Greeting more than one user

Continuing our example, let's add more users and greet them:

```
1  public class Users {
2    private final List<User> users = new ArrayList<>();
3
4    public void add( User u ) {
5      users.add(u);
6    }
7
8    public void greet() {
9      users.forEach( User::greet );
10   }
11 }
```

This is an **Aggregate** class. It is called that because it aggregates more than one User object into a single Users object.

The nice thing about it is that it follows the same principles as before:

- Behaviours first: add a User, greet all users
- Hidden data - no getters, no setters
- Intent revealing names add() and greet(). No comments required

Internally, it uses some features found in Java 8 and above, which you will be using by now.

Type safe collections, not raw arrays

The private field uses the well-known Java Collection class framework. The field itself is of type List<>. I chose that to reveal my intention that I'm not bothered which exact kind of list is used to store the users, so long as it can access them in the same order that add() was called.

The next design decision is to use the generic form List<User>. This clearly states that you must store only User objects (or subclasses of User) in that List. That is reflected in the add(User u) method.

Type safety is important in large programs. Without it, you can accidentally add in something that is not a User object and that does not support the greet() method.

What would be the point of that? The code would crash when you ran it.

Java has strong static typing to reveal this kind of error *before* you run the program. It is a powerful way of preventing bugs from ever entering the code.

Designing-out possible mistakes is a fundamental principle I use throughout all stages of development. If you make something so it cannot go wrong, everything is much easier to understand. You can literally read out what it can and cannot do, rather than be forced to guess.

There is another way that this approach is used here.

Java also provides raw arrays to store 'more than one thing'. They look a bit like this:

```
1  // Raw arrays in Java. Just say no
2  User[] users = new User[10];
```

I've always thought that this is just too low-level for me. I used to program in C and assembler language, where this is your only choice. It is *so* much better to have the collection classes available.

Here is the problem with low-level array use: You have to know in advance how many things you want to store; ten in the example above.

With an array, adding them to the end of the list is awkward. You have to track the last index you wrote to, then write to the next one, using the users[currentIndex] syntax.

Once you reach the limit of the array size, because it cannot grow, you have to create a new array, copy over all the contents, then copy over the 'last written to index'.

It's almost as if there should be an *object* to encapsulate all of that work for you, isn't it?

And, of course, that's exactly what the ArrayList class does.

So use it. Do not write your own.

One reason people complain that Java is 'verbose' is when they see novice code littered with raw array management - when the collections should have been used.

Is our Users class verbose? *Really?* I don't think it is.

Using forEach - not a loop

Look at the greet() method on our Users class.

If you've never seen that before, it looks a little strange.

It uses a feature new to Java 8 called lambda functions. Basically, you can pass a method into another method as a parameter, making it a bit more like how JavaScript uses higher-order functions.

Each Collection class like List now supports a method called forEach() which takes exactly one method as a parameter.

It does exactly what it says on the tin, which is why I love it: for each one of the things in the collection, run the function on it.

In our case, an internal iterator loops over the list of User objects. For each user object, it will call the greet() method on that object.

We made that happen by the odd syntax 'User::greet', called a 'method reference'. This is a Java 8 shorthand for 'use the greet() method on the User class for each object'.

There are four different kinds of method references you can use. All are useful.

The key point is how *direct* the code reads. 'For each of our users, call greet()'. It is simple, clear and concise.

What I really like is how this compares to using an index based loop over a raw array. Let's compare that:

```
1  // Looping over raw arrays. Just say no, again
2  User[] users = new User[10];
3  users[0] = new User("Alan");
4  users[1] = new User("Katy");
5  users[2] = new User("Stephanie");
6
7  // Please don't code like this. Pretty please.
8  for (int i=0; i < 3; i++){
9    users[i].greet();
10 }
```

Does that work? Sure. Does it suck? Oh yes.

The main problem here - to me - is that all I can see is *mechanism*. Loads of detail about exactly how painful it is to run a loop with some array indices over some array of User objects that have a greet() method on them, then we have to call greet()

What I *don't* see is my original 'business problem' - 'greet all the users'. It is clear in the aggregate object using forEach().

Another technical problem is with the loop end condition. I fudged it for the example. Can you see how the users array has ten slots? But then I only add three users. Can you see how I had to hard code the figure '3' into the for loop?

Again, arrays are just not the tool for the job. *They almost never are*. Do not use them where a collection class will do the job better.

Aggregate methods work on all the things

This simple rule helps split up your code into separate pieces.

- If the behaviour is on an individual object, put it there
- If the behaviour applies to all objects, put it in the aggregate

In our example, the greet() method in the aggregate Users will greet all the Users. The greet() method in individual class User only greets that user.

Common themes

Some common ideas for aggregate methods are:

- summarise() - create a summary of some common element
- broadcast() - send something to all objects
- submit() - send off a collection of orders

How many more can you think of?

Collaboration

Objects, like people, work best when they work together.

Designing a set of objects to solve a problem is a craft, not a science. There are many principles and guidelines but no step-by-step plan. You have to craft each solution as you go, working through the various trade-offs that present themselves.

Later chapters will explore some of these principles in detail. For now, let's start with the basics.

Basic Mechanics

Java gives us two basic ways to get objects to work together. We can pass an object as a parameter, or we can have an object as a field.

Let's see them both in code, so we know what they look like. We'll use two classes, Dog and Cat. The Dog wants to chase the Cat. Perhaps not 'collaboration' between the two pets, but let's see how we could do it using objects.

We'll make class Cat have a chase() method:

```
1  class Cat {
2      public void chase() {
3          System.out.println("I'm off up that tree");
4      }
5  }
```

The first way to give the Dog class access to a cat object is to hold it in a private field:

```
class Dog {
    private final Cat cat ;

    Dog( final Cat c ) {
        this.cat = c;
    }

    public void chase() {
        cat.chase();
    }
}
```

Dog objects then have a reference to a Cat object, which they can then use in any of the Dog methods.

Another way is to pass the Cat object as a method parameter:

```
class Dog {
    public void chase( Cat c ) {
        c.chase();
    }
}
```

This allows the chase method on a Dog object to access the Cat object.

We can call methods on the object passed in

The fact that we can call methods on the object passed in confused me the first time I saw it. I had only ever seen primitive values like numbers and strings passed into methods before. You just consume them. But when you pass an object in, you can call any available method on that object.

It's important to note that both objects - Dog and Cat, here - have the same equal standing. Cat is not an "output parameter", as you might see in other languages. It is a fully-fledged object, with 'equal rights' as Dog. See this method signature as a true collaboration between equals.

How to decide: field or parameter?

You will see both of these techniques used. The key to choosing one over the other relates to the lifetime of the objects.

- If you need the same collaborator throughout the lifetime of the using object, choose *field*
- If you need different collaborators on each call, choose *parameter*

The idea here is that by passing in a Cat object as a parameter, you could - if needed - pass in a *different* Cat object on each call to Dog.chase(). Using a parameter tells the reader of your code that this is what you wanted to allow. This is short-term collaboration.

Using the field approach tells your readers that you are fixing on a single Cat for that Dog object's entire lifetime. This is a long-term collaboration.

Example: Simple Point of Sale

We'll build a couple of classes that let us print out a till receipt from a Point of Sale terminal.

We'll start our code with a simple object representing one line item that we've bought. As before, we'll work in an agile, iterative manner. We'll grow objects out from behaviours we want them to have.

Behaviours first

```
class Item {
    public void print() {
        // todo
    }
}
```

First things first, we'll start with an Item that can print itself.

That's already making me think about the design. What should print actually do? We've gone for a void/void signature for now, as we're exploring the solution. But it's going to need access to something that can print in some way - unless it can do the job without help.

The way to decide is to have a clear vision of what we decide each object should be doing. That's our decision as software engineers.

Later chapters covering Test Driven Development and the SOLID principles will make this decision easier to make. But for now, let's decide that Item only knows *what* should be printed on the receipt and not *how* to print it.

This means that we need another object that does know how to print things in a bit more detail. Let's design the behaviours on that:

```
class Printer {
    public void print( String text ) {
        // todo
    }
}
```

For now, let's assume all we need is the ability to print a String value. We can leave the implementation as 'todo' and come back to this later.

We can give our Item a way to print itself now:

```
class Item {
    public void print( Printer p ) {
        // todo
    }
}
```

That's progress.

We are expressing the high level design and structure of our code to our readers. We are telling them that we have an Item object that can print itself. To do that, it will collaborate with a Printer object.

I want to stop at this point and underline the thinking here. All we have to show for our work is two classes of five lines each, where every method says 'todo'. I'm not expecting much bonus at this year's Annual Pay Review so far.

But this is the nature of OOP done right. It is why it is brilliant. We are designing the behaviours of the system first as *abstractions*. Abstraction means 'the general idea' or 'the thing that is always true, however you do it'.

We avoid getting bogged down in details of coding. Instead, we spend the time expressing *what* our system does, *where* each big idea splits apart and not *how* it is done. The how is, of course, very important. It is just less important than the structure of a program. Adding the 'how' can be done later.

Abstraction: the *essence* of what needs doing

If you have programmed in a lower level language, or are newer to programming, you'll probably recognise this as 'backwards' thinking. We are used to diving in with the details. I used to approach coding that way. I would dive right in with a big "shopping list" of detailed instructions.

This leads to hard to understand code. As we read it, we have to reverse engineer what is being done. By making our design explicit, we can directly read out the 'what'. It goes right back to the start of the book - our idea that 'clearScreen()' is a lot easier to understand than the binary code of how a screen is cleared.

Same here. We prevent that mess by designing OOP behaviours first.

Anyway, tea break over: back to the coding. We have only one Item on our receipt. Let's create an aggregate class that represents our full receipt:

```
class Receipt {
    public void print () {
        // todo
    }
}
```

The first behaviour I can think of is print().

Once again, we'll get back to the details of print after checking back with our requirements. But we know that it will collaborate in some way with our Item objects to print them out. We also recall our previous design of Item. It has a print() method that takes a Printer object.

Let's assume that the same Printer object is used to print out the entire receipt. On that basis, a reasonable design decision is to pass in a Printer object in the constructor of receipt and store it in a field for use:

```
class Receipt {
    private final Printer printer ;

    public Receipt( Printer p ) {
        this.printer = p;
    }

    public void print () {
        // todo - using field 'printer'
    }
}
```

More progress. Our receipt will be able to print itself.

But it needs something to print! We'll give it a way to add items on our receipt:

```
class Receipt {
    private final Printer printer ;

    public Receipt( Printer p ) {
        this.printer = p;
    }

    public void add(String description, Money price) {
        // todo
    }

    public void print () {
        // todo - using field 'printer'
    }
}
```

Another step forward.

I've made a design decision in the method signature of add(). It is based on the requirements. The method takes two value parameters - the item description as a String and its price as a Money object. I've chosen this over passing in an Item object. Maybe I'll change my mind on that later.

For now, the thinking is that Item will stay private to this cluster of classes to reduce coupling on callers. Coupling is the technical term for how connected different pieces of code are.

A Money class is needed, but it can be empty for now. The code only has to compile at this stage:

```
class Money {
    // TODO
}
```

We'll create a list of Item objects inside our receipt and add to them using add():

```
class Receipt {
    private final Printer printer ;
    private final List<Item> items = new ArrayList<>();

    public Receipt( Printer p ) {
        this.printer = p;
    }

    public void add(String description, Money price) {
        items.add( new Item(description, price) );
    }

    public void print () {
        // todo - using field 'printer'
    }
}
```

There's another design decision here. When we create an Item object, it now needs a constructor. We'll pass description and price into that, giving our Item object knowledge of what it needs to print:

```
class Item {
    private final String description;
    private final Money price ;

    public Item( String description, Money price ) {
        this.description = description;
        this.price = price;
    }

    public void print( Printer p ) {
        // todo
    }
}
```

After adding the constructor, I added the fields to hold the knowledge of description and price. I marked them as final to make this an 'immutable value' object. Immutable

means the value will not change. That makes systems easier to reason about - and safer to use in concurrent environments like web servers.

IDEs will add this kind of code for you

Working backwards like this, you give your IDE like IntelliJ all the clues it needs to offer to write the code.

From typing 'new Item(description, price)' in the Receipt class, it will offer to create a constructor in Item with the correct parameter names and types. From here, the IDE will offer to add the correct fields with correct names and types in Item.

Take full advantage of that IDE help! I always do.

We're getting there now. The next job is to fill in some details on this printing. Let's start outside-in, so we stick with big picture ideas first:

```
1  class Receipt {
2      private final Printer printer ;
3      private final List<Item> items = new ArrayList<>();
4
5      public Receipt( Printer p ) {
6          this.printer = p;
7      }
8
9      public void add(String description, Money price) {
10         items.add( new Item(description, price) );
11     }
12
13     public void print () {
14         items.forEach(item -> item.print(printer));
15     }
16 }
```

We've added a Java 8 lambda style line to print out all items. It uses the 'Tell Don't Ask' style to go through all Item objects and ask each one to print itself. We pass into that print method the Printer object we supplied in the constructor of Receipt.

This is an example of good collaboration design.

We're not pulling data out of objects using getters and setters. We're not even thinking of 'data'. Well, I'm not. I think of price and description as 'secret knowledge' that the Item is responsible for keeping to itself. Sure, that's a bit of word-play; but it helps me see OO design as being driven by behaviours and not pure data.

This is still a big picture idea. Receipt does not know anything about how Item objects look nor handle printing. It only knows that there are zero or more Item objects and that they can be asked to print. It's really quite a simple, descriptive way to create code.

The next step for me is to follow that chain of behaviour starting at print and stripe that through this slice. I like to build software as a series of working slices of code like this. At the end of each iteration, you don't have all the features. You don't have complete layers of the system either. What you do have is working software you can demonstrate to stakeholders. That's valuable.

Let's add the 'how' detail to the Item print() method. "At last!", you might think. But don't celebrate too long - it will pose another design challenge for us. Let's do the simplest thing first:

```
1  class Item {
2      private final String description;
3      private final Money price ;
4
5      public Item( String description, Money price ) {
6          this.description = description;
7          this.price = price;
8      }
9
10     public void print( Printer p ) {
11         p.print(description);
12         p.print(" ");
13         price.print(p);
14
15         p.newline();
16     }
```

```
17  }
```

One consequence is we need to add a newline() method to Printer. It should move on to the next line, the next time any print() method is called.

Can you see the design decision I'm talking about?

It is 'which object should be responsible for knowing how to format the Item on a line?'

Tricky. It's not a difficult choice, but we have alternatives to consider.

What I went with above was to print the description as a string, *assume* that one space can be printed to separate the price, then *delegate* responsibility for formatting price to the Money object. This means we need to add a print(Printer p) method to Money.

This method knows about the Printer class, and also knows how to format a Money object suitable for printing.

Code review/pairing time: What do we think?

I'm not sure.

For a simple system, a proof of concept, or an example in an early chapter of a book on OOP (ahem), this is adequate. It will work.

However, it has a number of code smells to me:

- The visual formatting will look wrong
- Why would Money know about Printer? And Printer-specific formatting?
- Why is the knowledge of formatting one Item split over two classes?
- How hard is it to read the code and know what to expect?

The real heart of OOP collaboration design is this. So far, it's been all plain sailing. But concerns like this - that don't easily map onto any existing single object - are the ones where we have to stop and think.

There are two options at this point. We can just press on. Leave it as it is and revisit the choice later. Or we can improve the structure now.

It is easy to over-engineer at this point, adding complicated features and structures that we will never need and no-one else will ever be able to read. Later chapters will cover ideas for knowing when to stop. In particular, Test Driven Development will actively limit the amount of change we make per step. The YAGNI principle ('You ain't gonna need it') helps focus our mind.

But for now, let's take the opportunity to fix this - with another collaboration. As usual, behaviours first, details later:

```
1  class Item {
2      private final String description;
3      private final Money price ;
4
5      public Item( String description, Money price ) {
6          this.description = description;
7          this.price = price;
8      }
9
10     public void print( Printer p ) {
11         new ItemFormat(description, price).print(p);
12     }
13 }
```

We've introduced an object responsible for formatting an Item object for print. We've given it all the information it needs to do that job. As ever, we're leaving the details until later. The first cut of the object will be a simple refactor. We will put the insides of the previous Item print method into the ItemFormat print method. We'll then make those detailed decisions about who-does-what between description, Printer and Money later.

It's time to build out a small test app for this. First, we need to make our Printer class do something. We'll add that newline method. We will also decide to use console output:

```
class Printer {
    public void print(String text) {
        System.out.print(text);
    }

    public void newline() {
        System.out.println();
    }
}
```

Fill in that Money class to include a print() behaviour plus supporting ‘secret knowledge’:

```
class Money {
    private final String amount ;
    private final String currency ;

    public Money( String amount, String currency ) {
        this.amount = amount;
        this.currency = currency;
    }

    public void print( Printer p ) {
        p.print(currency);
        p.print(amount);
    }
}
```

A real Money class in Java will store the amount as BigDecimal. This is a fixed precision, accurate way to represent decimal fractions - pounds and pence. The float and double types are never used as they trade size for precision. The higher the amount in a float, the less accurate the value stored is. Very bad for banks.

Our first stab at ItemFormat is from our simple refactor:

```
class ItemFormat {
    private final Money price ;
    private final String description ;

    public ItemFormat( String description, Money price ) {
        this.description = description ;
        this.price = price ;
    }

    public void print( Printer p ) {
        p.print(description);
        p.print(" ");
        price.print(p);
    }
}
```

Add a demo application:

```
class ReceiptDemo {
    public static void main( String[] commandLineArgs ) {
        new ReceiptDemo().run();
    }

    private void run() {
        Receipt r = new Receipt( new Printer() );

        // Nice cheese and wine evening
        r.add("Brie", new Money("1.95", "GBP"));
        r.add("Tiger Bread", new Money("0.95", "GBP"));
        r.add("Merlot", new Money("7.95", "GBP"));

        r.print();
    }
}
```

And spin it up to see:

```
Brie GBP1.95
Tiger Bread GBP0.95
Merlot GBP7.95
```

Review of design so far

Let's look back on our design.

We started with behaviours on our objects.

The design was driven by behaviours. These were supported inside our objects with hidden knowledge (price, description) and hidden algorithms (items.forEach() and the formatting of ItemFormat.print())

Objects were designed to do one small part of the job.

Objects collaborated with each other to get the whole job done.

The IDE helped us fill out implementation details, once we had decided them.

Design was done 'backwards'. We left all details until last. We prioritised getting the code structure right first.

Changing this software will be simpler. Splitting the job into separate objects will limit the impact of changes.

Object Oriented code looks different. Chains of behaviour get driven out by design. High level abstractions - the 'what', not 'how' - are first class citizens of our code. The details can be changed at will. Want to replace that items.forEach() line with a for loop? Go ahead. It will make no difference to the rest of the program.

We have planned a limited degree of resilience to change. ItemFormat will help us change the output format whilst limiting what code needs to change. Printer could have its insides changed to write to a real inkjet printer, say. The rest of the code would not change.

Later chapters will cover an even better way of allowing swapping out one kind of Printer for another, without changing the inside of Printer at all. That's what Dependency Inversion is all about. We shall meet it as one of the SOLID principles.

But looking back at our code even now, we can see the power of behaviours.

Emphasise **what** the code does, not **how**

Exercise: Total Amount

A new requirement comes in as an agile User Story:

- As a customer
- I want to see the total price
- So that I know what I need to pay

How would we drive that slice through, behaviour first?

Hint: Change the field Money.amount to a BigDecimal. Avoid the temptation to add a getAmount() method to Money. Think of how the Money object could collaborate with some other object, responsible for keeping track of the total and printing it on our Printer object.

Test Driven Development

We've seen how OOP is all about making calling code simple and hiding details inside methods. We build up a set of objects that collaborate with each other, using only those public methods. We've learned to design the behaviour methods first, add any needed private data, then add a constructor last. We design from the outside of an object in.

What if we could find a way to help us remember to do things in that order - and *run* the software, even before it's all finished?

Test Driven Development (TDD) is a way of doing just that.

TDD helps us get the design right. It forces us to think about how our object methods look to callers before writing any internal code.

TDD proves that our logic works as expected. We call a method, capture the output and compare it to an expected value.

In short, it's pretty much magic. And you should definitely use it.

TDD also provides a way to prove out that our object works correctly with its collaborators even before we write them. But we'll need to cover 'polymorphism' in a later chapter before we attempt that.

Let's start with the basics. How do we get started with TDD? How does it help?

Outside-in design with TDD

TDD helps us design the public interface of our objects to be easy to use.

The key is to write code that calls our object using what we think will be a good public interface. We do this *before* writing code inside our object. This forces us to think about making the calling code simple. Our 'backwards thinking' of OOP again.

This piece of code is known as the test code. It will set up our object, call a behavioural method on it, and then check whether that method worked or not.

This test does two things for us:

- verifies the behaviour is correct
- verifies the call is easy to make - a hallmark of good design

If our design is hard to set up or hard to use, we'll see that in hard to read test code.

Java projects use the wonderful JUnit test framework to help out with this. I like to add AssertJ to help with the checking part.

Let's use TDD to build a little calculator that can add up our restaurant bill. As you follow along, notice the *rhythm* of TDD - write a bit of test, see a failure, fix it with a bit of code, repeat.

Our calculator class should do two things:

- give us a running total of all the items we've added
- add another item.

We start by writing an empty test. We want this test to prove something that will be true of our BillCalculator class as soon as we create it. If we create this object and don't add any items to it, the total must be zero.

Let's start to write a test for that.

First test: total starts at zero

```
1  class BillCalculatorTest {
2      @Test
3      public void totalStartsAtZero() {
4          // ... todo
5      }
6  }
```

We've got a test class and a test harness method all named for what they do.

Now we do a tiny step of design for our object. We want to be able to access the latest running total. For this, despite all my bleating in previous chapters, a getTotal() method seems like the simplest, most clear design.

Optimise for Clarity again, even if it means ignoring what I say ;)

Let's add this design decision to the test:

```
1  class BillCalculatorTest {
2      @Test
3      public void totalStartsAtZero() {
4          // Act
5          float total = calculator.getTotal();
6      }
7  }
```

The compiler will be loudly complaining at this point, as indeed might you if this is your first TDD session: "There isn't any object yet!"

Quite right. Let's fix that.

Using your IDE shortcuts, create a new Class called BillCalculator:

```
1  class BillCalculator {
2  }
```

Now, we can fix the first of our compiler's many complaints and create a calculator object:

```
class BillCalculatorTest {
    @Test
    public void totalStartsAtZero() {
        // Arrange
        var calculator = new BillCalculator();

        // Act
        float total = calculator.getTotal();
    }
}
```

I'm using the 'var' keyword from recent Java editions (don't ask me which one; I never was a language lawyer kind of guy). I think this reads very clearly here.

The compiler will grudgingly accept we are now not complete idiots and remove the red X from under 'calculator'. It will then move on to its next complaint. We need to add the getTotal() method.

Let the IDE shortcuts do this, because the IDE has all the information it needs to get it right:

```
class BillCalculator {
    public float getTotal() {
        return 0.0;
    }
}
```

I love working backwards like this. The IDE has enough information to do most of the grunt work *and* not make as many typing mistakes as I do. It's also a very fast way to work once you get into the rhythm of it. You even look (to the clueless) like a *10x mega ninja rock star programmer*.

Anyway. The compiler will be happy, allowing us to add our check that everything is ok. We'll use the wonderful 'AssertJ'[1] library to provide the 'assertThat' method.

[1] https://assertj.github.io/doc/

```
class BillCalculatorTest {
    @Test
    public void totalStartsAtZero() {
        // Arrange
        var calculator = new BillCalculator();

        // Act
        float total = calculator.getTotal();

        // Assert
        assertThat(total).isZero();
    }
}
```

We can run this test - and it will pass! Our calculator object has passed its first and only test. It just so happens that the only thing we are testing is that the total is zero and the IDE generated code always returns zero.

Arrange, Act, Assert - a rhythm inside each test

Notice how writing our code by designing behaviours first resulted in the test having three sections.

We first decided on the behavioural method and added a call to it. This is known as the 'act' step - where we take action.

We also decided what that method should return. This means we know how to test it by comparing it to an expected value. This is the 'assert' step - we make an assertion about the output and confirm it is true.

And finally, we're going to need an object to test! This is the 'arrange' step, where we make all necessary arrangements for the test to begin. Here, we create objects, wire up and collaborators and load initial values.

Every test we write has this rhythm: Arrange, Act, Assert[2].

[2]see my video here https://www.viewfromthecodeface.com/how-to-write-a-tdd-unit-test-with-java/

Red, Green, Refactor - a rhythm in between tests

After writing our first test, we can get on to the real meat of development. We iteratively add features to our code one test at a time.

Before we do, TDD has one extra rhythm to guide us: Red, Green, Refactor.

Arrange, **Act**, **Assert** guides us *inside* a test, helping us write clean test code and remember all the required pieces.

Red, **Green**, **Refactor** helps us discover *emergent design*. It happens in between tests.

The rhythm goes like this.

A 'red' test means we have written a test which fails. This gets us running code quickly. It proves that the test is working; it correctly reports that the code we have not yet written is not yet working.

We follow the Arrange, Act, Assert process to get to a 'green' test. At this point, we have added just enough production code to make the test pass. In one sense, the code is 'finished'.

But it isn't.

If we continue working this way, I guarantee you will end up with a giant mass of unreadable spaghetti. Quite often, the process of TDD takes the blame for this, when it hasn't been done right. A bit like how people whack getters and setters everywhere, then blame OOP.

It's because we've missed a step. That missing step is 'Refactor'.

Refactoring is where we look at our code and improve its design - *without* changing how it works. We change the structure, not the behaviour.

The best book on this by far is "Refactoring: Improving the design of existing code" by Martin Fowler [^refBook]. The first edition has Java examples and is the one to read. I really recommend that you do.

In the TDD refactor step, we apply principles from this book to our code. We look at our production code *and* our test code. We do a mini Code Review and look for improvements:

- Do all names reveal intent?
- Can we eliminate duplicated code ('DRY' Don't Repeat Yourself)?
- Can we replace complex syntax with simpler syntax?
- Can we split something out to add an explaining variable or method?

If we find any improvements - or perhaps our pair or mob colleagues do - we make them now.

At the end of each Red - Green - Refactor cycle, we have code that has good OOP design, has been proven to work and has the right quality.

In our case, we've only just started. It's hard to mess up code on a blank slate, so there's nothing we need to do.

Second test: Adding an item gives us the right total

Time to add our first 'proper' feature - which in TDD means writing another test.

We retain all the tests we have already written. By doing that, we build up a whole suite of regression tests. A "regression" is where we accidentally break something in Feature #1 when we add Feature #2. By keeping all our tests passing, we prove that we have not done that.

Frequently during development - and certainly before a commit - we run all tests and make sure they all pass.

Of course, this is a massive boost to software quality. It might also seem rather obvious. But things are only obvious when you know how. I mention it because of one Bootcamp session, where one student ended up with only one test, deleting all the others in each new iteration!

Anyway. The new test:

```
class BillCalculatorTest {
    @Test
    public void correctTotalForOneItem() {
        // Arrange
        var calculator = new BillCalculator();

        // Act
        calculator.add(12.95);
        float total = calculator.getTotal();

        // Assert
        assertThat(total).isEqualTo(12.95);
    }
}
```

Designing the second feature

We proceed with the same Arrange, Act, Assert rhythm as before.

We're making more design decisions now. We are beginning to affect the shape of the code. I've made two decisions, at least one of which is questionable: naming the behaviour that adds to the total 'add', and using a float to represent the price. To be fair, that was done in the previous step.

Using float is the questionable one. Floats and money do not mix well outside of student exercises and trivial exercises in books (ahem). They have two problems for money: rounding errors and currency type.

Floats lack precision. They usually use the IEEE-754 standard to store a mantissa and exponent with a certain number of binary bits. The upshot of which is that certain fractions - like one third - cannot be represented accurately. This leads to rounding errors when adding numbers. Not great for adding up bills.

A raw float also does not have knowledge of the currency involved. For mixed currencies, often found in banking, this makes them useless.

Creating a Money class can fix both of these problems. But this is an example on TDD after all, not banking. Let's just move on and pretend neither of us said any of

that. Try to not look guilty.

Let's use that test to drive out a reasonable implementation:

```
1  class BillCalculator {
2      private float total ;
3
4      public void add(float itemPrice) {
5          total = itemPrice;  // 1. We need to talk about this...
6      }
7
8      public float getTotal() {
9          return total;
10     }
11 }
```

The test passes. Boom! High fives all round.

Is there anything to refactor? Nothing really jumps out, but I've decided I want to Refactor > Rename the class name to 'Bill'.

That seems to make more sense than 'BillCalculator' now we have the behavioural method 'add'. This isn't a calculator. It is a restaurant bill, and we are doing things with it. That's a different way of looking at this problem. It's common to get this kind of insight as your object oriented design unfolds and you learn more about the problem. This is why agile, iterative development works really well with OOP.

We'll use the IDE tools to rename BillCalculator to Bill and check the box to rename the test in lock-step. I'll Refactor > Rename the local variable 'calculator' in the test to 'bill' as well.

Refactor > Rename to Optimise for Clarity

TDD Steps - Too much? Too little?

Let's look at the design decision at (1). It reveals a tension in TDD.

I did strict TDD above, to show how it's done. I made the test pass by writing the simplest code that came to mind to do it. The normal trick is to then tidy up anything we don't like in the refactor step.

But this is something different. It's about how much progress to do in one step.

The implementation at (1) *assigns* the price to the total. It overwrites the existing total, rather than adding it up.

This means that the zero case will pass, and a single item case will pass. Adding a second item will fail.

Strict TDD *always* does the bare minimum at each step. This is called the 'YAGNI' principle.

YAGNI - You Ain't Gonna Need It

YAGNI stands for 'you ain't gonna need it'.

It was a response to the 1990s heyday of throwing in every conceivable Design Pattern, whether it was needed or not. Making things 'flexible, configurable, extensible' was the order of the day. Yet all you really needed was a simple app that did the job.

Once designs became littered with 'AbstractCalculationEngineBase' and 'BillSplitStrategyEqualPartitioningSingleCurrency' and 'ItemFactoryFactory', we realised things had gone too far.

And so YAGNI was born. The solution to all complexity is to ask if it is needed and just not bother with it if it isn't. If you aren't going to need it yet, don't write it.

YAYA - Yes, You Are

…which is great, until you look ahead to the next feature. Often, you realise that yes, you ARE gonna need it.

You might not find the acronym YAYA in any other books. Largely because I just made it up.

But just like we have the Yin and the Yang, we have the Yagni and the Yaya. Two opposing forces in development. Do we keep things bare minimum all the way, just to avoid over-complicating things? Or, given we know what's coming next, is *this* over-complicating things?

You have to decide on a case by case basis.

But one thing is for sure. Our next test is going to flush out that assignment to total and replace it with an addition:

```
1  class BillTest {
2      @Test
3      public void correctTotalForTwoItems() {
4          // Arrange
5          var bill = new Bill();
6
7          // Act
8          bill.add(12.95);
9          bill.add(2.05)
10         float total = bill.getTotal();
11
12         // Assert
13         assertThat(total).isEqualTo(12.95 + 2.05)
14     }
15 }
```

and we make the single-character change to our production code to make it pass:

```
class Bill {
    private float total ;

    public void add(float itemPrice) {
        total += itemPrice;
    }

    public float getTotal() {
        return total;
    }
}
```

You can see where the assignment has been changed to addition.

I'll leave it to you to decide if we could have done that in one step instead of two. Or if it even really matters.

Optimise for Clarity with well-named tests

Each test needs a name. We already know that names are important.

How do we 'Optimise for Clarity' for the name of a test?

For me, the answer is the same as for methods. Name each test according to the outcome of calling that method. Once the test has successfully run, what will we have learned about the method we're testing?

We've already seen an example in our Bill calculation test. The first test creates a bill with no items, then confirms that the total amount is zero. A good name is then:

```
class BillTest {
  @Test
  public void totalAmountIsZero() {
     // Arrange, Act, Assert goes here
  }
}
```

My advice is to drop noise words from the start of the phrase. It's a test case, so we don't need to say things like 'shouldCalculateTotalAsZero', 'returnsTotalAsZero' nor 'checkThatTotalIsZero'.

We definitely *do not* need to write clunky test names like

```
1   @Test
2   public void BillTest_calculateTotal_isZero() {
3      // JUST SAY NO!!!!!
4   }
```

Why?

For me, this goes back to the whole idea of methods being little abstractions that make our code less complicated.

We're not describing *how* the test is built. We are not even describing *how* the method we're testing is implemented. We are describing *what* we want the method we are testing to actually do.

We should use concise, outcome-driven test names. A reader should be able to skim read what the test expects the production code to do. This improves readability. It simplifies refactoring the production code internals later.

Being agile means that we can change our mind about the secrets our objects hide from callers. We reflect this same thinking in our test names.

TDD and OOP - A natural fit

TDD is a natural fit for OOP. OOP is all about designing an object from the outside-in. TDD is a process that scaffolds that.

We've seen two rhythms that guide us:

- Arrange, Act, Assert
- Red, Green, Refactor

We've looked at how YAGNI (and its corner-cutting cousin YAYA) help us prevent over-complicating code. We've seen how we can change our mind about the code, taking the trouble to improve it as we go. We've also seen how TDD helps us grow out a 'right-sized' design, one test at a time.

These ideas are at the heart of Extreme Programming (XP). This 'shifts left' (i.e. reveals earlier) any problems with logic, design, and code clarity.

FIRST Tests are usable tests

We like our acronyms in software development, so we've come up with yet another to help us design unit tests: FIRST.

Each unit test should be:

- **Fast** It should run quickly (milliseconds)
- **Independent** each test can run alone. Each test does not need any other tests to have run first, nor any hardware setup
- **Repeatable** Every time the test runs it gives the same result. The opposite is the *flaky* test
- **Self-checking** The test automatically checks the output. If you need a human to do it, it isn't a FIRST test
- **Timely** Each test takes about the same time to write as the code it is testing

The earlier code examples are all FIRST tests.

Keeping tests FIRST is hugely important. It is the key to having useful tests that help you develop code. Break FIRST, and you get the kinds of tests that only a manager could love.

Real-world TDD

Before moving on, it's worth sharing some experience with TDD in real projects. As you would expect, everything is not quite as rosy as I've just painted it.

The Good

I'm a big fan of TDD. It really speeds up development time (yes, really). The development cycle is way quicker than spinning up three services, a Dockerised database, Node plus a React UI and then clicking through five screens to test that your Bill can add up two items.

The regression testing you get on each code change is worth actual money, in terms of decreased spend on bug fixing and customer reputation.

TDD gives me a lot of confidence in my work. It lets me split things up into small chunks that my limited brainpower can cope with. I used to write hundreds of lines at a time before running code. I have literally zero clue how I did that now.

The Bad

Some teams dislike TDD and refuse to use it.

TDD only works with clean designs that are split up to allow the tests access to the code. Legacy code often cannot use TDD. You cannot get the access you need in the giant ball of spaghetti that passes for code.

I've had somebody jealous of my use of TDD, who made life hard for me in a team. They seemed to feel it somehow showed them to be inferior and that this was something I intended. It's best to get everyone on the same page.

TDD doesn't work at all when you *cannot* decide upfront what a method should do. Whilst that sounds obvious, a lot of real-world development is experimental.

You're tinkering around trying to figure out what code might work. You might be wiring up libraries you don't understand for a feature that isn't fully specified.

The solution here is to do a *spike*. Forget tests: Hack. Make notes on the things that worked and the rough shape of the code that did it. Revert the spike then use TDD to follow your notes and engineer it properly.

The Ugly

TDD can create a truly colossal mess in the wrong hands. Think of toddler-with-Sharpie kinds of mess.

The problem is that TDD does not - and cannot - design your code for you. Which is why I put it after OOP ideas in this book. It just can't.

Instead, TDD gives you *design feedback*. Here are the three classic test code smells:

- Messy Arrange Step: Your object is over coupled to other objects. Split it.
- Messy Act Step: Your object usage pattern is too complex: Simplify it.
- Messy Assert Step: The results are hard to use. Rethink it.

What often happens is that these test smells get ignored. Time pressure, lack of leadership, or perhaps lack of knowledge all lead to this.

Whatever. The hapless coder copies and pastes like it were Groupon vouchers for free pizza and digs a deeper hole. The tests then actually *lock-in* the bad design. It gets hard to change - as all these bad tests get in the way!

A very similar thing happens when the tests are too tightly coupled to the object's inner secrets. We learned to avoid this by naming tests well and letting that guide our thinking.

But if we don't, our tests prevent our objects from changing. The tests force our future code to be a kludgey series of workarounds.

That isn't the fault of TDD. It's the fault of *not doing TDD right*.

But be aware of it, because it happens in the wild.

Polymorphism - The Jewel in the OOP Crown

By far, the most useful tool in Object Oriented Programming is polymorphism. It is the key to everything that follows.

It is all about making objects that can be swapped.

Polymorphism gives us a nice alternative to conditional logic. It allows us to split up our program in really interesting ways. We can test code in isolation, decouple code from external systems and reduce if / switch statements.

There are a couple of ways of using this technique in Java code. Let's look into the clearest approach with an example: Interfaces.

Classic example: Shape.draw()

The classic example is Shape.draw(). It is simple and gets the point across well.

This example is about having several classes that each know how to draw a single shape. Circle knows how to draw a circle. Square draws a square. We want a piece of code that can draw every kind of these shapes, yet allows us to easily add more kinds in the future.

The Shape Interface

We start with a pure interface. This interface defines the behaviours all objects that implement it must support. This is the part that gives us polymorphism.

It is a promise to all calling code that any object that implements Shape has a method draw() that we can call.

```
interface Shape {
  void draw();
}
```

What I really like about this is that it expresses *design intent.*

I'm telling readers of my code that all Shape objects know how to draw themselves when asked. That is the only thing the calling code has to know: it can call draw()

To make this useful, we need to create some concrete classes that hide the details of how to draw their specific shape.

To simplify this example, draw() will write a text description of the shape to the console. A more complete example passes in a Display object to draw(), which gives us access to graphics operations. We'll see that later in the book.

For now, let's focus on the basics of exploring polymorphism and making draw() do different things in different objects.

```
class Circle implements Shape {
  public void draw() {
    System.out.println("Look how round I am");
  }
}
```

This is a class that knows how to draw a circle. Just for this example, that means it writes some text to the console.

```
class Square implements Shape {
  public void draw(){
    System.out.println("Four sides all the same for me");
  }
}
```

Square is a class that knows how to draw a square.

Tell Don’t Ask - the key to OOP

This allows you to write code that takes any Shape and asks it to draw itself. This is the **Tell Don’t Ask** principle in action. Think back to our object basics chapter, where we talked about not telling people how to do their jobs. We just ask them to do them. This is how that works out in code.

It’s magic, and we’ll talk about why in a minute. But first, let’s see how the calling code looks:

```
1  var shapes = new ArrayList<Shape>();
2
3  shapes.add( new Circle() );
4  shapes.add( new Square() );
5
6  shapes.forEach( shape -> shape.draw() );
```

This snippet makes a list of objects that implement Shape. It adds instances of Circle and Square to that list. It uses the Java 8 forEach method to go through each object in the list and call the draw() method on it.

shapes.forEach(Shape::draw) is shorter. This is easier to learn from

Because draw() is polymorphic, the Java runtime routes the call to the code in the *implementing class*. Even though the calling code appears to call the same draw() method each time, *it doesn’t*. The runtime routes it down to whatever actual class it finds.

The output looks like

> Look how round I am
> Four sides all the same for me

Let’s talk about that magic I mentioned. You’ll notice a couple of brilliant things about this.

- No `if` or `switch` statements
- No instanceof statements (ugh! Just, no)
- The loop code does not know what kinds of shapes are in the list
- We can add new shapes without affecting any others

Having no if or switch statements is a huge win. An alternative approach is to have a single draw() method that has a giant switch statement in:

```
1  public void draw() {
2      // DO NOT DO THIS!!!
3
4      switch( shapeType ) {
5          case "CIRCLE":
6            System.out.println(“Look how round I am”);
7            break;
8
9          case "SQUARE":
10           System.out.println(“Four sides all the same for me”);
11           break;
12
13         default:
14           throw new IllegalArgumentException("Unknown Shape type " + sh\
15 apeType)
16     }
17 }
```

You can see how code like that grows and grows as new shapes get added. The code itself is complex, as all conditionals are. They have extra possible execution paths through the code. Each one needs testing. Each one increases the chance of coding mistakes.

The polymorphic approach completely separates those pieces of code. The Circle class has no connection at all to the Square class. That would not be true in our ‘giant switch statement’ case. So changes to one do not affect the other.

The other result that follows is that adding a new shape is easy.

```
class Triangle implements Shape {
  void draw() {
    System.out.println("I really like having three points");
  }
}
```

The draw loop code is completely unchanged - but can now draw triangles

To add in our triangle, we only have to make an addition to our object creation code:

```
var shapes = new ArrayList<Shape>();

shapes.add( new Circle() );
shapes.add( new Square() );
shapes.add( new Triangle() ); // The only new line of code

shapes.forEach( shape -> shape.draw() ); // No change here
```

This is an example of the **Open/Closed Principle** (OCP) which we will cover in the chapter on SOLID. It is the "O" of the acronym. It is important because it means we can keep code open for adding new features to - but closed against changing existing code that we know works.

The SOLID Principles

As object oriented programs grow large in size, it gets more essential to split that system apart into little pieces.

But what does ‘split apart’ mean? And what is a ‘little piece’, exactly?

The five design principles known as SOLID came about as solutions to this. They have some pretty indecipherable names but are easy enough - once you understand what problems they solve.

The five SOLID principles

Let’s just list them out, so you know what the acronym means:

- **SRP** Single Responsibility Principle
- **OCP** Open/Closed Principle
- **LSP** Liskov Substitution Principle
- **ISP** Interface Segregation Principle
- **DIP** Dependency Inversion Principle

You can see the acronym in the first letters.

The titles of each principle are jargon in the proper sense. Once you know what each one means, the short titles make sense. But until then, they won’t mean much.

It’s important to note that the order of these is ‘what sounds best’ - not ‘what is most useful’. So, unlike every other book on SOLID, I’m going to explain the principles in order of usefulness - “SDLOI”. I’ll give you that it doesn’t roll off the tongue as easily.

SRP Single Responsibility - do one thing well

Code that does many things is hard to understand.

If we have a class that opens files, reads lines of text, does a spelling check, blanks out passwords, then writes an HTML page - it's a headache.

It's doing more than one thing.

The solution is to split this up into separate classes where *each one does one thing*.

That is the SRP. Give a Class one single responsibility.

It is both the simplest and most widely used of the SOLID principles. Unfortunately, it is also the most difficult to pin down.

What is 'one thing', anyway?

The problem with it is in deciding what 'one thing' is.

I'm writing this on a web-based Word Processor. To me, it is 'just one thing' - the thing I type on.

But inside, it will have many different moving parts: Text management, storage management, web input and output, to name a few.

It's hard to create any rules around this. When we look at 'Hexagonal Architecture', we'll see some things that help us figure out single pieces, by splitting out anything concerned with an external system.

For most of our code, though, a good rule is to look at the language we use in our test names.

Do they talk about 'the same thing'?

As an example, look at our test class for our Bill object from earlier, the BillTest.

It has methods relating to adding up different total values for different items on a bill.

That's doing one thing.

If it started having tests for `createsHtmlWithUnderlinedTotal()` or `savesTotalToDatabase()` then we can see that our Bill class has grown some extra responsibilities.

The solution is simply to refactor those extra responsibilities into their own classes. Go back to basics and redesign your object collaborations.

But before we can do that, we're going to have to look at our next SOLID principle: Dependency Inversion.

DIP Dependency Inversion: Bring out the Big Picture

This is the most important and most useful idea of SOLID.

By Inverting Dependencies, you create code by assembling swappable 'plugin' style components.

This gets you two big benefits:

- Changes in one component do not ripple through to any others
- You can replace components with ones designed to help TDD tests

What is an 'inverted dependency'?

The easiest way to understand this is to take some code without it, see what problems it has, then refactor it.

Let's start with a simple method that lives in a class somewhere. Its responsibility is to take some keyboard input, make it all upper case, then display it on the console:

```
public void showInputInUpperCase() {

    // Fetch Keyboard Input
    Scanner scanner = new Scanner(System.in);
    String inputText = scanner.nextLine();

    // Convert
    String upperCaseText = inputText.toUpperCase();

    // Display
    System.out.println(upperCaseText);
}
```

There's nothing remarkable about it, at first sight. We see the three sections to fetch input, convert, display.

But notice that new keyword. That's a problem.

Because of this new, the method has a *direct dependency* on how the keyboard is read and how the output is displayed.

It also calls System.out.println() for output, which has the same problem. We are tightly bound to a global method. But let's just consider input for now.

Why is 'new' such a problem?

The new keyword itself is not a problem. It is *where it is in the code.*

Dependencies like this are hard-coded. They cannot be changed.

If we wanted a different input method, we would have to go into this method and change the code. We would delete the two lines relating to Scanner, and replace them with something else.

This doesn't sound too bad for a four line method. But as a design principle, it is exactly what we don't want. We don't want it so that unrelated areas of code have to change.

Think about it this way. What is the main purpose of our little function? It is to change something into upper case. Yes, it needs some input from somewhere to do

that. But the function does not need to know anything at all about how that input is obtained.

This is where TDD really forces a good decision early.

How could we write a unit test for our function? Spoiler: We couldn't.

We would have to admit defeat. We don't have a way for the test to make keyboard input at this level. We are locked-in to using a Scanner object.

But let's re-phrase that question and not give up: How *could* we write a unit test? What would we have to change?

Inverting the input Dependency

What we need is some way for this code to fetch input, without knowing anything about how that is done.

We've seen how to do that before. Our polymorphic Shape.draw() knew how to ask an object to do something without knowing anything about how.

Let's use the same technique here.

We'll introduce an interface called Input, which we can ask to give us some text:

```
1  interface Input {
2      String fetch();
3  }
```

This is an abstraction. It says 'you can ask me for an input string, but leave it to me to figure out where it will come from'.

It replaces the two lines of code relating to the scanner in our example code.

```
public void showInputInUpperCase() {
    String inputText = input.fetch();

    // Convert
    String upperCaseText = inputText.toUpperCase();

    // Display
    System.out.println(upperCaseText);
}
```

I've also removed the comment. The interface makes it clear enough without.

As it stands, this would not compile.

We need to make two more changes:

- Create an implementation of this interface
- Add a parameter or field 'input' - so our method can use the interface

Making a concrete KeyboardInput class

We can easily make our first implementation of this interface:

```
class KeyboardInput implements Input {
    @Override
    public String fetch() {
          Scanner scanner = new Scanner(System.in);
          String inputText = scanner.nextLine();

          return inputText;
    }
}
```

We've copied-and-pasted the Scanner code into a new class that implements the interface. It is called 'KeyboardInput' because this class is allowed to know exactly where the input comes from.

This class is what is meant by inverting a dependency.

The old method newed up the Scanner inside the method. It depended on Scanner.

The new method code depends only on the interface. The new KeyboardInput class wraps up Scanner and also depends on the interface. That's the inversion part.

We've changed the dependency from "method depends on Scanner" to "class containing Scanner depends on interface".

The dependency is precisely backwards to before. It is inverted.

This thought process is known as the **DIP** Dependency Inversion Principle.

Dependency Injection - using our inverted dependency

Our original method needs a way to use this inverted dependency.

The most typical way is to inject it into a constructor:

```
class TextConversion {
    private Input input ;

    // This is where we inject the dependency
    public TextConversion(final Input input) {
        this.input = input;
    }

    public void showInputInUpperCase() {
        String inputText = input.fetch();

        // Convert
        String upperCaseText = inputText.toUpperCase();

        // Display
        System.out.println(upperCaseText);
    }
}
```

and we would build a little application to wire it up and run it:

```
public class TextUtility {
    public static void main(String[] commandLineArgs) {
        Input input = new KeyboardInput();

        new TextConversion(input).showInputInUpperCase();
    }
}
```

Running this, we get our original behaviour. This has been a refactoring exercise, not a feature change.

Can you see how we have *externalised* the details of keyboard input from the actual text conversion?

Swappable input sources

Here is the most remarkable thing about this. This design isolates changes to the input source.

If we decided to change from using the keyboard to using a database, we would not change the TextConversion class in any way. Nor would we change the keyboard input class. Which means we wouldn't break anything we'd already written.

We would write a new concrete class called DatabaseInput that implemented the Input interface. We would wire that up in our application instead of the keyboard input:

```
public class TextUtility {
    public static void main(String[] commandLineArgs) {
        Input input = new DatabaseInput();

        new TextConversion(input).showInputInUpperCase();
    }
}
```

You can see nothing has changed except for the class name we create with new.

Inverting the output to display

In the same way as abstracting out the input source, we can do the same thing for display output:

Create the interface:

```
interface Output {
  void display( String toDisplay );
}
```

Create a concrete class to display to System out:

```
class ConsoleOutput implements Output {
  @Override
  public void display( String message ) {
    System.out.println( message );
  }
}
```

Inject both Input and Output dependencies in the constructor:

```
class TextConversion {
    private Input input ;
    private Output output ;

    public TextConversion(final Input input, final Output output){
        this.input = input;
        this.output = output;
    }

    public void showInputInUpperCase() {
        String inputText = input.fetch();
        String upperCaseText = inputText.toUpperCase();
        output.display(upperCaseText);
    }
}
```

And create the concrete classes in the application:

```
public class TextUtility {
    public static void main(String[] commandLineArgs) {
        Input input = new KeyboardInput();
        Output output = new ConsoleOutput();

        new TextConversion(input, output).showInputInUpperCase();
    }
}
```

That allows us to swap to any input or display source. We only change the objects that get created with new. Nothing else.

And *that* helps us with our unit testing. We can swap the input for something we can control and swap the output for something that captures anything sent to it.

We'll see how this works in the chapter on 'TDD and Test Doubles'

Inversion - Injection: two sides of the same coin

It's quite common to have heard about DIP, SOLID and DI frameworks like Spring, without ever really connecting the dots.

Dependency Inversion is the *design* technique. The thought process by which we remove a hard-coded coupling from the software.

Dependency Injection is the *implementation* technique. We have inverted a dependency. Now we need some way of choosing a suitable concrete implementation and plugging it in.

DI frameworks are often useful. But don't forget - simply new-ing up classes and passing them into a constructor is all that DI is at heart.

LSP Liskov Substitution Principle - Making things swappable

Barbara Liskov is a computing legend. LSP is the name given to an important principle she discovered.

The principle helps us reason about a problem we have in polymorphism.

Let's go back to our Shapes example to see what it is.

When Shapes go Bad

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw(){
        System.out.println("I'm round");
    }
}
```

This is the 'good' case.

A Circle implements the draw() method of interface Shape and can be used anywhere a Shape is expected.
You'll recall the example calling code which could mix together Circle, Square and Triangle without code changes.

But can every class that implements the interface method be swapped in? How about this one:

```
class CardDealer implements Shape {
    private final Deck cards ;

    CardDealer( final Deck cards) {
        this.cards = cards;
    }

    public void draw(){
        Card next = cards.next();
        System.out.println(next.asText());
    }
}
```

This might be a class for a card game; Bridge or even Snap.

Class CardDealer implements the method Shape.draw() and so it is polymorphic with it.

We can add it into our list of shapes easily enough:

```
1   List<Shape> shapes = List.of( new Circle(), new CardDealer());
2   shapes.forEach( Shape::draw );
```

That will compile and run.

But will it work *as expected*?

Substitutability

Intuitively - No. Of course it won't work as expected. CardDealer does not fit in with our expectation of what Shape.draw() should do.

The whole setup of Shape.draw() and Circle and so on suggests that we expect our classes to draw geometric shapes. Not draw cards from a deck.

This was the insight Barbara Liskov had. She went on to formalise this intuition mathematically. It is that formalism which is known as the Liskov Substitution Principle (LSP).

LSP reminds us that classes implementing interfaces are only useful if they can be substituted for that interface in all circumstances.

Specifically, a class meeting this *must*:

- Accept all possible inputs
- Produce all expected outputs for those inputs

In short, you can swap out any of the implementing classes without causing the code to break.

Liskov gave a rigorous definition as:

> if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program

Practically, I always end up asking that simple question: "Are there any cases where this class would break the interface contract?"

LSP also applies to inheritance in general - classes and subclasses, abstract classes and so on. It is not restricted to interfaces and implementations.

The only reason I describe it in those terms is that I find interface/implementation to be the simplest, safest usage of inheritance in OOP. It is 'the happy path'.

OCP Open/Closed Principle - adding without change

The Open/Closed Principle describes code that can be made to behave differently without changing that code itself.

We've seen this before in our Shapes example. The code that draws Shapes can draw any shape without needing any change itself.

```
1  class Shapes {
2      private final List<Shape> shapes = new ArrayList<>();
3
4      public void add (Shape s) {
5          shapes.add(s);
6      }
7
8      public void draw() {
9          shapes.forEach(Shape::draw);
10     }
11 }
```

This aggregate class allows you to add shapes to an inner list then draw them.

Where OCP comes in is that you can add any LSP compatible Shape implementation using the add(shape) method. You can then change the behaviour of Shapes.draw() without changing anything inside that class. The change of behaviour is done outside of the Shapes class, not inside.

This is what is meant by Shapes is Open for extension (you make it do different things), but closed for modification.

OCP is essentially Dependency Inversion in disguise.

Strategy Pattern: Externalising behaviour

There is a general-purpose pattern for OCP.

First described in the book Design Patterns by the 'Gang of Four', the Strategy pattern is a general way of allowing behaviour to be injected into a class.

Let's think of our Shapes class as being part of a bigger graphics program, like Photoshop. We want to add a way to process each Shape to our Shapes class. Our first algorithm could be to add a blur effect to each shape.

One way to do this might be to add a blur() method to Shapes:

```
interface Shape {
    void draw();
    void apply(Filter f);
}

class Shapes {
    private final List<Shape> shapes = new ArrayList<>();

    public void add (Shape s) {
        shapes.add(s);
    }

    public void draw() {
        shapes.forEach(Shape::draw);
    }
```

```
16
17      public void blur() {
18          shapes.forEach(s -> s.apply(new BlurFilter()));
19      }
20  }
```

To support this, we have also added an apply(Filter f) method to Shape to give us a way to run the processing on each Shape. Filter is an interface. Each different kind of graphics processing must implement it. We pass in a concrete class of our choice, depending on what kind of graphics processing we want. In this example, it is a BlurFilter, which gives each shape a gentle Gaussian blur effect.

This works just fine. But we have had to modify class Shapes.

In general, we want to avoid modifying classes that we've already written. Or at least only do it for a good reason. We don't want to introduce bugs into code that already works. We certainly don't want to create giant classes with loads of methods. That's the motivation behind the Open/Closed Principle.

Adding a feature like graphics processing to Shapes is usually going to need a code change like this. Any other way of tackling it would be a stretch.

If we only need the one method blur(), this isn't a bad way to do it.

But if we needed more kinds of processing - sharpen, lighten, darken, posterise say - could we do better? Could we limit it to making only the one change and then injecting in those other behaviours?

This is what the Strategy pattern does well:

```
1   class Shapes {
2       private final List<Shape> shapes = new ArrayList<>();
3
4       public void add (Shape s) {
5           shapes.add(s);
6       }
7
8       public void draw() {
9           shapes.forEach(Shape::draw);
```

```
10      }
11
12      public void filter(Filter filter) {
13          shapes.forEach( s -> filter.applyTo(s) );
14      }
15  }
```

Here, the filter() method injects a class implementing the Filter interface into the method. It then runs it across each of the shapes.

Filter is a *Strategy Pattern*. It defines the strategy for how we filter a Shape. It externalises behaviour.

Our Shapes.filter() method has no knowledge of how exactly we process each Shape. It takes what it is given from outside the class and applies it to each Shape.

In the same way that Shapes.draw() does not know how to draw any shape, Shapes.filter() does not know how to filter any Shape. The Strategy pattern formed by our Filter interface pushes that variable behaviour outside the class, leaving it open for extension but closed for further modification.

Any future new processing will be done by creating a new class that implements Filter. This will be done outside of the Shapes class and will require no changes to it.

The Strategy pattern is a powerful way of parameterising behaviour. We use the power of polymorphism to leave our choice of behaviour open-ended.

ISP Interface Segregation Principle - honest interfaces

The Interface Segregation Principle states that calling code should not be forced to depend on methods it does not use. It's a good way of putting it.

I think of ISP as keeping interfaces as small and honest. It's about avoiding polluting an interface with methods that only sometimes make sense. The trick is to only add methods that *always* make sense. We move the 'sometimes needed' methods elsewhere to where they are always needed, then redesign to make use of them.

Bad Example: TV Controls

The hardware team proudly tell us that their latest low-cost digital TV product is ready for us to get some code into. Our first task is to implement the remote control code.

We keep it simple and write an interface exposing all there is to be controlled:

```
interface Control {
    void on();
    void off();
    void channelUp();
    void channelDown();
}
```

We can stub that out for testing our remote control adapter and create a production class that drives the TV hardware.

All good.

A couple of weeks later, the team tell us they have got some improved Smart TV hardware. We will be launching two products, one low-cost, entry-level version and our high-end Smart TV.

It features digital programme recording as well as YouTube and Netflix internet TV.

We might decide to add the extra features into our existing Control interface:

```
interface Control {
    // entry level TV features
    void on();
    void off();
    void channelUp();
    void channelDown();

    // Smart TV Only features
    void startRecording();
    void endRecording();
```

```
11      void launchYoutube();
12      void launchNetflix();
13  }
```

We start coding our class to drive the TV hardware, and we want to keep the same codebase, rather than fork it per-product.

Ah.

If we keep a single class for both entry-level and Smart TVs, we soon find a problem with the Smart TV methods on the entry-level product: they don't work.

Initial ideas are to add `if (isSmartTV)` statements everywhere needed. That itself is an OOP code smell. We should probably replace that with polymorphism, using two separate classes that implement that Control interface. Call them BasicControl and SmartTvControl.

This gets rid of the if statements. More specifically, it pushes them out to the edge of the system where we create the appropriate one of those two classes. But it still leaves us with a problem. What should BasicControl do with all the Smart-TV only methods?

Typically, these will either be no-operation (no code in there) or throw an UnsupportedOperationException.

The root cause of the problem is that we have violated ISP. We have forced the BasicControl implementer of the interface into using a bunch of methods that make absolutely no sense to it.

Fixing our ISP violation

This is where I wish I could say it was easy to fix. But stuff like this gets pretty involved or pretty hacky.

The first suggestion is to simply split the interface into two:

```
interface BasicFeatures {
    void on();
    void off();
    void channelUp();
    void channelDown();
}

interface AdvancedFeatures {
    void startRecording();
    void endRecording();
    void launchYoutube();
    void launchNetflix();
}
```

One approach would be to make AdvancedFeatures extend BasicFeatures. Above, I've chosen not to, because I don't think it helps us much.

By doing this split, we have fixed ISP in a technical sense. We can create a class (possibly two classes) that will drive the BasicFeatures on both entry-level and Smart TVs. We can create a separate class to drive the AdvancedFeatures on the Smart TV only. Each class only gets methods it cares about.

Redesigning to Command objects

In a practical system, with one codebase being used for two products with different feature sets, fixing ISP isn't enough.

The issue is that we want to expose one set of features to an entry-level TV user and a different superset of features to a Smart TV user. Java interfaces do not express that model.

A Java (and C# and C++) interface is statically bound. We can't merely add methods to it based on our TV type. It is static binding that leads us to write smaller interfaces and use ISP in the first place.

Our application ideally needs some way of modelling that an entry-level TV has three methods - on, off, selectChannel - and our Smart TV has more.

The way to do this elegantly is with a design pattern known as 'Command'.

```
interface Command {
    void execute();
}
```

You can see that a Command interface is a level of abstraction higher than we were using. It knows nothing at all about our specific application. It only knows that any objects that implement Command can be told to 'execute()' - to run their code that makes that command happen.

So we would see classes like

```
class ChannelUp implements Command {
    void execute() {
        // code to drive the hardware to show whatever
        // is on the next TV channel 'up' in the list

        // This is the same code as we would have in our
        // class that implemented BasicFeatures - it is
        // just a refactor into a different code structure
    }
}
```

We would have one class for each of our original methods in our 'big interface'.

Then, we would map each command object against something we got from the TV Remote control. Suppose our hardware team gave us a remote control driver that delivered us a number between 0 and 15 inclusive to represent each command. We would write a class that knew this mapping and could route the command number accordingly:

```
class Commands {
    private final Map<Integer, Command> commandsByNumber =
        new HashMap<Integer, Command>();

    public initialise( boolean isSmartTv ) {
        register(0, new On());
        register(1, new Off());
        register(2, new ChannelUp());
        register(3, new ChannelDown());

        if (isSmartTv) {
            register(4, new StartRecording());
            // ... register the others
        }
    }

    public void execute(int commandNumber) {
        Command c = commandsByNumber.get( commandNumber );

        if ( c != null ) {
            c.execute();
        }
    }

    private void register( int commandNumber, final Command c ){
        commandsByNumber.put( commandNumber, c );
    }
}
```

Using this, we get our 'dynamic interface' effect. The Smart TV has more commands available to it. We don't have any ISP violations, because we have designed that out. ISP is not possible here.

We initialise this class knowing what kind of TV product we have. Then we can pass our commandNumber to execute() and have this class do the right thing, Tell-Don't-Ask style.

But it seems a bit complicated …

Pragmatics: I would choose to do it wrong

Given all that, I would probably choose the solution that violates ISP.

We've only got two products. Most probably, any unsupported commands should have no effect. Given these factors, I would probably go with one big interface and no-operation methods for the Smart TV features.

I love the idea of dynamically mapped Commands at the granularity of one Command per user-visible feature. In this case, it just strikes me as over-engineering.

We're Agile, after all. So if we get yet another TV product out, then I might be tempted to refactor it into Commands then.

Just not now: YAGNI and KISS. You ain't gonna need it. Keep it simple.

Which way would you jump with this?

TDD and Test Doubles

Now that we've covered SOLID, we can use it to improve our OO designs and our testing.

Let's go back to our TextConversion example - our spectacular 1.99 App Store smash hit that converts anything you type into upper case. Think of it as an automated SHOUTING app for the Cancel Culture.

Here was the code as we left it:

```
1  class TextConversion {
2      private Input input ;
3      private Output output ;
4
5      public TextConversion(final Input input, final Output output){
6          this.input = input;
7          this.output = output;
8      }
9
10     public void showInputInUpperCase() {
11         String inputText = input.fetch();
12         String upperCaseText = inputText.toUpperCase();
13         output.display(upperCaseText);
14     }
15 }
```

We had inverted dependencies on external systems at this point.

This code uses interfaces Input and Output to fetch input and display output, respectively. It does not have any knowledge of the mechanism of how that input is collected. It knows nothing about where the output is sent to.

This is the secret sauce for our unit test: we can make test versions of those interfaces. Think of them as 'stunt doubles' for actors in a Hollywood film.

We even call these classes ‘Test Doubles’.

Test Doubles - Stubs and Mocks

The big idea is to fix the problems you have with a manual test.

We can manually test this code by typing ‘abc123’ at the keyboard and visually observing the output of ‘ABC123’ on the console.

This has the usual negatives of manual testing. It is slow. We often forget to do it. It is expensive - and it delays delivery. The solution is to automate the test somehow.

But how do we automate keyboard input? How do we replace the human observer looking at the console?

The answer that SOLID gives us is that *we don’t*.

External systems always mess up unit testing. We cannot assert what they are showing, usually, nor can we alter the input at a physical keyboard.

Instead, we replace the keyboard and console completely by Test Doubles - classes that will automate the input and analyse the output.

DIP for Unit Tests - Stubs and Mocks

Dependency Inversion provides the answer to our testing problem.

Because our code does not know anything about how input is collected or where output is delivered, we can write test-specific classes to help out. These are our test doubles.

We then inject these test doubles into our TextConversion code inside our unit test.

The huge advantage here is that we are not changing the code under test in any way. If it passes Unit Test, that code will be good to go in production. It will be the exact same code.

We need a double for our Input interface and a double for our Output interface. Each one is subtly different in what it needs to do. We have some jargon terms so we can discuss these differences clearly.

- The Input test double is a **stub**
- The Output test double is a **mock**

A stub is simply an implementation of one of our dependency interfaces that *returns known data*. Because we know in advance what data will come out of it, we can predict what effect it will have on the code under test.

One way to do this is to create a StubInput class. This can implement the Input interface and always return the string 'abcde123' from fetch().

Another approach is to use a Mock object. A Mock is complementary to a stub. The mock implements the interface, and *records any interactions* with it.

A MockOutput class can implement Output. It can then capture whatever we send to the display() method. We can write the test to assert against this captured data. This will prove that our code called the output interface using the correct method and with expected data.

Let's see this in code:

```
1  class StubInput implements Input {
2      private final String stubValue ;
3
4      public Input(String stubValue) {
5          this.stubValue = stubValue ;
6      }
7      public String fetch() {
8          return stubValue;
9      }
10 }
```

This will provide our known data whenever we call fetch().

I made a design decision to pass in the stub value I want to return, rather than go with the easy option of returning a constant. The reason is about making the test more readable; we'll see that later.

Now for our MockOutput. Its responsibility is to record any interaction with the display() method:

```
class MockOutput implements Output {
    private String actual ;

    public void display( String toDisplay ) {
        this.actual = toDisplay ;
    }

    public String getActual() {
        return actual ;
    }
}
```

A simple getter seems to do the job of exposing the captured data well here. You could make the field public; you could provide a method 'boolean hasExpectedValue(String expectedValue)' if you like. Despite my general dislike of getters in OOP, it seems the right tool for the job here.

We can now use these test doubles to write a fully automated unit test:

```
class TextConversionTest {

    @Test
    public void displaysUpperCasedInput() {
        // Arrange
        var in = new StubInput("abcde123");
        var out = new MockOutput();

        var tc = new TextConversion( in, out );

        // Act
        tc.showInputInUpperCase();

        // Assert
        assertThat(out.getActual()).isEqualTo("ABCDE123");
    }
}
```

This test uses our stub and mock objects and follows the usual Arrange, Act, Assert pattern. It uses the well-known JUnit test framework. The @Test annotation marks this out as a test that JUnit can run.

Follow the code through, and you can see that we:

- set up our stub input value
- create a mock to capture output
- dependency inject the stub and the mock to code under test
- execute the code we want to test
- assert against the captured output, given the known input

You can see now why I chose to pass the stub data into the StubInput class. It's all about locality of reference. The test method above clearly shows an input of 'abcde123' and a few lines later clearly shows the expected output of 'ABCDE123'.

This is optimised for clarity. It is easier to skim read that test than it would be if you had to click through into StubInput to see what the stubbed value was.

Mocking libraries

Now that we understand how Dependency Inversion allows us to write test doubles, we can simplify our lives.

Writing a lot of test doubles by hand is pretty tiresome, as you can see. It is classic boilerplate code.

As you would expect, libraries can help us.

The most popular in Java is Mockito. It is a library that allows us to generate mocks and stubs using only annotations.

Our unit test above would become this, using Mockito:

```
class TextConversionTest {
    @Mock
    private Input input ;

    @Mock
    private Output output ;

    @Before
    public void beforeEachTest() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    public void displayUpperCasedInput() {
        // Arrange
        Mockito.when(input.fetch()).thenReturn("abcde123");
        var tc = new TextConversion(input, output);

        // Act
        tc.showInputInUpperCase();

        // Assert
        Mockito.verify(output).display("ABCDE123");
    }
}
```

Mockito provides the @Mock annotation to auto-generate test doubles. These function as both Mocks and Stubs.

Stub behaviour is provided by Mockito.when(). The most common use is to chain that with `.thenReturn(stubValue)`. You can arrange to throw exceptions, or return a sequence of results in this way.

Mock behaviour is supported by Mockito.verify() which allows checking that a method was called and with what parameters.

Mockito is a widely-used third-party library. It provides a lot of tools for generating mocks and stubs. I recommend you take a look at it. Also check out AssertJ which

helps write more readable assertions.

Self-Shunt mocks and stubs

One last technique is worth mentioning: self-shunt.

This is where we make the test class implement the interfaces itself. It means there is no need for hand-rolled test double classes and no need for mocking libraries. Tests run faster. We even get a little design feedback when our interfaces get too big; the test gets annoying to write.

```
1  class TextConversionTest implements Input, Output {
2      private final Input input = this ;
3      private final Output output = this;
4
5      private String actualOutput ;
6
7      // Input interface stub
8      public String fetch() {
9          return "abcde123";
10     }
11
12     // Output interface mock
13     public void display( String output ) {
14         this.actualOutput = output;
15     }
16
17     @Test
18     public void displayUpperCasedInput() {
19         // Arrange
20         var tc = new TextConversion(input, output);
21
22         // Act
23         tc.showInputInUpperCase();
24
```

```
25          // Assert
26          assertThat(actualOutput).isEqualTo( "ABCDE123" );
27      }
28  }
```

The test class acts as its own test doubles. The fields input and output make the intent easier to understand.

I worked on one project where this style was the only one accepted. I liked it; but overall, it's a bit weird. I've not seen it used elsewhere.

But it's a nice tool to have in the toolbox.

Refactoring

Every technique so far has been about structuring code for the first time.

We have worked hard to make our code readable and simple. To help us do that, we have designed and tested from the outside in, choosing to make what we are solving more prominent than how we are solving it.

But what about when we haven't got it right? What happens when we learn a new insight into our problem? What about when we have thrown down 'the simplest possible code' to make a test pass, but now we want clean code?

What is refactoring?

Refactoring is the name given to changing the structure of our code whilst keeping its behaviour the same. Re-structuring our code, but without breaking anything.

The name comes from factoring, which comes from algebra. Factoring is the basic idea of splitting something up into smaller parts. Refactoring is simply 'doing it again'.

I am always refactoring.

I'll refactor as part of the TDD process, as is standard. I will write a failing test, add the simplest production code to make it pass, then refactor.

When faced with new code that's hard to understand, I'll refactor it first. That way, I capture any insights I get and preserve this hard-won understanding in the code.

That's why refactoring is useful. But how do we do it?

Martin Fowler's book Refactoring is the reference on this subject. The first edition is all Java-based and the one I recommend for Java devs. The second edition uses JavaScript ES6 with class support for the examples. It's okay, but not as direct for us Java developers.

Here are the refactoring steps I use all the time - and why.

Rename Method, Rename Variable

```
1  class User {
2      private final String text ;
3
4      public User( String name ) {
5          this.text = name ;
6      }
7
8      public void printout() {
9          System.out.println("Welcome back, " + text);
10     }
11 }
```

This User class will print out 'Welcome back, Alan' after creating the object passing in the name Alan.

The name is stored in a field called 'text'. Hmm. It is text - but that's not really telling me what that field is used for. It is being used to store the User's name that we want to put into the welcome message.

The fix is to use Refactor > Rename of that field. Call it 'name'.

In the same vein, method 'printout()' isn't very helpful in understanding the code, either. Yes, the method will print something out. We can read that. But that isn't *what* it is doing, that's *how*. We want our methods to explain why they are there and what their outcome is.

We can use Refactor > Rename on the method name to give it a descriptive name. Let's choose 'welcome()'. This better explains what is being done.

```
class User {
    private final String name ;

    public User( String name ) {
        this.name = name ;
    }

    public void welcome() {
        System.out.println("Welcome back, " + name);
    }
}
```

These two refactorings - changing the name of a field or method - are the ones I use all the time. Literally. To the point where I no longer stress out about thinking up 'the best' names. I just code, come up with a first stab at a name, then come back and refactor it when I get a better idea.

I find my brain works like that. Something happens in the background as I am working away. I'll get a better insight a little later.

Prefer IDE tools over manual changes

Your IDE if it is IntelliJ, Eclipse, NetBeans or VS Code will automate this process. As Java is a strongly typed language, the IDE can find all references to that field text and change them all in one go. Same for the method and all its call sites.

Use the IDE Refactoring tool always

Use the tools. It is much faster. You will introduce fewer bugs.

Extract Method

Another favourite that I actually use for several purposes:

- Split up a long method into logical chunks
- Replace comments by turning them into method names
- Clarify how a method works, by adding more descriptive names

Here's a SpaceInvaders class to give an idea of all three in practice:

```
class SpaceInvaders {
    private final List<Invader> invaders = new ArrayList<>();
    private final Display display ;

    // constructor and other methods omitted

    public void update() {
        // Move invaders
        for ( Invader i : invaders ) {
            i.updatePosition();
        }

        // Draw invaders
        for ( Invader i : invaders ) {
            i.draw( display );
        }
    }
}
```

My first refactor step would be to break up the method. Not because it's too long in this case, but so I can turn those comments into methods.

```
class SpaceInvaders {
    private final List<Invader> invaders = new ArralyList<>();
    private final Display display ;

    // constructor and other methods omitted

    public void update() {
        moveInvaders();
        drawInvaders();
    }

    private void moveInvaders() {
        for ( Invader i : invaders ) {
            i.updatePosition();
        }
    }

    private void drawInvaders() {
        for ( Invader i : invaders ) {
            i.draw( display );
        }
    }
}
```

This has extracted the two for loop blocks into private methods. The comment names were then turned into method names.

Doing this is really helpful. It makes the code more self-documenting. Best of all, it stops comments going out of date.

A comment is effectively duplication in the code. It says in English what a block of code is doing. By turning this block into a method, we get the chance to add a method name to explain what that code is doing. If the code changes, you are more likely to change the method name than to change both the method name and a comment.

As a matter of style, as the class is called SpaceInvaders, I would probably shorten the names to move() and draw(). We already know the class is all about Invaders. The shorter names are clear because we have context from the class name.

The refactored public method now becomes a high-level overview of what it does and not how it does it. Again, this is all about bringing the problem domain to the fore, and hiding implementation details.

Change Method Signature

This is an excellent refactoring to emphasise the problem domain.

A genuine example from a Web Conferencing startup I worked at was this:

```
1  public void Chat {
2    public void sendMessage( String toUsername ) {
3      // code ...
4    }
5  }
```

We had a simple string to represent the name of a user. This was passed into methods as above. We used this everywhere. It was one of those early design decisions you make so you can just get moving. A String might not be the best long-term decision, but anything else seemed over-engineered at the time.

This worked well. But we had two problems in the end.

One issue was that we needed to change the code so that any given user was homed on a particular server. The design later changed so that we needed both username and home server information to be passed around. It was tough to change, because all we had were Strings, everywhere. Search and replace just turned up thousands of Strings, as you would expect. They were not always consistently named 'username', either.

The other issue was simply that a String based username leaves you guessing what format the String should be in. What *exactly* do you pass in? 'almellor'? 'al@webconf.com'? '1172397f'? It is impossible to know.

The answer was to Refactor > Change Method Signature for each method like this. Instead of accepting a String, it would accept a User object:

```
class Chat {
    public void sendMessage( User recipient ) {
        // code ...
    }
}
```

This clarified the code, prevented a whole class of bugs and gave us the freedom to add the home server information into that User object.

As you can see, the whole 'abstraction level' has been raised again. We can now clearly see that a Chat can sendMessage to a User. It's written in plain English, right there in the code. We see the 'what' more prominently than the 'how'. We see 'User' and not 'String'.

Extract Parameter Object

I love this one. This refactoring causes some great emergent design insights. It is used once you see a method with a parameter list that seems too long.

Back to Space Invaders for an example.

```
class InvadersGame {
  public void draw( int x, int y, Image currentFrame,
                    int deltaX, int deltaY) {
    // code
  }
}
```

Here we have a draw() method with a handful of parameters. There's nothing immediately wrong with that. But it's making me feel uneasy. It feels like there is some better code just waiting to emerge.

The parameters look suspiciously related. An (x, y) coordinate pair. A (deltaX, deltaY) motion vector. An Image with a current animation frame.

Hmmm.

They sound like … well, they sound like the insides of an Invader. Let's pull them out as an Invader object, using Refactor > Extract Parameter Object:

```
class InvadersGame {
    void draw( Invader i ) {
        // code
    }
}

class Invader {
    private int x;
    private int y;
    private Image currentFrame ;
    private int deltaX;
    private int deltaY;

    public int getX() {...}
    public int getY() {...}
    public Image getCurrentFrame() {...}
    public int getDeltaX() {...}
    public int getDeltaY() {...}
}
```

I've omitted the code inside those getters, fields and constructors for the Invader object. The structure is important here, not the detail.

We have taken all those parameters in the original draw() method and put them into a new class - called a 'parameter object'. The existing draw() method will then replace each use of the original parameters with i.getX(), i.getY() and so on.

Invader is not yet an OOP object with behaviours. It's a pure data structure.

Here is where it gets interesting. If we move the code from the original draw() method into a draw() method on the Invader class, we can get rid of all those getters:

```
class InvadersGame {
    void draw( Invader i ) {
        i.draw();
    }
}

class Invader {
    private int x;
    private int y;
    private Image currentFrame ;
    private int deltaX;
    private int deltaY;

   public void draw() {
       // code can use x, y and so on. It's an object!
    }
}
```

You can see this is less procedural and more like the OOP we have been learning.

Can you also see that it is starting to look 'back to front' - and more descriptive?

Our old code would have called draw() with a load of primitive values from somewhere.

This refactored code creates an Invader object that knows how to draw itself and move around. This has made the InvadersGame.draw() method shorter, more obviously about an Invader - and redundant.

Extract Parameter Object is such a powerful refactoring that it turns procedural designs inside out to become Object Oriented designs. You can expect the changes to ripple out, dragging more and more objects into play.

By the end, your code will have that simplicity of objects collaborating together, with hidden 'secrets' and meaningful names.

Can we refactor anything into anything else?

I heard a fascinating talk on this, by local Manchester XP Surgery legend Kevin Rutherford.

His view was 'Not really'. I agree. Refactoring is powerful, but it has limitations.

Refactoring gets hindered by two things: lack (or excess) of tests and hidden couplings.

The first one sounds surprising, but it's true. Your tests need to be 'just right', just like Goldilocks' porridge.

Your tests must support the refactoring you want to do. You need sufficient test coverage at the same level as your calling code. If you have that, you can safely refactor knowing that any errors will be caught.

Often, that's not the case. There might be too few tests here. Perhaps all the testing effort went closer-in to individual classes you now want to refactor. In that case, you will probably invalidate the test code as you change the production code.

These classes are the 'how' - implementation details - of the higher level class you want to refactor. Tests will be coupled up to those. As you change implementation details, those tests break. That means you no longer have proper coverage of features.

Hidden couplings are tricky as well.

You might find that you break some tests as you refactor, yet it's not apparent why. As you debug, you find that two different areas of code need to be changed in lockstep with each other. There might be a string constant, for example. You refactor that in one class, but forget to refactor it in the other, then find out that the code no longer works.

Those kinds of issues are challenging - and often painful - to sort out.

Refactoring is a commonplace, recommended and excellent tool. Just be aware that it cannot overcome all design flaws, practically speaking. Spending a little time upfront to sketch a small piece of OO design still pays off.

The Fowler book remains the best resource there is on this subject.

Hexagonal Architecture

What happens if you turn polymorphism and Dependency Inversion up to 11?

You get Hexagonal Architecture.

First described by Alastair Cockburn in 1993, then called ‘ports and adapters’, it has been written about ever since. Yet very few people I’ve worked with seem to use it.

That’s a shame. It solves a big problem for us: external systems.

The problems of external systems

External systems exist outside our application code, outside of its operating system process and outside of its memory space.

There are many examples. Database, files, REST controllers, web servers, I/O hardware, printers, GUIs, displays, keyboards are common. You can probably think of more.

These things are a right royal pain when it comes to testing our software.

Think about keyboard input from a user.

We can easily write Java code to accept keyboard input.

But how do we test it?

A traditional TDD unit test would need its Arrange, Act, Assert steps.

How do you arrange for keys to be physically pressed on a keyboard by a unit test?

The simple answer is that you can’t.

The Test Pyramid

The standard answer to this is called the Test Pyramid.

Unit tests aren't the only game in town. There are other kinds of tests.

Integration tests take several pieces of code, join them up and test them as a whole. Generally, some of these pieces will be external systems. So you might test some business logic against a running database.

End-to-end tests expand on that idea. You wire up the entire system and test it just as a user would. You drive the user interfaces and run against actual databases and real web services. The whole thing is as 'live-like' as possible.

Contract tests are interesting. These split a test into two directions. They are used to break apart testing against external service.

With these, you test twice. First, you test that your application logic can call a stub for the service correctly. That's what 'testing the contract' means - the contract between your application and whatever it talks to.

Second, you test the contract between the code that talks to the external service and make sure that it, in fact, does talk correctly.

There are browser tests, which are a form of end-to-end testing, that use a special kind of web browser. You can script mouse movements, taps and keystrokes as if you were a real user. It enables capture of the HTML and data sent to the browser for checking.

Each level of testing gets more problematic. An integration test is slower than a unit test, as it has to set up and control the external system. End-to-end tests are slower still. They are somewhat harder to write.

The worst problem is about test *repeatability*. We want tests to either always pass when the code is right or fail when it's not.

Sadly, higher-level tests tend to be 'flaky'. The code may be right, but something in the external system might still cause the test to fail. It is often simple things, like having the wrong data in a database, or a slow response.

Flaky tests can also be caused by intermittent problems. Things like network failures and database connection limits are all problems here.

By way of example, one particularly tricky one to diagnose was an integration test against a search web service. The service was behind a load balancer.

In the test environment, the load balancer was set to round-robin mode. We had two test instances of the service. The load balancer would route requests to service 1, then the next request to service 2. At least *it did* - until one service was switched off without anybody telling us. Oops.

All the integration tests passed then failed, passed then failed like clockwork.

It was purely down to the 50/50 routing of requests from the tests to one good service and one down service.

You can also imagine that there might be test duplication amongst these different levels of tests.

A small unit test that confirms that some kind of user option is available might be duplicated by a check box's browser test.

As with all duplication, this is a problem. It makes things harder to change and harder to understand.

Having live-like test environments to support higher-level tests has its problems. Hosting costs go up, as we need dedicated test machines. We may have data protection and privacy issues if our test data relates to a real person. And the slow running speed hampers development.

The Test Pyramid is a simple visual aid. It reminds us to use mostly fast unit tests. Back those up with slower, flakier integration tests where we must. But not as many. Then fewer still browser and end-to-end tests.

By minimising the amount of slow, potentially intermittently failing tests in our system, we maximise our ability to test. We increase velocity and ship sooner.

So if the slower tests are less than ideal, the obvious solution is to avoid the problem and use more FIRST unit tests.

If only it weren't for those pesky external systems …

Removing external systems

Of course, we can't just throw out the database and the keyboard. Our program wouldn't work.

But we can eliminate those details from the bulk of our code. We already know how to do this. We can use the Dependency Inversion Principle.

Let's look at removing a database dependency from our 'NotFaceBook' app.

Our NotFaceBook system needs to lookup a User profile, based on a logged in user ID. We can display a profile page based on whatever the user stored.

At first sight, this needs database code. It must do. The user stores their profile data in the database: name, profile selfie, and favourite cat photo. Our code must fish it out from there.

That's the *how* of how we do this: the implementation details.

But *what* are we actually doing, at a high level?

We are finding a User object by its user id. We can assume that User object knows how to supply profile information as its secret knowledge.

We can code that up:

```
1  interface UserRepository {
2      User findUserById( String userId );
3  }
```

UserRepository is an interface that allows our code to access the User and its profile information, without knowing anything about databases.

We can write different implementations. We can just as easily code an implementation that talks to a file as one that talks to a Postgres database. Or MongoDB. We can even code a test stub, returning fixed, known data.

We're using the word Repository here as the most general way of describing "Anything that stores, generates, and obtains User objects, somehow". We're using it to suggest that the precise storage details are irrelevant.

Maybe it is not even stored. For example, in some games, the userID value could be used to *generate* that profile information on the fly - not store it. That is still a 'repository'.

We can use it in calling code like this:

```
class NotFaceBook {
    private final UserRepository users ;
    private final Display display ;

    public NotFaceBook( UserRepository users, Display display ) {
        this.users = users ;
        this.display = display;
    }

    public void showProfile() {
        String userId = "almellor_19019";

        User u = users.findUserById( userId );

        u.displayProfile(display);
    }
}
```

We use constructor dependency injection to pass in one of our concrete implementations of UserRepository.

You can see that UserRepository is an *abstraction*. It is the essence of what finding a User object in a database means. UserId in, User object out.

This is a classic dependency inversion.

Our application code - the NotFaceBook class - no longer depends on details of the database. It depends only on the abstraction.

This makes unit testing simple. We can simply create a StubUserRepository class that always returns a representative User object. The stub never goes anywhere near a database.

This gives us all the benefits of a FIRST Unit test. It avoids problems of database testing.

Database testing for real is slow and rather error-prone. You tend to get several issues:

- Your test code accidentally connects to the real database. Oh dear. Catastrophic when you test 'deleteAllUsers()'

- The test database does not have a User for the UserId you supply. The test will fail as a false negative. The code was right, but the data was wrong.
- Testing addition of data fails the second time. The first time, 'User1' does not exist and gets inserted. The second time, the database rejects the insert: 'User1 already exists. Primary key conflict: UserId'

By using a simple in-memory stub, all of those problems go away. Instead, we get:

- Most of the system under fast unit test
- A clear domain model - the 'what' of the system
- An 'inside out', experimental approach is possible
- Simplified changes of things like databases and web server technology

The Hexagonal Model

You can see that our code now consists of two main parts. The application logic, which is free of external systems. Then the external systems, which are free of application logic.

We give those parts names:

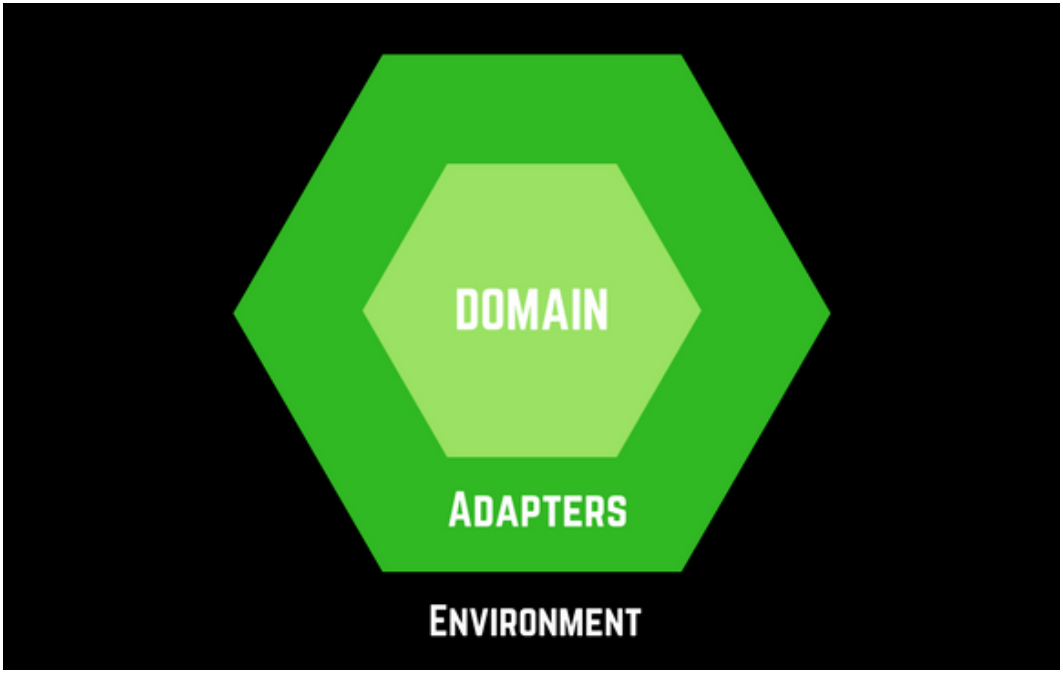
- Domain Layer - application logic, no external systems
- Adapter Layer - external systems, no application logic

Graphically, we can draw this as two concentric hexagons.

The inner hexagon is our domain layer. These are pure application classes. The domain layer is where application logic and our abstractions of external systems live - interfaces like UserRepository.

The outer hexagon contains the adapter classes. They hold the details of talking to the external systems, implementing those interfaces.

We use hexagons so that each outer face can represent one external system. Mosts code has no more than six external systems it talks to. Hexagons are easy to draw. They make a good general-purpose graphical template.



Hexagonal Architecture

We have two simple rules to get the design right:

- The Adapter Layer implements interfaces and uses classes from the Domain Layer. It may depend on the domain layer.
- The Domain Layer MUST NOT use classes from the Adapter Layer.

One way only: Adapter depends on Domain

The domain layer knows nothing about the adapter layer. There are no dependencies at all on external systems. They simply do not exist in our domain layer. There is not a single reference to a database, web server or anything else. It knows only about business logic - the core of our problem.

This aids readability and testing.

The adapter layer is the only place where details of the external systems appear. This is the place for complicated SQL statements, HTTP calls, HTML templating and the like.

This split helps us limit what code needs to change if we change technology in an external system.

Want to change a SQL database like Postgres for a NoSQL one like Mongo? You write a new adapter class and inject it in. Your domain layer stays unchanged. The domain layer tests stay unchanged. This is a huge benefit.

Hexagonal Architecture relies on polymorphism and the SOLID principles to work. The Adapter Layer implements interfaces that are fully substitutable because they obey LSP. Each adapter class does only one thing, following SRP. The Domain Layer is then open to changes in external systems without needing modification. It obeys OCP.

Inversion / Injection: Two sides of the same coin

In casual talk, developers say about Dependency Injection and Dependency Inversion as if they were the same.

They're not. But the two are closely related.

Dependency *Inversion* happens at design time. That's when we create abstract interfaces like UserRepository.

Dependency *Injection* happens at run time. That's when we pass implementations like PostgresUserRepository to calling code consuming a UserRepository interface.

They combine together in Hexagonal Architecture to make an extensible, testable system.

Handling Errors

Wouldn't it be great if everything went to plan?

…And you're back in the room. Real life is messy. Things go wrong.

Software designs get their fair share of the unexpected, too.

Confused users, deliberate attacks and hardware failures are all things that mean business as usual will not be happening today.

The question is 'what do we do about that?'

Three kinds of errors

Whilst a whole book could be written about 'stuff that goes wrong in computer systems', all errors boil down to one of three types:

- Errors we can fix
- Errors we can't fix
- Errors we don't care about

Examples of each kind would be 'User typed wrong number' (ask again), 'Disk drive has crashed and lost all data' (oops. Bad luck), 'Temperature sensor not working, when temperature feature is off' (ignore).

Detecting an error is one thing. But then we must let our software know about it. We need a mechanism to do that.

Java and OOP have several ways available. Each has its own trade-offs.

Let's use a running example of our UserRepository. The User findUserById(id) method should always return one and only one User object. The User object has one method showProfile(), which displays profile information - on a good day.

To keep code samples small, we'll only show the method in isolation, and it will only return the error case. Don't go looking for the 'happy path' code, because there isn't any.

How can we tell calling code that something went wrong?

The null reference

In the beginning, there was null. And the null was shapeless. And dangerous. And the null filled the Java landscape …

```
User findUserById( String userId ) {
    return null ;
}
```

We've all seen this. It is baked deep into Java since Java 1.0.

The concept of null was invented by Computer Science luminary Tony Hoare. He also invented Quicksort, which was a good idea. Unlike null.

Tony described null as his 'billion-dollar mistake'.

Returning null is ok, in theory. The method either returns a valid User object if one is available or returns null.

Calling code must check for the null return and decide what to do.

The problems with this include cluttering up the code with error paths, all those extra if statement paths to write and test - and the chance of forgetting the test.

If findUserById(id) returns null, any further access will spectacularly fail:

```
  User u = users.findUserById( "alan0127" );
  u.showProfile(); // BLOWS UP if u is null with NullPointerException
```

Returning null complicates calling code with two paths. It is an accident waiting to happen. You will forget to check for null at some point.

You will. You know you will. I know you will. I know I have!

Everything that follows is a rather desperate attempt to design-out the bugs caused by unhandled null references.

Null object pattern

A reasonable attempt was proposed by Bobby Woolf in 1998. Known as the Null Object Pattern, this uses polymorphism to return one of two objects. Either the valid User or a NullUser object will be returned:

```
User findUserById(String id) {
    return NullUser();
}
```

NullUser is fully (LSP) substitutable for User. We can do this either by having User as an interface and creating RealUser and NullUser implementations. Or, we can simplify a bit and create a User class with a NullUser subclass.

Either way, the NullUser class has a 'no action' method for showProfile:

```
class NullUser extends User {
    public void showProfile() {
        // No Action
    }
}
```

> Always comment with No Action. It shows it is intentionally empty.

When the calling code runs, it will be given either a User object, or a NullUser object if there were errors. It then calls showProfile() on whichever object it got.

It is safe to do this because:

- It will always receive an object. There will be no null to check for
- The NullUser has a method that implements correct behaviour: do nothing

The NullObject pattern is very effective. It designs-out null reference bugs by not using null. It simplifies calling code - our big goal in this book - by eliminating a conditional branch. That simplifies TDD testing. All of that speeds up reading and understanding this code.

Zombie object

There's something a bit grubby about NullUser, though: That whole subclass thing.

I've heard complaints from pairs that "We only need one class and now you're adding two or three! Come on, Al - keep it simple!"

It's a fair point. Although you can argue that you *don't* only need one class. You are representing two completely separate ideas - a User that exists and one that doesn't.

But arguing with your pair is all a bit tawdry, really. So let's go and make a nice hot cup of tea each, and have another think.

What else could we do?

The 'Zombie object' is so named because the way it represents null is to be 'the walking dead': The outside of the object looks normal, but the insides are bad.

We must modify our User class to add a 'zombie detection' method:

```
1  class User {
2      boolean isValid() {...}
3
4      void showProfile() {...}
5  }
```

The new isValid() method tells us whether the object has valid data or not. If isValid() returns false, you must not use the object. You must check this first - every time you use the object.

Calling code has the conditional just like the null check. But it reads nicer:

```
User u = findUserById("alan0127");

if ( u.isValid() ) {
    u.showProfile();
}
```

This is a judgement call. Is this any better than null and null checks?

My view is "sometimes, yes". The minor improvement in readability can be worth having. Using 'Optimise for Clarity' as our guiding principle, sometimes this makes the calling code more descriptive.

You'll have to decide for yourself. But if you see code like this written by others, this is why. That distinctive pattern of `if (object.isValid()) { /* use object */ }` is the signature of a Zombie object.

Exceptions - a quick primer

There is another mechanism entirely to report things have gone South in Java.

Exceptions have been present since the start. They are an alternative control flow mechanism. Because of that, they are rather controversial.

Let's see how they are used in this code snippet:

```
class ExceptionsDemo {
  public void demonstrateExceptions() {
    try {
        System.out.println("This will print out 1");
        methodWhichThrowsAnException();

        // Line below will never run ...
        System.out.println("This code will never be run 2");
    }
    catch( RuntimeException e ) {
      System.out.println("You will see this - exception caught");

```

```
13          System.out.println("Exception message was: " + e.getMessage());
14        }
15    }
16
17    private void methodWhichThrowsAnException() {
18          System.out.println( "This will print out 2" );
19          throw new RuntimeException("Used to report an error");
20
21          // Unreachable code - this will never run
22          System.out.println("This code will never be run 1");
23    }
24  }
```

The big idea here is that the keyword 'throw' when it runs changes the execution flow. It stops running code in sequence at that point. The code will immediately exit whatever method it is in, without executing any more lines of that function. It is a form of early return statement.

Significantly, if the method was supposed to return something, that will not happen. Nothing is returned. Not null. Nothing at all.

The throw keyword is designed to be paired up with the try and catch keywords, in a construct called a try-catch block.

If some method throws an exception inside a try block, then code execution will resume starting at the first line of the matching catch block.

The throw keyword has a parameter. This parameter should be an object instance that either is - or extends - one of the two built-in classes Exception or RuntimeException.

The catch keyword has a matching parameter that receives the object used in the throw statement. It is a way of passing information with the exception object, to give diagnostics information to the catch block code.

In the example, the text passed into the RuntimeException object's constructor will be displayed by code in the catch block.

You can see how exceptions allow a method to be simply terminated early and control passed to a catch block - called an exception handler.

Exceptions are a powerful way of signalling 'There was an error. I could not continue executing that method. Here's why'.

But they are controversial.

In practice, there are three ways of using exceptions.

Design By Contract, Bertrand Meyer style

Inventor of the Eiffel language Bertrand Meyer had an excellent idea about methods and exceptions.

He proposed that a method call was a contract between the code inside the method and its calling code. Our findUserById(id) method declares a binding contract: if you call that method, you *will* receive a valid User object.

Meyer saw exceptions as the way to indicate 'that contract cannot be satisfied'. You would throw an exception in the findUserById() method if for any reason a User object could not be returned. The contract had been broken.

In this view of exceptions, they are commonplace. You can write calling code that relies on the contract being met:

```
try {
  User u = findUserById( "alan0127" );

  // if we reach here, we are guaranteed User u is valid
  // due to our contract
  u.showProfile();
}
catch( Exception e ) {
    // if we reach here, we know the User object was invalid
    // we can handle that
}
```

Seen in this way, exceptions clean up calling code. They eliminate null passing. They eliminate conditionals. Indeed they design-out the idea that a returned object might be invalid. That is not possible. You either get your result or an exception.

Java fully supports this. It even provides an extra tool. If exception classes are made to extend Exception, the Java compiler will enforce that the exception is either caught at the call site, or is declared as being thrown up the call stack.

By contrast, anything extending RuntimeException doesn't have as much scrutiny. It's actually the preferred way, these days, of using exceptions.

Fatal errors: Stop the world!

Another view on exceptions is that you only use them for an unfixable problem that needs to stop the program immediately.

These will be a RuntimeException (or subclass) that is thrown and never caught. The Java Virtual Machine will then kill whatever thread was running when that uncaught exception happened. If that happens to be the main thread, the program exits.

In this view of the world, we cannot use exceptions to represent errors we can fix, or errors we don't care about. We have to decide at the point where we throw that everything is lost. Perhaps our disk failed, and we cannot retrieve data.

Perhaps the input data is corrupted, and we have no idea how to process it. Throw the exception and get outta there.

This extreme approach is often advocated by those who view exceptions *not* as a way of reporting abnormal behaviour, but as a 'giant goto statement'.

Because exceptions affect control flow, breaking out of our neat structures and call hierarchies, they can seem a bit hacky to some. Just like we invented if/else and while/for loops to avoid using goto - based on the David L. Parnas' 'Goto considered harmful' paper from 1972 - then exceptions are seen as another goto that we should avoid.

Combined approach: Fixable and non-fixable errors

A third way combines the two.

We use exceptions to represent fixable errors. We use different exceptions to represent non-fixable errors.

The classic example is a timeout on an access to an unreliable system, say a REST service call that may fail if the network is down:

```
for (int attempts = 0; attempts < 3; attempts++) {
    try {
        callUnreliableService();
        return ; // happy path - exit the method
    }
    catch( ServiceTemporarilyUnavailableException se ) {
        Thread.sleep(1000); // wait one second
    }
}

throw new ServiceUnavailableException();
```

This snippet calls a method 'callUnreliableService()'. That may fail by throwing ServiceTemporarilyUnavailableException. That may well be a temporary glitch. So we wait a second, to give things a chance to recover, then try again.

If it hasn't worked three times in a row, we admit defeat and throw the ServiceUnavailableException. This exception tells the calling code that it has a problem, and should work around it, or terminate the program.

This whole approach is based on the idea that each software subsystem can only really report that it is not working. But it cannot really know if that is a problem to its calling code or not. The calling code is best placed to know how to handle this event.

Our code has handled all the exceptions it knows how to - then delegates upwards when it has nothing left to attempt.

Which approach is best?

Blimey! I'm not going to get dragged into that massive argument.

But you need to ideally pick one approach and stick to it. Otherwise, people get very tetchy as they end up converting between approaches.

Anyway - now we can get back to our missing user problem.

NullPointerException

We've touched on this earlier. If you return a null, then attempt to call a method on that null, Java goes boom.

To be fair, Java goes a lot less boom than C or C++ do. It is actually fairly civilised. It throws a NullPointerException.

One approach to indicate that we cannot return a user object is simply to return null. We can omit the null check and catch the exception instead:

```
try {
    User u = findUserById("alan0127");
    u.showProfile();
}
catch( NullPointerException npe ) {
    // handle missing user
}
```

It's similar in nature to the return null with a null check. You'll see this in the wild. I'm not sure I really like it - even though I've used it myself.

Application Specific Exceptions

One small improvement is to define a more descriptive exception:

```
class UserNotFoundException extends RuntimeException {
    public UserNotFoundException( String diagnostics ) {
        super(diagnostics);
    }
}
```

This allows us to indicate an error like this:

```
User findUserById( String userId ) {
    throw new UserNotFoundException("<reason why not available>");
}
```

and the calling code becomes a little more clear:

```
try {
    User u = findUserById("alan0127");
    u.showProfile();
}
catch( UserNotFoundException unfe ) {
    // handle missing user
}
```

This is an implementation of Meyer's Design by Contract idea.

Error object

There's something a bit off about exceptions and Object Oriented Programming.

OOP is all about behaviours and objects collaborating. Exceptions are all about disturbing the control flow in a very procedural sense. Those two things don't really match.

Error objects are very different to exceptions. They are normal objects that will be returned out of methods. Those methods will run and complete normally.

They work by using polymorphism and tell don't ask. Twice, as we'll see.

Let's go back to our findUserById(id) running example. It can either return a valid User or tell the calling code that there isn't one.

Using OOP, we can define each case as a separate object:

```
class ValidUser {
    private final User user ;

    public ValidUser(User u){
        this.user = u;
    }
}

class InvalidUser {
    // Nothing to represent ...
}
```

Which of course sounds like a wonderful application of SRP, until we ask 'How do we return them?"

Ah.

Java functions can only return two separate objects if there is some kind of inheritance relationship - extends or implements - between them. That's not the case here.

We can fix that by simply making them both implement an interface, of course.

Let's make findUserById(id) return a UserResult interface:

```
interface UserResult {
    // empty
    // ... we'll come back to this ...
}
```

which can be returned simply:

```
UserResult findUserById( String userId ) {
    // ... code
}
```

and the inside of that method can return either of our objects, if we make them implement our empty interface:

```
class ValidUser implements UserResult {
    private final User user ;

    public ValidUser(User u){
        this.user = u;
    }
}

class InvalidUser implements UserResult {
    // Nothing to represent ...
}
```

Which is all great. We're making steady progress.

But hang on - was that an *empty* interface? What use is that?

It's a fair point. Currently, the only thing we could do in the calling code is call instanceof to find out what concrete class was returned. *That is worse than a null check!*

We'll need to do better.

Thinking back to basic OOP, what we really need is a common behaviour. Something that unites both 'InvalidUser' and 'ValidUser'.

Obviously, that won't be something like a getName() method, because the InvalidUser does not have one.

The fix for this is to use the Tell Don't Ask principle. We can tell our object to 'do its thing'. Quite literally. We add an applyTo() method:

```
interface UserResult {
    void applyTo(); // ... we'll come back to this...
}
```

That's better. Our calling code now no longer needs to know about which concrete class we returned. It only has to call 'applyTo()' then the relevant object will apply itself.

Ah. What will it apply itself to, exactly?

We're missing another interface:

```
interface UserConsumer {
    void validUser( User u );
    void invalidUser();
}
```

which we can use in the UserResult interface:

```
interface UserResult {
    void applyTo( UserConsumer consumer );
}
```

We have to update our ValidUser and InvalidUser with the new method:

```
class ValidUser implements UserResult {
    private final User user ;

    public ValidUser(User u){
        this.user = u;
    }

    public void applyTo( UserConsumer uc ) {
        uc.validUser( user );
    }
}
```

```
12
13 class InvalidUser implements UserResult {
14
15     public void applyTo( UserConsumer uc ) {
16         uc.invalidUser();
17     }
18 }
```

Finally, this makes sense. We tell our result objects to apply themselves, but this time to some new consumer object we will make.

This new consumer object will have two methods validUser, passing the User object as a parameter and invalidUser. In each case, the consumer object is guaranteed to either be given a valid User object or that no such object exists.

The idea here is that findUserById(id) now returns an object. We can tell that to applyTo() some other object we have created to handle results. That object knows how to handle either a valid or an invalid object.

This is a 'push' solution. We use Tell Don't Ask across two objects to resolve the valid/invalid behaviour.

If you think back to a null check solution, you would have two blocks of code that ran; one for null, the other for not null. Now, we have two different methods that get called for each case.

As an example, lets create a simple object that will print the profile of a valid User and write an error message to the console for an invalid one:

```
1 class ProfileTextDisplay implements UserConsumer {
2     public void validUser( User user ) {
3         user.showProfile();
4     }
5
6     public void invalidUser() {
7         System.out.println( "Sorry, we don't know that User." );
8     }
9 }
```

We can create calling code that looks like this:

```
ProfileTextDisplay display = new ProfileTextDisplay();

UserResult result = findUserById( "alan0127" );
result.applyTo(display)
```

That's the final result.

The calling code - at last - looks clear, simple and neat.

There are no nulls, no exceptions, no conditionals. Just objects that know how to do the right thing as part of their secrets.

The plumbing code we need is quite extensive: UserResult, ValidUser, InvalidUser, UserConsumer. But the trade-off is that each piece is nicely separated out. It is easy to unit test in isolation.

Optionals - Java 8 streams approach

Java 8 introduced some pure functional ideas with its use of streams.

The big idea was to combine iterating over collections and transforming each object in that collection in one step.

A typical example would be to take a List of our User objects, find the first one with username starting with 'Al' then change that name to all uppercase:

```
String fetchDisplayNameForFirstAl( List users ) {
    return users.stream()
        .filter( user -> user.nameStartsWith("Al") )
        .findFirst()
        .map( user -> user.getName().toUpperCase() );
}
```

This approach combines pure functions into a pipeline. You'll see that this doesn't really live in the world of OOP. Typically, classes are abused as pure data structures - getters and setters, no real behaviours, no encapsulation. The external functions add behaviour onto that raw data.

As a result, Java needed a new way to handle errors inside these chains of functions.

In the snippet above, what happens when there are *no* users whose name starts with Al?

That method simply could not work as written. There needs to be some way to signal to that function chain that 'the data was not found'.

Java 8 borrowed from other functional languages to give us the concept of an *optional value*. Just like it says, this is a value that may, or may not, contain the data we want.

At its simplest, we can make our findUserById(id) method return an optional User:

```
Optional<User> findUserById( String userId ) {
    // ... code
}
```

Inside the method, we can use the construct `return Optional.of( user );` to return a valid User object, or `return Optional.empty();`, to indicate that no valid User exists.

The calling code is where this gets interesting. We can choose to handle these cases either with a simple if statement, or by supplying a 'lambda function' to act on our valid User.

The if statement resembles a null check:

```
Optional<User> result = findUserById( "alan0127" );

if (result.isPresent()) {
    result.get().showProfile();
}
else {
    System.out.println( "Sorry, we could not find that user" );
}
```

Technically, there is no improvement over a null check. But I do think the readability of this is much better. It is *intentional*. We can clearly read that this code is all about an optional value and code that handles the value if it is present.

But Java 8 provides its own way of eliminating the if statement:

```
findUserById("alan0127")
    .ifPresent( user -> user.showProfile() )
    .orElse( System.out.println("Sorry, we could not find that user"));
```

This is a simple refactoring of the if statement above, but using the ifPresent() method of Optional. This takes a *lambda function reference* as an argument - we pass in some code to execute if the value is present. The chained method orElse() similarly takes in a lambda function reference to run if the value is not present.

This has a readability advantage - once we are familiar with the syntax of method references. It also has all the usual advantages of avoiding null checks.

This style is gaining popularity, as newer (Java 8 and onwards) libraries adopt Optional<>.

Review: Which approach to use?

We've covered several ways of reporting errors. They are all options. You will see all of them out in the wild.

The most basic - the null check - we simply can't avoid. It is too baked into both Java libraries and Java programmers alike. Null checks are a thing. They are here to stay.

I don't recommend them, though. Nulls clutter up calling code and introduce the possibility of an uncaught NullPointerException. That's enough to take the entire app down.

For new Java 8 and onwards code, the Optional<T> approach can work well, especially combined with @NotNull annotations that can be checked via static analysis tools.

It's a shame that Optional wasn't made into its own primitive type. In that case, an Optional return would be guaranteed to never be null. As it is just a regular object, you can return null instead of an Optional, defeating the whole point. I can see why such a fundamental change to Java was not introduced. But I'm allowed to grumble at the missed opportunity.

Error objects require a lot of plumbing. But the calling code is pure OOP and very clean. They have a lot of advantages once you get used to them. They are easy to use

in more advanced ways, like adding ‘error(ErrorInfo reason)’ kinds of methods, in addition to valid() and invalid() methods.

Exceptions are a fairly clean approach. Detractors say they are ‘a glorified goto’. That’s fair. They point out that exceptions should - in their view - only be used for things truly unrecoverable, errors that nobody was expecting. I hope I’ve shown you that Bertrand Meyer’s ‘Design by Contract’ shows an alternative viewpoint.

Zombie objects are surprisingly useful. Combining minimal plumbing with enhanced readability, they form a ‘cheap and cheerful’ error object.

When deciding, you need to think through the trade-offs given your existing code, your existing developers and the *feelings* towards the various approaches.

You’ll often find it is the ‘feelings’ part that is the deciding factor.

Design Patterns

It can be hard work, this mapping behaviours to different objects. Just like it is hard re-inventing a house from scratch, for every house on a street.

By the mid-1990s, the object oriented world had discovered the same thing. A lot of objects had the same kind of 'shape'. Just like houses on a street.

To understand what a pattern is, let's come at this in reverse. Imagine you've written an object that has three data fields:

- A name
- A timestamp
- An amount

These are the secrets of your class. Without overthinking, what's the first thing that comes into your head that this represents? No cheating, now! 3, 2, 1 … Go!

I get:

- A line item on a bill: "Rioja 125ml, 8.30pm, £2.95"
- A lease: "12 month rent of 15A Acacia Avenue, 1 Jan 2020, £2,150"
- A membership: "Alan's Geek Club 1 year, 20 Oct 2020, £49.95"
- A discount code: "Xbox Live, expires 6 June 2021, $79.99"

There are plenty more examples.

Many different kinds of objects share a similar "shape".

By the mid-1990s, this had been learned, but from the other way around. The behaviours of a Customer object were seen to be more or less the same as a Member object, which were more or less the same as a Patient object. Sure, there were specific extra behaviours - but why the similarity? What was behind that?

Thinking back to our User.greet() example, we might have a Customer class with a welcome() method. It's a different class name, and a different method - but the same basic idea.

This is known as a *Design Pattern*.

Objects form *patterns* at the design level. Groups of objects jump out as having a similar behaviours in one subject area as they do in another.

A pattern gives us a jump-start on our designs. We can start with something similar and tailor it to our needs. We're buying an off the peg, mass-produced object design and tweaking it, rather than cutting cloth ourselves.

And mangling our metaphors. I do apologise.

The 1990s jumped on this idea with vigour. It gave rise to several books, papers, journals and conferences. Some were good, some were seminal, some were rather forgettable.

The rest of this chapter is my take on patterns. These are the ones I use professionally, with my explanations of how I understand them.

Before we dive in, I want to raise one important distinction about where patterns fit. It's related to mechanism and domain.

Mechanism and Domain

Any piece of software consists of two broad parts: Mechanism and Domain.

Mechanism covers things which are the details of how stuff works. Things like assignments, loops, branches and object definitions. The insides of methods,

Domain is the actual subject matter area. Things like Rental Agreements, Inventory, Orders, Cancellation Policy, User. The outside of methods and their classes.

Clear designs **emphasise domain** and hide mechanism

This distinction is important when studying design patterns because patterns exist at both levels. They solve different problems.

Earlier patterns solved mechanism issues and came up with things like Iterators and Strategy patterns. Later attempts looked at solving domain problems, coming up with things like ItemDescription-Item and Order-OrderLineItem patterns.

Like the rest of this book, the split is about *how* things work versus *what* they achieve.

Here are the patterns I have found most useful. I'll use some initials to say which book I found them in first:

- **GoF** Design Patterns by the 'Gang of Four' ISBN 978-0201633610
- **DDD** Domain Driven Design by Eric Evans ISBN 978-0321125217
- **Coad** Java modeling in color, Peter Coad et. al. ISBN 978-0130115102
- **C2WIKI** The C2 Wiki at https://wiki.c2.com/
- **PoEAA** Patterns of Enterprise Application Architecture ISBN 978-0321127426
- **Own** Patterns I have picked up from work or devised ISBN 978-1527284449

Patterns: Not libraries, not frameworks

This idea of *patterns* is important. These are not cut-and-paste code snippets. Nor can they be baked into a library.

The code examples below are not intended to be copied line for line. The idea is to understand how that code works, then apply the principles to your own situation. Each pattern uses polymorphism to achieve its effect - so look out for where that is used. That will give you the key to unlocking how it works.

Strategy

[GoF] Gives you pluggable *behaviour*

Strategy is used when you have an object that needs a way to change its behaviour. Depending on configuration, or in response to external events, it needs to do some processing differently. You could do this with conditionals like switch statements. But you want to respect OCP and keep that object unmodified.

Injecting a Strategy object enables this.

As an example, employees can have different bonus schemes as they hit different targets.

Here is our Strategy pattern interface. It will allow us to make various implementations:

```
interface BonusScheme {
    void applyTo(Money pay);
}
```

We can now inject that in our Employee objects:

```
class Employee {
  private BonusScheme bonus ;

  public Employee(BonusScheme bonus) {
      this.bonus = bonus;
  }

  Money totalPay() {
      Money pay = calculateBasePay();

      // Use the strategy
      bonus.applyTo(pay);

      return pay;
  }

  private Money calculateBasePay() {
      // ... code
  }
}
```

This Employee object is now open for behaviour changes in its bonus scheme. But it is closed to modification. We don't need to change the insides just to change a bonus scheme.

Let's demonstrate that with two simple schemes - NoBonus and BonusTwenty

```
class NoBonus implements BonusScheme {
    void apply(Money pay) {
        // No Action
    }
}

class BonusTwenty {
    void apply(Money pay) {
        pay.add(new Money("20.00"));
    }
}
```

When we create employee objects, we inject whichever bonus scheme concrete object we want to use.

Real payment systems are more complex and would have logic in the bonus scheme. That would collaborate with other objects, like Targets perhaps, to see if we qualify. The BonusScheme objects might even get Strategy pattern objects of their own.

Strategy crops up everywhere you want to vary *behaviour*. In the same way a variable handles variable data, the Strategy pattern handles variable behaviour. It's as simple as that.

Strategy can be a low-level mechanism technique. At a higher level, it can express domain ideas that have changing behaviour in the real world. The classic being Employee objects that start out as juniors but then get promoted to managers. Strategy makes for a direct model of that.

Examples: Tax Calculation, Payment schedule, Data source selection, Graphics filters, Plugins of all kinds, Extension points, Customisations, Skins, Complex rules.

Observer

[GoF] Triggers behaviour somewhere else

Observer is a decoupling technique that splits apart a 'thing that happened' from its response.

The classic use of Observer is in GUI toolkits. You have a Button object that can be clicked on. The Button class should be completely reusable, but you need some way for it to trigger your application code. We clearly can't put our application logic inside the reusable Button code. It wouldn't be reusable then.

We need a solution. We need to tell the Button code which part of of our application code we want to run. We can do that with the Observer pattern.

We make the Button 'observable', which means that we can attach a list of observers. These 'observers' will contain our application code.

When we click the Button, this will execute a method on all the observers that we register. We will call the interface 'Actionable', because that's what it provides:

```
interface Actionable {
    void doAction();
}

class Button {
    private final List<Actionable> observers = new ArrayList<>();

    void register(Actionable a) {
        observers.add(a);
    }

    void onClick() {
        observers.forEach(Actionable::doAction);
    }
}
```

That class is part of our GUI toolkit. It has methods not shown here. Typically these will allow a Button to draw itself, and to receive a click notification from the operating system.

When a click happens, method onClick() in Button is called. That calls the doAction method of each registered Actionable observer.

Our application code can then make classes that implement Actionable. These classes are fully decoupled from the Button class itself. The Button lives in the GUI toolkit

and has zero knowledge of your application code. Your application code only needs to know about Actionable and that it can receive a call to the doAction() method. What your code needs to do in response to the click lives there.

Observers are so good at decoupling 'what happens' from 'what caused it' that they crop up everywhere. GUI toolkits, web frameworks, message queue systems, process pipelines.

The Observer pattern is a low level mechanism pattern in general. Obviously, that can 'scale up' to a domain concept if it fits. A PayrollRun object can easily have observers of BankTransfer and PayslipPrinter.

Systems like this often end up as event driven architectures.

Examples: GUI frameworks, Event driven systems, Interrupt handlers, Communication channels - TCP handlers, JMS messaging.

Adapter

[Gof] Adapt an interface for use by something else

We've seen this pattern in Hexagonal Architecture, where it is a perfect fit.

Adapter solves the problem where we have calling code expecting one kind of interface, and a supplier that has another. We need to bridge that gap somehow. The way to do it without changing existing code is to write new code that knows how to talk to both.

We say this new code adapts one interface to another.

The example we used earlier was an interface to find a user:

```
interface UserRepository {
  User findUserById( String userId );
}
```

which is the interface the calling code expects.

We can take any source of user information - a database, a web service, a test stub - and *adapt* it into satisfying that interface.

Suppose we had a small in-house SQL database library. It has class Database that has a query method that returns a Rows object given an SqlQuery object:

```
class Database {
    Rows executeQuery( SqlQuery query ) {
        /// ... code
    }
}
```

We might create an adapter class for our UserRepository like this:

```
class UserRepositoryDatabase implements UserRepository {
    private final Database db ;

    public UserRepositoryDatabase( Database db ) {
        this.db = db;
    }

    User findUserById( String userId ) {
        SqlQuery userQuery =
          new SqlQuery("SELECT * FROM Users WHERE id=" + userId);

        Rows r = db.executeQuery(userQuery);
        User u = new User( userId, r.getColumn("dateOfBirth") );

        return u ;
    }
}
```

Please forgive the SQL Injection vulnerability and the fact it will most likely crash if there is not exactly one user of that userId (ahem).

The gist is that we search our database - using SQL - for the relevant row in our Users table. We pull columns out of that row and use them to create a User domain object. We return that object.

We have not altered any of the calling code in our domain that depends on UserRepository to know anything about the SQL database library. We have not altered any of our in-house SQL database library to know anything about the domain and its User class.

We have written code that *adapts* what our domain model wants into what our existing libraries can provide. We have respected SRP and OCP whilst doing this.

Adapters are really powerful. They are fundamental to Hexagonal Architecture - there is a whole layer of them, after all. Adapters see much wider use.

Any time we have an object that needs something from another, but their methods don't quite line up, we can use an Adapter.

The GoF book lists a related pattern called *Bridge*. It has an interesting section on the similarities and differences to Adapter. I never really saw them as being different. They both make one thing work with another thing that wasn't designed to work with it. I tend to take a simple view of things.

Examples: Hexagonal Adapter layer, Converting data formats, converting presentation formats, linking different versions of published APIs.

Command

[GoF] Gives us a self-contained action

A lot of time we're responding to events in programs. Things happen; a user submits a form. A timer counts down to zero. A loyalty card is scanned. Each one of these events would need some action from our system.

A command object contains the code for that action. It presents a uniform interface to calling code, so that all actions can be called in the same way. This allows us to build libraries and frameworks, where our calling code can pass Command objects into the framework to be run at the right time.

```
interface Command {
    void execute();
}
```

The basis of a Command object is an interface with the single method execute. We tell the concrete class to execute whatever action code it contains.

As Commands are good at handling events, a reasonable example is to see what our GUI Button class would look like with one:

```
class Button {
    private final Command action ;

    public Button( Command action ) {
        this.action = action;
    }

    public void onClick() {
        action.execute();
    }
}
```

Variations include:

- Passing the Button into the execute method to identify the cause
- Maintaining a list of actions, each triggered in order
- Adding an 'undo()' method to the Command interface to reverse the action

Composite

[GoF] Makes many things look like one thing

Composite is a kind of adapter class. It adapts a collection of multiple things into the same interface as just one thing. This makes it a nice pattern you can use in addition to others.

Going back to our Shape.draw() example, the Composite pattern gives us one very useful extra Shape: ShapeGroup:

```
class ShapeGroup implements Shape {
    private final List<Shape> shapes = new ArrayList<>();

    public void add(Shape s) {
        shapes.add(s);
    }

    public void draw() {
        shapes.forEach( Shape::draw );
    }
}
```

This class can be used anywhere that a Shape interface is expected. It is fully swappable for a single Shape. It allows any number of Shape objects to be added using add() then will draw all of them, when its draw() method is called.

Composite objects like this have uses everywhere. The classic use is when some existing code you cannot change expects a single object - like a single Shape above - but you need it to work with multiple objects.

Simply wrap up a collection of objects into a Composite and we're good to go.

Facade

[GoF] Simplifies how we call a complex group of objects

The term Facade comes from buildings architecture. It describes how a 'fake front' is built out of different materials than the rest of the building to improve its appearance.

We use a Facade in OOP when we want a simple interface to our calling code, but the objects needed are harder to work with.

A web service might be made up of a large number of complex objects. A Tax calculation engine might have many objects representing tax bands, allowances, exemptions, thresholds and so on.

Having our application code create each one of these objects, wire them all together and configure them would be quite hard. It would expose a lot of needless detail.

By writing a simple Facade that does all this for us, we can simplify the calling code:

```
class TaxEngine {
    public TaxEngine( Configuration c ) {
        // ... Code to create objects, configure, wire up
    }

    public Money calculateTaxLiability(DateRange taxPeriod) {
        // ... code
    }
}
```

This TaxEngine facade hides all the detail of wiring and configuring the many objects required. It presents a simple interface to the caller.

Builder

[GoF] Makes complex objects easier to construct

When objects get data-heavy as part of the secrets they hide, they grow large constructors. When parts of that data are optional, then the calling code often ends up supplying null and empty parameters. Messy.

One way to improve this is to make multiple different constructors. Each one only lists non-empty parameters, then uses this() to call other constructors. We push the defaults into the constructor in this approach.

This works, but the number of constructors needed gets out of hand quite quickly. Without an IDE to help, it's hard to know which parameter is which. What does `new User ("alan", 3, 198736543, "red")` mean, anyway?

Builder is a pattern to help with this. It provides a number of named methods which describe the purpose of each piece of data supplied. It contains the default and empty values to be used. And it only requires a single, 'full' constructor for the object being built.

We might have three optional fields in our User class :

```
class User {
    private final String userId ;
    private final Date dateOfBirth ;
    private final User linkedAccount ;

    public User( String userId, Date dateOfBirth, User linkedAccount) {
        this.userId = userId;
        this.dateOfBirth = dateOfBirth;
        this.linkedAccount = linkedAccount;
    }
}
```

We could create a User with default dateOfBirth and linkedAccount by the calling code:

```
User u = new User( "alan0127", new Date(), User.NOBODY );
```

Where User.NOBODY is a public static constant null object signifying ‘no user’.

This works reasonably well for three parameters, but it does obscure which ones are the genuine parameters and which are defaults. As parameter lists grow, the situation gets worse.

A UserBuilder would look like this:

```
class UserBuilder {
    private String userId = "";
    private Date dateOfBirth = new Date();
    private User linkedAccount = new NullUser() ;

    public User build() {
        return new User( userId, dateOfBirth, linkedAccount );
    }

    public void userId( String userId ) {
        this.userId = userId ;
    }
```

```
13
14      public void dateOfBirth( Date dateOfBirth ) {
15          this.dateOfBirth = dateOfBirth;
16      }
17
18      public void linkedAccount( User linkedAccount ) {
19          this.linkedAccount = linkedAccount ;
20      }
21  }
```

You can see that the default values are built into the Builder. We can change them with the 'setter' methods provided. Once we have set all we need, we create a User object using build().

I prefer to put the build method at the top, just as a matter of style. I think it is the 'most important' method and the one that helps you understand the class.

A Java library exists to automate writing builders like this, which is useful:

Project Lombok at https://projectlombok.org/features/Builder

> It's worth noting that large parameter lists are often a code smell in OOP code. It's usually a sign that the object should be broken down further. Do consider that before creating a Builder.

Repository

[DDD] Represents stored objects, without saying how they are stored

Another foundational pattern. We have some data in some kind of storage that we want to work with. But we don't want our application to know anything about that storage technology. We do want to get that data back out and use it to create objects.

Repository allows us to create an abstraction of stored data, in our domain terms. We've seen this earlier. We've used it to store User objects as an example of the

Adapter pattern. Repository focusses on the interface that is called, rather than the adapter class implementation.

To show how useful Repository is, let's add a few common methods:

```
interface UserRepository {
    User  findUserById( String userId );
    Users findUsersWhoseNameStartWith( String startOfName );
    Users listAllUsers();

    Users execute( Query q );
}
```

Users is an aggregate class, containing methods relating to multiple User objects.

This interface uses concepts known in our domain. It says nothing about the storage mechanism. This makes it easy to read, easy to stub for testing and easy to swap storage technologies.

Our domain model code can do all it needs to do using only this interface. It gives total separation of concerns.

The methods listed above are typical:

- find a single User by various unique things, like a userId
- find multiple users matching some criteria
- return everything

The final method is really interesting. It returns all User objects matching a specified Query, passed in as a parameter object.

Query is itself an abstraction. It is not an SQL 'SELECT' query. It is not a Mongo query. It is a group of pure Java objects that represent a query. You would see things like:

```
Query q =
  new QueryBuilder()
  .attributeMatches( "name", "Al*")
  .and()
  .timeAttributeBefore( "creationPoint", new Time("2020-01-05T11.59.00Z\
") )
  .build();
```

Which would build an Abstract Syntax Tree (AST) describing a query that matches all Users whose name begins with “Al” and whose creationPoint timestamp was before 11:59 on Nov 5th 2020.

The Query object has a translate(QueryDialect dialect) method on it. This gets passed a concrete object that knows how to generate a specific query for a specific technology - like MySqlQueryDialect. Often, this is done using the Visitor pattern. It is probably the canonical example of why you would use Visitor.

This maps the generic query to be mapped into something useful to an implementation. It also means the domain code query above does not change when the storage technology changes.

Examples: Relational data access, NoSQL data access, REST data store clients, caches

Query

[PoEAA] Describes what to search for, but not how it will be done

Our applications need to be able to find objects once they are created.

In memory, once we create an object using new, we receive a reference to that object. We keep hold of that reference and use it to access the object.

For objects we stored in the Repository earlier, we don’t have this reference. To find our object(s) again, we need a Query. Some way of specifying ‘look for the object(s) best matching these search terms.

Queries take all forms. We’ve seen one frequently in out findUserById(userId) method. Here, the search criteria “by user ID” is baked into the method name.

That's readable, simple and testable, But it isn't very flexible.

Often, we need a general-purpose query system. Just like the kind we find in an SQL statement.

But here's the catch: we don't want to express it as SQL.

We've used a Repository object. We've used Hexagonal Architecture so that our domain layer knows nothing about how that Repository will be implemented.

If we were to pass SQL statements into our finder methods, that defeats the whole purpose of that. We have created a *leaky abstraction*, one where parts of one specific implementation 'leak through'.

The solution is to split a query into two parts:

- Pure English: An abstract representation of search criteria
- Specific Details: A translator turns that into some specific technology

It's easier to understand once we realise all queries can be written in English. "Show me all the users who joined yesterday" is an example. That's the role of the Query object pattern. The Query object expresses things in pure domain terms. A translator object then converts this into whatever form is needed.

Simple Query Object

It's often useful to have a very simple kind of Query object. It is intentionally limited in the queries it can represent, making it simpler to write the translator object for. The trick is to make it good enough to do the job.

```
class Query {
    private String name ;
    private Object value ;

    public void attributeEquals(String name, Object value) {
        this.name = name ;
        this.value = value ;
    }

    public void translate(Translator t) {
        t.generate(name, value);
    }
}

interface Translator {
    void generate(String name, Object value);
}
```

This bare-bones Query object shows the basic split of work. We can represent just the one query - 'is <attribute> equal to <value>?'. We can call translate() passing in some Translator object to generate an implementation-specific query.

Here is a simple SqlTranslator object that will create a SELECT statement for us, then display it - just for this example. It would normally link up to a Database object. This would manage a JDBC connection, run the SQL, then return a User object.

```
class SqlTranslator implements Translator {
    private final String table ;
    private String sql ;

    public SqlTranslator( String table ) {
        this.table = table ;
    }

    public void generate( String name, Object value ) {
        // Note: this only works for text values
```

```
        sql = String.format("SELECT * FROM %s WHERE %s = '%s'",
                                table, name, value);
    }

    public void displayForDebug() {
        System.out.println( sql );
    }
}
```

We use these classes to implement our findUserById() method:

```
User findUserById( String userId ) {
    Query q = new Query();
    q.attributeEquals( "id", userId );

    Translator sql = new SqlTranslator();
    q.translate(sql);

    sql.displayForDebug();

    // real database code omitted
}
```

You can see how the details of SQL are hidden in the SqlTranslator object. The details of what we want to search for are hidden in the Query object itself.

Significantly, they are expressed in domain terms - not SQL or database terms. We could write a MongoTranslator or FileTranslator and make those work.

You can also see the comment: it only works for String values.

More complete Query objects need to convert between Java data types and whatever type the storage technology needs.

In fact, Query objects and translators get complex, fast.

The full-fat version builds an Abstract Syntax Tree of Criteria objects, complete with operators such as AND, OR, LIKE, NOT, EQUALS, BETWEEN. They will convert

between all the data types and use the Visitor pattern to translate the Criteria graph into the implementation version.

I have actually built one of those for a real project. It was a system where a user could build their own queries to report on a fleet of printers in an enterprise network. The actual implementation was intended to use a number of backend database systems to store a snapshot of that fleet's status.

The best reference on complex Query objects is Patterns of Enterprise Application Architecture by Martin Fowler [PoEAA]. Remember you can start simple and still get the benefits for many applications.

CollectingParameter

[C2WIKI] An object that collects results

How do we use tell-don't-ask style coding with an aggregate object when we want to produce some kind of summary or total result?

A Collecting Parameter is an object we can pass into a method. The same collecting parameter will be passed to a number of different objects, so that each object can add its contribution:

```
1  class UserMetrics {
2      void ageLessThan25();
3      void age25To35()
4      void age35Up();
5
6      void premiumSubscriber();
7      void inactive90days();
8
9      void display( Display d );
10 }
```

UserMetrics is a collecting parameter class that can be passed to multiple User objects. The User object would call the relevant ageXXX() method to collect age demographics, adding one to the relevant total number of users in that cohort.

It would also call premiumSubscriber() and inactive90days() as appropriate. These are sales pipeline targets that we have decided we want to alter, so we need a measurement.

We would pass it into Users like so:

```
class Users {
    private final users List<User> users = new ArrayList<>();

    public void add( User u ) {
        users.add( u );
    }

    public void collect(UserMetrics metrics) {
        users.forEach(user->user.reportMetrics(metrics));
    }
}
```

We would call Users.collect(), pass in a UserMetrics object instance and it would be populated. After that, we can display those metrics.

All sorts of uses are made of this. Totals, reports, summaries, Bills of Materials, Inventory. It's a very useful technique. It is a simplified version of the 'Visitor' pattern.

Examples: Management Reports, Upgrade items, Orders, Totals, general Data Processing.

Item-Item Description

[Coad] Keeps a Thing and its description separate

This useful pattern crops up in e-commerce a lot.

You have a Thing - product, service, offering, movie, song - whatever. The Thing needs a description so that it can go in the catalogue.

The Thing is logically different than its description. You can talk about how many copies of 'The Andromeda Strain' movie you have to buy. You can also talk about

who directed that film and who wrote it. The two are different views, so it makes sense to separate them out.

```
class MovieDescription {
    void displayTitle(Display d){...}
    void displaySynopsis(Display d){...}
    void chargePriceToAccount(Account a){...}
}

class Movie {
    private final MovieDescription description ;

    public Movie(MovieDescription description) {
        this.description = description;
    }

    void buy(Account a){...}
}
```

Examples: Catalogues, Media Collections, Products, Legal proformas, e-commerce, Classified Ads, Discount Vouchers

Moment-Interval

[Coad] Captures a period of time, beginning to end

Moment-Interval represents a period of time defined by a start time and an end time. Behaviours can either be general-purpose - like boolean isAfter(time), boolean wasBefore(time) - or application specific.

Time ranges are important in many domains. Rental agreements, leases, validity checks, access control windows are all subject to time limits. Moment-Interval gives us a clean way to model them.

```
class SubscriptionPeriod {
    private final Date start ;
    private final Date end ;

    public SubscriptionPeriod( Date start, Date end ) {
        this.start = start;
        this.end = end;
    }

    public void applyTo( Subscription s, Date now ) {
        if ( now.isAfter(start) && end.isAfter(now) ) {
            s.subscriptionActive();
        }
        else {
            s.subscriptionExpired();
        }
    }
}

class Subscription {
    private SubscriptionPeriod period ;

    public void renew(Period p) {
        this.period = p ;
    }

    public void provideService( Date now ) {
        period.applyTo( this );
    }

    public void subscriptionExpired() {
        // action when expired
    }

    public void subscriptionActive() {
        // action when active
```

```
37        }
38    }
```

Software-as-a-service (SaaS) businesses rent out access to their offering for a fixed period of time. The period you can use them ranges from the moment your payment is received until some point in the future, often based on how much you pay.

The Subscription above would represent a time-bounded service. Calling method provideService() with the current time would delegate to the Period object.

This is our Moment-Interval pattern object.

This would check the time now against its limits. If it is within those limits, it will call back on the Subscription active, triggering the 'subscriptionActive()' method. This would provide the paid-for service.

If the subscription has lapsed, then Period will call back to 'subscriptionExpired()', which can then attempt to get the user to renew their subscription. If they do, Subscription method renew() would be called, with an updated Period.

Clock

[Own] Provides a way to represent current time that can be stubbed

Time driven functions need to know the actual real-world time now. Writing this, it is the 11 Nov 2020, 22:54:23 BST.

All usable systems provide a way to get real-world time. Java provides the older `new Date()` syntax to access the system clock. The Java 8 Joda-time inspired update provides newer features and a less zany syntax.

But both systems share the same problem. If you use them directly in your code, you are stuck with the actual time, right now, in the real world. This makes testing either hard or impossible. Recreating a production fault from logs captured earlier is impossible, too.

In both cases, we need a way to force the time to a test value.

The Clock pattern is a simple abstraction of the system clock. Using the older Date syntax for simplicity, it looks like this:

```
interface Clock {
    Date now();
}
```

This is a Dependency Inversion (DIP). Our code now depends on only this interface for its source of the current time, using the now() method.

For production, we define a SystemClock class:

```
class SystemClock implements Clock {
    public Date now() {
        return new Date();
    }
}
```

We Dependency Inject this SystemClock class to everywhere that needs to know the time. Often, this comes from a Config class that runs at application startup.

For testing, we inject a stub class:

```
class StubClock implements Clock {
    private Date date ;

    public Date now() {
        return date ;
    }

    public void setTo( Date d ) {
        this.date = d;
    }

    public void oneHourLater() {
        // code to go forward one hour
    }

    public void oneHourEarlier() {
        // code to go back one hour
```

```
18      }
19  }
```

This stub has features to set to a specific time, then move to an hour later or earlier. You would change these to your specific test requirements. The idea is that you are creating a Domain Specific Language (DSL) about test times, to make your test code into readable documentation.

This is one of those insanely useful patterns that it is actually muscle memory for me.

Rules (or Policy)

[Own] Captures a set of rules to apply in a single place

Rules are part of pretty much every business process you want to automate. Things like applying for a mortgage, verifying a user account, or checking a message for being spam will all have a ruleset to apply.

In many cases, these rules get coded ad-hoc somewhere in the system.

It is usually good for clarity if we pull out all the rules into a single object. This also gives us the possibility of treating the rules object a Strategy pattern. That gives us pluggable rules that can change.

An example is in calculating Tax from a given amount using a banded tax rate system.

Say you pay zero below 10,000. Then 20% between 10,000 and 30,000. Then 40% above that. Don't take the numbers too seriously; I'm hoping I'm a better developer than a Chancellor of the Exchequer.

Here is one way of doing this using the Rulebook pattern:

```
class TaxBand {
    private final Money lowerLimit ;
    private final Money upperLimit ;
    private final BigDecimal percentage ;

    public TaxBand(Money lowerLimit,
                   Money upperLimit,
                   BigDecimal percentage) {
        this.lowerLimit = lowerLimit ;
        this.upperLimit = upperLimit ;
        this.percentage = percentage ;
    }

    public void apply( Money amount, Money totalTax ) {
        if ( amount.atOrAbove(threshold) ) {
            totalTax.add( calculateTax(amount) );
        }
    }

    private Money calculateTax( Money amount ) {
        Money taxableAmount = amount.subtract( lowerLimit );
        return taxableAmount.multiply( percentage );
    }
}
```

This is a Rules object. You call apply() with the amount to calculate tax on and pass in a CollectingParameter to collect the total tax.

Each TaxBand object is constructed with the secrets it needs to apply that specific rule.

In this design, TaxBand objects are chainable. We could either add a method chain(TaxBand next) - internally forming a linked list of Tax Bands - or create an Aggregate object called TaxBands. Both read well in this case, so take your pick. I have a slight preference for the Aggregate approach. It separates the concerns of 'one' and 'many' more fully.

In many systems, rules are configurable. A rule to limit screen time might be

configured with a maximum duration. This is often done as a simple variable set by configuration.

Rules objects are really useful when the *behaviour* itself has to be configured, rather than just some threshold value.

Examples: Access Control, retention policies, business process rules

Aggregate

[DDD] Represents a group of objects and the behaviours that apply to all

Aggregate objects were covered in an earlier chapter. Class User represents one user. Class Users represents a group of them.

They wrap a collection of single objects. Behaviours are the ones common to group operations. Methods operate on all objects in the collection.

Cache

Speeds access to an object that is slow or expensive to fetch

Many systems end up fetching data from somewhere else, often a database or a web service. These things live at the other end of a network connection. That makes fetching data five or six orders of magnitude slower than it would be if it was inside a local object.

We can't simply avoid calling that remote data. That's where the data actually lives, like it or not.

What we can do is make that remote call as infrequently as possible. We do this by keeping an up-to-date copy of it as a local object.

Cache is the name we give to this idea. A simple cache is a map of objects most recently used:

```
interface UserRepository {
    User findUserById( String userId );
}

class UserCache implements UserRepository {
    private final Map<String, User> cache = new HashMap<>();
    private final UserRepository repository ;

    public UserCache( UserRepository r ) {
        this.repository = r
    }

    public User findUserById( String userId ) {
        if ( cache.containsKey( userId ) ) {
            return cache.get( userId );
        }

        User user = repository.findByUserId( userId );
        cache.put( userId, user );
        return user;
    }
}
```

This is actually a 'buy one get two free' of design patterns: a Repository, Decorator and a Cache. We even have some LSP thrown in. Let's explain those one at a time to see why they are a perfect match here.

The Cache part is the HashMap called cache.

This is used to store users in a simple lookup table, against their userId. Given a userId, we can check the cache variable to see if we already have the relevant User object stored in there. The first three lines of method findUserById() do that. If the User exists in our cache, we return that.

This is the key benefit of the cache - we return straight away with a local object. We avoid that slow round trip to the UserRepository.

If the User is not present in the cache, that must mean we haven't asked for that

userId before. So, we go off to the UserRepository. We fetch the User object, store it into the cache, then return it. The next time we ask for this userId, it will be returned from the cache.

Caches are very commonly used to speed up slow operations. Repositories and External Systems typically benefit from a Cache.

That said, they are difficult to design right.

Our simple example has two serious problems:

- **No updates** We never fetch updates to a cached User
- **No eviction** Users are cached forever. Memory will run out

These two issues are part of *cache invalidation*, the act of marking a cached item as no longer valid.

Real caches get increasingly complex to handle this in production.

You often see an associated Rules pattern object to describe when to clear out the cache. Popular choices are Least Recently Used (LRU) which requires a timestamp being stored alongside the cached object and the simpler Most Recent cache, where we only make space to cache the last N items.

They say there are four hard things in computing: Concurrency, Cache Invalidation and off-by-one errors …

As for the other two patterns, we are using Repository to abstract away the details of how we store User objects. We've covered this before.

We are also using the Decorator pattern, first mentioned in the [GoF] Gang of Four Design Patterns book. That's a useful pattern worth breaking out into its own section. Let's do that next.

Decorator

[GoF] Add extra features to an object in a substitutable way

Decorator gives us a way to add features to a class without altering that class. It is a wrapper for that class, similar to an Adapter. It differs from Adapter because the new class can be substituted for the original one it wraps. It is fully LSP compliant.

This means that calling code can be fed either the original class or the decorated one and not have to change.

In the explanation of the Cache pattern, we used a Decorator around a Repository object for this very reason:

```
interface UserRepository {
    User findUserById( String userId );
}

class UserCache implements UserRepository {
    private final UserRepository repository ;

    public UserCache( UserRepository r ) {
        this.repository = r
    }

    public User findUserById(String userId) {
        // Extra functionality goes here

        User user = repository.findByUserId( userId );

        // Extra functionality goes here
        return user;
    }
}
```

This is the same code example but with just the parts relating to Decorator left in. The functions relating to the Cache have been removed as they are not part of the Decorator pattern itself. That's much simpler.

The tell-tale signs of this being a Decorator are the constructor which takes a UserRepository object, and the class which implements the UserRepository.

It is both substitutable for a UserRepository and yet has access to one itself - always a different object instance. Here, you could imagine we have MySqlUserRepository as an instance to pass into UserCache. The end repository is itself substitutable.

This object allows us to decorate our UserRepository to give it extra functionality. Here, it is to work with a hash map to locally cache the User objects. Only if we do not have them locally do we go further out to our 'real' UserRepository.

Other common uses for Decorator are to implement access control to the decorated object, or logging. Logging is critical in production systems. It is the only way you can monitor that your system is working - and diagnose what happened when it falls over at two in the morning.

Another use is when a framework hands you one of its own objects, and you need extra application-specific features. You cannot change the framework code, so wrapping it with a Decorator can often help.

Examples: Cache, Access Control, Production Logging, Extending Frameworks

External System (Proxy)

[GoF] Represents our view of an external system locally

External Systems are a challenge for our applications. They always need some kind of raw data transfer between them, somehow mapping to our objects. Because these systems live outside our memory space, communication will be much slower. It can often be 100,000 times slower than calling a method locally.

Dealing with these challenges is simpler when we introduce a local object to represent that external system.

The best way to deal with an external system is to

- Create an interface to represent it
- Use only our domain terms to abstract the details away

We covered this fully in the chapter on Hexagonal Architecture. This is the pattern behind it.

An External System object is powerful because it decouples us from the details of that system. This gives us several possibilities. We can use a stub for testing. We can create simple Fakes to help us split up development across our team. We can wrap the system with things like a Cache, or logging or monitoring.

> Fake: a stub used in development whilst we build the real thing

Specifically, an external system object represents *only what we need* of that system. Google Maps, as one example, has a large number of useful features. But if all we needed was the ability to get the name of the nearest town to a point, then our interface will be:

```
1  interface Geography {
2      String getNameOfTownNearestTo( float lat, float long );
3  }
```

Of course, we would create a GoogleMapsGeography object that implements this interface and actually connects to Google Maps API over HTTP.

This is part of a larger, very useful design idea known as *consumer driven contracts*.

The idea is that all that matters is what a specific client needs. You can use this to drive out required behaviours of servers, outside-in, based on what is actually needed - not what 'might be nice to have'.

> Astute readers (which is, of course, all of you lovely lot) will notice that this is exactly what we have been doing this whole book with OOP. Just swap 'object' for 'server' and we're done: Consumer Driven Contracts.
>
> I do tend to view a lot of software engineering like this. It's a handful of core concepts that crop up all over the place. BDD, Tell Don't Ask, Consumer Driven Contracts - it's all the same idea. The jargon sometimes makes it sound harder than it really is.

The original pattern I saw was Proxy in [GoF]. That was a more general kind of object. It had the same interface as some other object and stood in place of it. In my work, Proxy's overwhelming use has been to represent an external system, so I prefer to think of it as above.

Configuration

[Own] Gives a way to change how the system is set up

Once an Object Oriented program starts doing useful work, it will need various things setting up, both at start up and throughout its runtime.

At start up, it will need all its objects creating as instances of classes. All the various dependencies need injecting. URLs of external web services need to be provided. The connection string and address of our databases need providing.

These kinds of things are often specific to the environment we run in. The database string will be different when we are using the test database as to when we run in production. The web service URL might point to a fake server made with WireMock, rather than the real service on the web.

Even the *kinds* of object may change. We might have a setting in a file to use either an Oracle database or Microsoft SQL Server. A different object might need to be created in each case, then injecting into all its callers.

We call this Configuration. It includes setting data values like URLs, creating specific objects and 'object wiring' - inject dependencies between objects.

The best place to put all this is in a Configuration class:

```
class Configuration {
    private boolean useOracle;
    private String jdbcConnectionString;
    private boolean useDarkMode;
    private Date offerExpiryDate ;

    public void load() {
        // Code to read a configuration file, or server
    }

    public UserRepository userRepository() {
        if ( useOracle ) {
            return new OracleUserRepository(jdbcConnectionString);
        }
```

```
15
16          log(WARN, "Using STUB User Repository - TEST/QA ONLY");
17          return new StubUserRepository();
18      }
19
20      public UiStyleScheme styleScheme() {
21          return useDarkMode?
22              new UiStyleDark() :
23              new UiStyleLight() ;
24      }
25
26      public Date getOfferExpiryDate() {
27          return offerEXpiryDate ;
28      }
29  }
```

Gives the basic idea. Configuration is held *somewhere* then read in. Based on this data, the right kinds of objects are created for use elsewhere. The rest of the code refers to this Configuration object to get its dependencies and values.

The actual configuration data is stored typically a 'config.json' kind of file, or perhaps a database at a fixed location, or a web service. Some providers like Spring Cloud use network broadcasts to discover the configuration server at run time.

Once read in, the methods on the Configuration object can be used to create the specific objects needed. They can be initialised with the required values of things like URLs, timeouts and connection strings.

Often, whole objects can be created here. That leads to nice clean code. But I have added a getter to return a pure data value. Configuration classes are one place where that can often be a good idea, resulting in the clearest code.

As the application grows, it makes sense to split into multiple Configuration classes.

These can align with your Hexagonal Architecture so that each subsystem gets its own Configuration object. You will probably end up having a Configuration interface in your domain layer, then writing a specific Adapter for it to link it up to your actual source.

I've also shown a common idea where we normally use the Oracle database in production, but we can enable a test stub in our configuration file.

This simplifies development across a team. The QAs can do their manual exploratory testing of new UI changes without needing a database connection. The other devs who are not working on the Oracle Repository can get on with their work.

This particular piece of design is as much about team leadership and management as it is about clean code.

Order-OrderLineItem

[Coad] Gives structure to a list of items on an order

Orders crop up everywhere in commercial systems. You'll routinely see point of sale receipts, invoices, bills of materials and e-commerce customer orders.

Each one has the same basic pattern.

There is a line item describing the single thing that has been ordered. Typically, it knows about some combination of values: quantity, part code, description, price, tax, location. This, plus suitable behaviours, go into the OrderLineItem object.

As orders often have more than one line item, we aggregate these into an Order object, describing the total order. That often knows about the time it was placed, delivery and billing details, related accounts. Typical behaviours on Order are display, takePayment and place, which moves the order onto fulfilment.

Order-OrderLineItem is simply a specific use case for the more general Aggregate object idea. But it's so common it's worth pointing out.

I first saw it in the excellent [Coad] book, which lists a toolbox of common business processes and patterns that model them.

Examples: Point of Sale receipts, invoices, bills of materials

Request-Service-Response

[Own] Describes a flow to get a response from a service

This crops up a lot in Hexagonal Architecture designs. It is the 'inner hexagon' domain model of a web request - without any reference to web technologies. As such, you can use it with any kind of input and output adapters. It represents a general request to a system.

To carry on our running theme, let's show how this can be used to find a user by their id.

A web request will have come in, let's say as RESTful format as GET /users/alan0127. We would write an Adapter to extract the userId of alan0127 from the path parameter, then store that in a request object:

```
class FindUserByIdRequest {
    private final String userId ;

    public FindUserByIdRequest( String userId ) {
        this.userId = userId;
    }
}
```

So far, this hides away the userId we extracted. Next up, we need to find this user from our repository. We can do this because we're entirely inside the inner hexagon now - the pure domain model. Our UserRepository lives there:

```
interface UserRepository {
    User findUserById( String userId );
}

class FindUserByIdRequest {
    private final String userId ;

    public FindUserByIdRequest( String userId ) {
        this.userId = userId;
    }

    public User execute( UserRepository users ) {
        return users.findUserById( userId );
```

```
14      }
15  }
```

We add the execute() method which is supplied a UserRepository implementation of some kind. That will be a stub for unit testing and something real like PostgresUserRepository for production. The execute takes the userId we stored earlier, runs the findUserById() query and returns the User object.

Our web layer Adapter class will then take the User object and format it into a web response.

In the dim and distant days of old, this would be an HTML response. But the last few years have seen server side HTML generation replaced by JavaScript in the browser. We'll probably return a lump of JSON formatted data to a React (or Angular, or Vue, or vanilla JS…) application that formats it for display.

This is a straightforward pattern that does what it says on the tin. Its value is that it separates out the various technologies from the business logic. Using Liskov LSP substitutable objects for our interfaces makes this a mix-and-match system. The same domain model that drives a web site can just as easily power a desktop GUI written in JavaFX as it can a console app.

Anti-Patterns

Design patterns are powerful, but they are easily overused. There are a couple of anti-patterns to avoid.

Design Pattern Soup

Don't create a 'kitchen sink' design.

Whenever you see you have written `class StrategyFactoryFactoryBuilder`, as my friend Jenny would say, you need to 'Go into the corner and have a word with yourself'.

There might be a reason you ended up down this rabbit hole. But try to re-design your way out of it, if you can. If that's not possible, rename this monster. Go back to basics: in terms of *the application*, what does this beast gives us? Why is it here?

Unneeded Flexibility

Don't over-design objects.

You might add a pattern 'just in case for future extension'. Perhaps our simple User object could sprout an Observer pattern, allowing future code to listen for changes to the user name.

Let's just take that out, shall we?

Leave the future to itself. Our crystal balls aren't good enough. We can't know what code will be best in the future.

Even if the requirement lands, we often find that when it does, things have moved on. The assumptions we made in our original guess at future code were wrong. The code is now cruft that we have to write a workaround for.

Just write the code for today. Let tomorrow take care of itself. We will refactor as required.

Mechanism Madness

Don't over-emphasise how something works.

We're going back to basics here. Good design highlights what something gives you, rather than what it is made from.

Rather than copy-paste the patterns directly, blend them in.

If our User object *did* need an Observer pattern for name changes, let's not call it that. Let's create an interface called NameChanged rather than Observer.

Patterns are meant to be used blended in like this. Otherwise, they would simply be a library of fixed code. Then they would lose their value.

OOP Mistakes - OOP oops!

So far, we've covered ways to use OOP well.

We've seen a lot of techniques that make code readable, easy to test and compact. Remember how easy it was to combine Repository with Cache with Decorator and get a pluggable speed boost?

It turns out that not everybody agrees that OOP is a good thing.

In part, of course, they are right. Plenty of things just are not suited to OOP. I can think of simple batch scripts, build scripts and Terraform scripts that set up infrastructure. We also have 'Serverless Designs', which use many small functions to do their work. OOP often gets in the way there.

Another driver of non-OOP designs has been processing power limitations.

For decades, CPUs followed Moore's Law. Computing power doubled every two years. But then it hit a physical limit. CPUs had as many transistors on them as could be fitted. "I cannae change the laws o' physics!" said Scottie in Star Trek; So it is with photolithography. A transistor can be made only so small, but no smaller. Then physics fights back.

To get around this, CPUs created multiple cores - multiple copies of a CPU design on the same chip.

This impacted software design. Suddenly, we needed parallel processing rather than sequential. The kinds of state that lived inside one object of OOP was not accessible to other CPU cores. OOP became less useful as a model of communicating state across CPU cores. Functional Programming - "stateless" programming - is useful here.

However, none of that was what gave OOP a bad name. What gave it a bad name was OOP *done wrong*.

So, what are the common mistakes?

Broken Encapsulation - Getters Galore!

At Number 1 in my chart of badness, a very common error:

```
class User {
    private String name ;

    // This is painful to type ...
    public String getName() {
        return name ;
    }

    // ... so is this
    public void setName( String name ) {
        this.name = name;
    }
}

class UserGreeter {

    // I swear fairies and kittens are dying right now
    public void greet( User u ) {
        System.out.prinltn( u.getName() );
    }
}
```

We've all seen this - getters and setters everywhere!

It makes me sad. It really does.

Now, to be fair, Java has to take a lot of blame for this. Right from version 1.0, Java had this idea of "Java Beans" which were things like User above.

You had fields, but every field had a getter and setter. Somehow, that became A Thing (TM), and it was used everywhere. Then *taught* everywhere. Then somebody decided this was an "object" - because it is in a class and has private data and public methods.

This caught on with developers who had understood procedural programming but hadn't yet learned OO. They hadn't learned about objects exposing behaviour and hiding secrets.

When you think in that way, every problem looks like getters-and-setters. Object secrets are made un-secret. Code that should be a method on the User object now appears in some redundant "class" UserGreeter. It's not really a class, because it doesn't really have any real secret. It has stolen a secret from User.

It's just procedural programming, plain and simple.

Procedural designs simply do not gain the benefits of OOP. But they do have the words 'class' and 'private' in them. To the uninformed, they look like an OO design. But they are not.

When people criticise OOP for not delivering on its promises - but base it on code like this - it is obvious where the fault lies: This one is on them.

If you don't even realise your code is not OOP, then don't criticise OOP for your code!

Broken Inheritance

The darling of the early 1990s, inheritance was going to be magic! You whacked a few classes together with inheritance and Boom! Your code was done.

But it wasn't done. Instead, it was just a mess.

There were several failings with inheritance. They are the ones that appear in "OOP is dead" blog posts these days.

Let's learn from the classics of that particular genre.

Bird extends Animal

I roll my eyes anytime I read 'class Animal' in an example, in exactly the same way as I do when I see a dead end in a horror movie.

Here's how the problem unfolds. We start with an Animal base class with methods eat() and makeNoise(). We add a subclass of Dog. Then we add a subclass of Bird adding a fly() method. Then two subclasses of Bird, traditionally Sparrow and Dodo. The Dodo cannot fly, so we maybe throw an exception or do nothing in Dodo.fly().

The whole mess looks like this:

```
abstract class Animal {
    abstract void eat();
    abstract void makeNoise();
}

class Dog extends Animal {
    void eat() {
        System.out.println("Meat");
    }

    void makeNoise() {
        System.out.println("Woof");
    }
}

abstract class Bird extends Animal {
    abstract void fly();
}

class Sparrow extends Bird {
    void eat() {
        System.out.println("Seeds");
    }

    void makeNoise() {
        System.out.println("Cheep cheep");
    }

    void fly() {
        System.out.println("Flap flap");
```

```
31      }
32  }
33
34
35  class Dodo extends Bird {
36      void eat() {
37          System.out.println("Nothing, extinct");
38      }
39
40      void makeNoise() {
41          System.out.println("No sound, extinct");
42      }
43
44      void fly() {
45          throw new UnsupportedOperationException("error: cannot fly");
46      }
47  }
```

There are a couple of things wrong with this, which the calling code would reveal quickly. Oddly enough, the "OO is dead" examples never include unit tests or calling code, do they? Funny, that ...

The key issue is the broken contract of Bird.fly() when we inherit that method in Dodo.

A Dodo cannot fly, so the expected contract "make it fly" cannot be met.

That's a misuse of inheritance - technically a misuse of a static type to incorrectly model a behaviour that does not exist. It fails LSP. We know we should not use inheritance when LSP is going to be broken. It is not a good fit.

That's a problem, yet the bigger point is usually missed entirely in these articles. Why does Bird even derive from Animal? They don't have many use cases in common.

Recall how we do this right. We start outside-in and design behaviours we want our objects to have. If we are going to treat our objects polymorphically, then we make sure that all the objects can respond to the same methods.

How about in this case? No way is it going to work like that.

Think about what the calling code needs for polymorphism. It wants some Base classes to use. If it uses Animal, then Bird does not exist - the calling code will never see Bird, just Animal. You cannot call fly() on an Animal. If it uses Bird, then Animal does not *need* to exist. Bird simply has three methods eat(), makeNoise() and fly(). You could simply put those into a Bird interface, and the calling code has no clue whether an Animal is involved or not.

Neither Animal nor Bird makes a good base class for this. Yet here we are, proudly showing an inheritance tree of useless base classes, in an article saying OOP doesn't work. Good job. Loving your work …

The fundamental problem missed is that you cannot call any of this in a Tell-Don't-Ask polymorphic style. You have to know whether you have only an Animal, or a Bird, or a Dodo - and then remember never to call fly().

The entire advantage of polymorphism in OOP has been missed

Inheritance is only useful when classes are LSP substitutable in calling code. If that calling code has to grub about with instanceof to work out what class it is dealing with, you haven't done OOP. It's no better than using a bunch of if statements and a big procedure.

The usual hack is to store things as Animals and use instanceof to work out if they are the different abstraction of Bird. Ugh. No thanks.

So, how might we sort this mess out?

We simply accept that we jumped the wrong way with this abstraction.

Back to basics time. The object behaviours that we are trying to model are eat(), makeNoise() and fly(). Different kinds of Animal have these behaviours in different combinations.

Lets start by creating interfaces for each behaviour.

```
interface Flying {
    void fly();
}

interface Feedable {
    void eat();
}

interface Audible {
    void makeNoise();
}
```

As a first stab, we can straighten out some of our LSP failures:

```
class Dodo implements Feedable, Audible {
    void eat() {...}
    void makeNoise() {...}
}
```

We've fixed the kludgy exception for Dodo.fly() by not inheriting the behaviour *that Dodo does not have* in the first place. Avoiding problems: always smart.

We can do the same thing with Dog and Sparrow. Note that Animal and Bird simply vanish in this scheme, *because they were the wrong abstraction.*

This improves our calling code somewhat. If we pass Dodo to a method that wants a Feedable thing, it will work. It is LSP substitutable for that. Dodo will never bomb out on fly() because it does not have that method in the first place.

However, our animals are still not very polymorphic. This might be fine. It means the calling code needs to know which interfaces it needs.

How could we make a fully polymorphic Animal, that we could tell-don't-ask to do the right thing?

```
interface Behaviour {
    void do();
}

class Animal {
    private final List<Behaviour> behaviours = new ArrayList<>();

    public void add(Behaviour b) {
        behaviours.add(b);
    }

    public void behave() {
        behaviours.forEach(Behaviour::do);
    }
}
```

We move up a level of abstraction.

You might recognise interface Behaviour from the section on Design Patterns. Behaviour is a Command pattern. It provides a method do() which will cause our animal to do its thing.

Animal now represents a single kind of animal that can have any number of behaviours. In a language like Java, an interface or class can only describe a *fixed* set of behaviours. The classic bad example tries to use it to model a *dynamic* set of behaviours and fails miserably. We've fixed the root cause of the problem with our new Animal class.

We can make up Behaviour classes to represent our animal behaviours:

```
class AnimalNoise implements Behaviour {
    private String noise ;

    public AnimalNoise(String noise) {
        this.noise = noise;
    }

    void do() {
        System.out.println(noise);
    }
}

class Flight implements Behaviour {
    void do() {
        System.out.println("flap, flap");
    }
}

class Eating implements Behaviour {
    void do() {
        System.out.println("nom nom");
    }
}
```

Then we configure each Animal with the behaviours we want - using composition, not inheritance:

```
Animal dodo = new Animal();
dodo.add( new AnimalNoise("Dodo noise - brrrraaaak!") );
dodo.add( new Eating() );

Animal sparrow = new Animal();
sparrow.add( new AnimalNoise("cheep cheep") );
sparrow.add( new Eating() );
sparrow.add( new Flight() );
```

We can make an aggregate object to hold all our animals:

```
class Animals {
    private List<Animal> animals = new ArrayList<>();

    void add(Animal a) {
        animals.add(a);
    }

    void behave() {
        animals.forEach(Animal::behave);
    }
}
```

We can add all our animals and make them all do their thing:

```
var animals = new Animals();

animals.add( dodo );
animals.add( sparrow );

animals.behave();
```

We now have a set of pluggable animal behaviours that benefit from OO polymorphism. We have used objects to encapsulate data, hide secrets and separate concerns.

The general idea here is to *prefer composition over inheritance*.

Our original problem was that inheritance and static types had been used to model dynamic behaviours. That cannot work. We found the correct way to model this. It is not how the common example does it.

I think it's time to declare "OO is dead" articles dead.

As dead as our Dodo.

Java interfaces model static, not dynamic sets of behaviour

Square extends Rectangle

At Number 2 in our chart of depressing classics is "Square extends Rectangle".

This problem unfolds when we do a good job of our Shape.draw() exercise on polymorphism, then get a bit carried away.

It's all going well, with an interface Shape that has one method draw(). We knock it out of the park adding class Circle, class Triangle and class Rectangle.

Flush with success, we decide to draw a Square.

We notice that a square is a kind of rectangle. IS-A! Boom! We can inherit from Rectangle - perfect!

Here is where we get up to:

```
1  class Graphics {
2      // Coordinates (0,0) top left of screen
3      void line( int x1, int y1, int x2, int y2 );
4  }
5
6  interface Shape {
7      void draw(Graphics g);
8  }
9
10 class Rectangle implements Shape {
```

```
11      private int xLeft;
12      private int yTop;
13      private int width;
14      private int height;
15
16      public Rectangle( int xLeft, int yTop, int width, int height ) {
17          this.xLeft = xLeft;
18          this.yTop = yTop;
19          this.width = width;
20          this.height = height;
21      }
22
23      void setWidth(int width) {
24          this.width = width;
25      }
26
27      void setHeight(int height) {
28          this.height = height;
29      }
30
31      void draw(Graphics g){
32          g.line(xLeft, yTop, xLeft+width, yTop);
33          g.line(xLeft+width, yTop, xLeft+width, yTop+height);
34          g.line(xLeft+width, yTop+height, xLeft, yTop+height);
35          g.line(xLeft, yTop+height, xLeft, yTop);
36      }
37  }
38
39  class Square extends Rectangle {
40      void setSize( int sideLength ) {
41          setWidth( sideLength );
42          setHeight( sideLength );
43      }
44  }
```

Huzzah! We're done. That Square class is all you need to draw a square, provided

you call setSize() with the length of each side you want.

That's the problem, though. In this case, nothing particularly bad happens. But we have laid a trap for later. 'Future Us' isn't going to be best pleased.

Suppose we needed an extra method that had a different behaviour for a Square as compared to a Rectangle. Let's say it was to power a dashboard, interested in counting how many squares and rectangles we had:

```
interface Shape {
    void draw( Graphics g );
    void report( Dashboard d );
}

class Rectangle implements Shape {
    // ... other code as above ...

    void report( Dashboard d ) {
        d.reportRectangle();
    }
}

class Square extends Rectangle {
    // ... other code as above ...

    void report( Dashboard d ) {
        d.reportSquare();
    }
}
```

This would work well - until you had a Rectangle object and called setWidth(5) and setHeight(5) on it.

At this point, we have an object of type Rectangle. We have set its width and height to be the same. So now, it will draw as a square. It IS-A square, mathematically. It should report to the dashboard that it is a Square.

But it isn't and it won't. It IS-NOT-A Square in Java. It is a Rectangle with equal sides. In Java, *that is not the same thing*.

It boils down to this idea that in Java, a class or interface represents a fixed type. The kind of class of an object is set when you create it. It *cannot* change according to data values of fields inside it.

In this example, mathematically, a square is a rectangle. That is in no doubt. But Java classes are unable to represent that change of type dynamically. That is not how Java was designed. It is not the model of OOP that Java supports. Other languages may do this. But not Java.

You have to work with what you have. Respect the limitations.

Java interfaces model fixed, not varying types

Inheriting implementation

The last of our three ugly ducklings is using inheritance just to pull in some extra code, without thought to IS-A relationships, Liskov LSP or anything in the domain:

```
1  class MySqlDatabase {
2      void insertSql(...) {...}
3
4      // ... other database methods ...
5  }
6  class User extends MySqlDatabase {
7      private String name ;
8
9      void setName( String name ) {
10         this.name = name;
11
12         insertSql("USER_TABLE", "NAME", name);
13     }
14 }
```

Contrived, but a real warning nevertheless.

In this example, our User class wants to save itself to our MySqlDatabase. We drag in the database code using inheritance. For sure, we save some keystrokes, as we now have available all the database methods like insertSql() available to us.

The problem with this is that a User is not related to a database in any way. A User object is not - or at least should not - be swappable for a database in all contexts.

Yes, it saves keystrokes. But it breaks our designs. Early 90s OOP was full of this stuff in the popular press and has somewhat tarnished what classes are really all about.

Inherit/Implement for substitution, not just grabbing code

Broken Shared State

This one is a bit of a straw man, really.

Objects hide secrets. One obvious secret is a changeable value. We call this ‘state’:

```
class Counter {
    private int total;

    void oneMore() {
        total++;
    }

    void display(Display d) {
        d.print( "Total so far is: ");
        d.println( total );
    }
}
```

We can create a new Counter, call oneMore() as often as we like, and display the total, using some Display object.

The criticism comes from concurrent systems, like multi-threaded Java Servlets or multi-core CPUs.

If you create one Counter, then call oneMore() *from different threads* you get strange failures. What happens is that sometimes Thread A is partway through adding one to the total, when Thread B comes along, takes the partial result it finds, adds one to that and updates total.

The value gets corrupted. That's not the best news to get that day.

OOP critics treat this as meaning that OOP cannot be used in concurrent systems. Or it is difficult to use. Or that Functional Programming (FP) is the only thing that makes sense.

The truth is, there are plenty of techniques that allow safe, simple use of objects in concurrent systems.

The simplest is to make objects immutable. If you do not share state between threads, you have no problem.

For the cases where you *do* need to share that state, in most code, that state lives in a database. A Repository pattern object will access that database. The database will manage concurrent access for you. Job done.

Where you have no database, Java provides primitives (Thread, synchronized) and some powerful high level objects (java.util.concurrent package) to simplify this. Frameworks like Akka provide Actor objects, which is a way to treat each object as its own, concurrent thing.

It's got nothing to do with OOP really. FP avoids the problem by pushing state out to the edges of the system. But we can do that anyway, using immutable objects, if we so choose. Or add synchronisation. Or use the objects with built-in concurrency.

Ordinary Bad Code

Even with an elegant OO design, you can still snatch defeat from the jaws of victory by messing up your methods.

In one project, I encountered code rather like this, from an outsource provider:

```
1  public boolean ok (Object o) {
2      boolean result = false ;
3
4      if ( o == null ) {
5          result = false
6      }
7      else if ( !(o instanceof Boolean) ) {
8          result = false ;
9      }
10     else if ( ((Boolean)o).equals(Boolean.FALSE) ) {
11         result = false ;
12     }
13     else if ( ((Boolean)o).equals(Boolean.TRUE) ) {
14         result = true ;
15     }
16     else {
17         result = false ;
18     }
19
20     return result;
21 }
```

If you are going to write methods like that, it really does not matter a fig how good your OO design is. This was a real "What *were* they thinking?" moment.

The reality is that you can get paid to write very low-quality code. That code will get reviewed by peers who also get paid to write very low quality code. That decision will be rubber-stamped by non-technical management who do not know what code quality is, nor care, so long as the deadline was met.

Hopefully, we can agree that we can do better than this.

We can, can't we?

Data Structures and Pure Functions

We've looked at how object oriented designs are driven outside-in by behaviours with hidden secrets. We've also covered how getters and setters are the ultimate evil.

Well, ok, maybe I didn't quite put it like that. But you know what I mean.

This chapter is about setting the balance straight.

Sometimes, the perfect solution to a problem is a pure data structure. Yes, a full-on, old-skool class that has *only* getters and setters.

Let's look at where they add value and why.

System Boundaries

We've seen how we can make our application talk to external systems easily by using Dependency Inversion. This provides a local interface that represents our view of that external system.

But we still need to *talk* to that system, over the network.

It should be obvious[3] that we don't actually *send* an object over the network. What would that even mean?

What we have to do is "flatten it out" in a process we call *serialisation*.

This is where we take all the private data inside that object and write it out, together with some extra metadata that says what the Class was.

This flattened out data is pure data. It's not an object. It can't have methods when it is just ones and zeroes over a network.

[3] Ahem. I say obvious. You should know that the first time I used a fax machine, I photocopied the document before I sent it. I wanted to have my own copy to keep…

Using a pure data structure at a system boundary makes perfect sense.

When using Hexagonal Architecture designs, we simply write an Adapter class at each end.

These all look similar. They take the domain object, convert it into a pure data structure, then hand that to whatever networking library we are using. A similar Adapter will be at the receiving side, mapping the pure data structure into a domain object.

Some frameworks will handle this mapping for you, leaving you to only write the domain object. This is helpful, sometimes. But often, there is an "annotation soup" that couples your domain layer directly to the external system layer.

That coupling is worth avoiding, usually. Writing your own pure data class in the adapter layer is the answer. These are often callen Data Transfer Objects (DTOs) or 'model' objects.

Fixed Data, Changing Functions

Objects work best when we know the set of behaviours we want them to have.

That way, we can write the public methods and add any private data we need afterwards. This is great when the methods are stable, but how they work might change.

Sometimes, the problem we are solving is the other way around. The data is the stable thing, and any methods that work on it are liable to change.

The classic example is a reporting system.

We've just built an e-commerce system, our own mini-Amazon. We have plenty of sales, inventory and order history. We would like to see reports on KPIs. We start off with a 'Total Sales' report.

The CFO loves this and gets very excited. She asks us for a new report on 'Top Five Selling Product Categories by Region'.

A little later, Marketing pipe up on Slack. 'Can we have a report on ROI on Facebook Ads by Category by Time of day, please?'.

The success of our reporting tools is becoming its own worst enemy. For every new report we deliver, we get asked to create two more.

If we choose to model this as an object, it gets unwieldy pretty quickly. It sprouts hundreds of pretty random-looking methods and reaches out to all kinds of other objects that it only wants the private data from.

We can use some patterns to help us; CollectingParameter, Strategy or even Observer can come into play. But it tends to obscure what's going on.

We really only care about the data for a report. The data is fixed. It comprises payment details, product data and categories. All of this stays the same. It is the behaviours - the reports themselves - that are in flux.

The sensible approach is to create a set of pure data classes, resplendent with getters and setters. I might even go as far as making them have public data *fields* and no getters, setters nor methods at all. I think this is honest. But so we don't break SonarQube (and freak out our team) we can stick to old-skool JavaBeans, with get/set pairs. That's fine in this context.

The reports can then be simple, separate classes that process these data objects, spitting out a report at the end.

An approach I like sticks with an OOP domain model, and some of those domain objects can be asked to produce a ReportingModel object. This is the get/set data structure for use with the reporting package.

Algorithms and Data Structures

This idea - a data structure with separate algorithms - is a different way of approaching a problem. The difference is where the behaviour lies.

With OOP, behaviours are primary. They are what the object is all about. Data is hidden inside an object. This makes changes to data and other object secrets easy, but adding new behaviours will cause change to calling code.

With a data structure approach, these trade-offs are inverted. Changing the data layout will ripple out changes, but adding a new function will not. Data is exposed. Behaviours are secondary.

Both approaches are useful, given the trade-off between what is more likely to change - data or behaviours.

What's important is to understand which one you are using and why. Then do it well.

Putting It All Together

Well, that's it. That's all I use day to day for building out object oriented applications.

No step-by-step plans

The first thing to notice is that there are no step by step plans here. I often wish there were.

Novices often want to learn the steps needed to build different kinds of software. They get frustrated to learn there aren't any.

What we have instead is a Toolbelt of Techniques.

It's like being a builder. You have hammers, saws and specialist tools. You have wood, metal and other materials. You know the principles of measuring, joining, load-bearing and more. It is only when you come to build a specific house that you work out which ones to use, in which order.

The house gets built from thousands of smaller jobs. We cut a hole, fit a door frame, screw a hinge. We never "build a house" with the "build a house tool".

Software is just the same. You don't "write a word processor". You represent text, save it to storage, fetch it from storage, layout a paragraph, draw a font to a display. We create objects to do the work and get them to collaborate. Over time, we build out the objects we need to create the features we want.

The trick is to approach each problem with confidence. As soon as we break down to a small enough problem, one or more of our tools will solve it.

Getting Started

The way I get started is by taking an Agile, iterative approach.

Start with a simple user story - a single, small stripe of functionality that a user would see. That might be "Add a new Customer to our system".

From that, we can imagine what kinds of objects we need. A Customer. A CustomerRepository. Some implementation of that for storage, depending on how we have decided to store things. Some kind of interface to the Customer - possibly a Web Service REST API to power a JavaScript front end.

We can begin to write tests to get small pieces of this code up and running. Refactor it into shape. Add pieces as "stripes" through the whole system. Consider our domain/external system splits. Think about making the domain ideas clear, hiding mechanisms.

Start small. Scaffold with tests. Build out.

Perfection and Pragmatism

I'm an idealist at heart. I genuinely enjoy building 'elegant solutions'. Just the right pattern. Finessed names. Good splits of where things go.

I don't aim that high in practice, though. Not quite, anyway.

Two factors frustrate perfection: deadlines and limited foresight.

Real-world code has to ship. It has to be out being used for it to have any value. And that means compromises. My advice is to pick your battles wisely. Start out with clean code and good approaches. Maintain them. But allow yourself to introduce Technical Debt. This is the deliberate use of a 'worse' technique - cutting a corner - to get code shipped now. You should always note Technical Debt and schedule it for removal in your work tracking system.

My view is that it's ok to do when you understand the trade-off you are making. What's not ok is to write awful code all the time and lie about it, saying that it is 'Tech Debt'.

Bad code is just that: Bad. As we've seen in the book, it's not like clean code takes any longer to write, when you know how. Tech Debt is all about leverage. It's like when you take out a mortgage so you can live in a house before you can afford the full price. Tech Debt like this is good.

Bad code is like borrowing fifteen grand from the bank to blow it all on a party. It's fun for the night, but pretty grim afterwards. You're paying it all back with nothing to show for it.

If deadlines force compromises on us, then our limited foresight also frustrates us.

Many times, my code gets cleaned up in a later change. I wrote it as good as I could at the time. But since then, I have *learned* more about the subject area I am modelling. I know more about what the code needs to express in its domain model. I couldn't have written any better earlier than this.

I'll always clean code up as I learn more - when it is safe to do so. By that, I mean 'when it doesn't impact anybody else'. A case in point is when you learn more about code that is already in use by others. Any change affects those users. You should not simply push changes out without letting them know and having a plan.

I also like to 'Boy Scout' code, as it's called. Leave code a little cleaner than when you got hold of it. Small, incremental improvements in clarity build up over time. When you're working on a feature, it's usually good to clean up code around it. It makes adding your feature easier and improves that code for the future.

Getting Past Stuck

Programming is a creative craft. We have our tools and raw materials. But we must find our own ways to apply them.

That requires creativity.

I used to think creativity was exclusive to poets, playwrights and fiction writers. I see that's not fully true. Anytime we have to come up with an idea of our own, one that we don't just copy from elsewhere - that is creativity.

And at times, it doesn't happen.

We can get burned out, depressed, stressed, sick or any of a myriad of other things that will just shut our creativity down. At the less dramatic end of things, we can just get stuck. We stare at that blank screen on the IDE thinking 'How on earth am I going to do this?'

As some final advice in this book, here's what I do in these situations.

First, accept it. Don't panic or feel guilty or stupid. We're just stuck. In a short time, we won't be. That's just how this works.

Create some space. Do something else. I'll often focus my brain on something I can do. Perhaps another piece of work. Perhaps I'll write something that I know about. Or stare out of the window. Or read or watch a YouTube.

Space gives your subconscious mind a chance to work in the background. Don't ask me how, but many times, that's all it takes. Literally sleeping on a problem works wonders.

Other ideas that work are to ask for help. Explain the problem to somebody else and what you are thinking about. Many times, this activity itself gives you the answer. This even has a name - 'rubber ducking'. You might as well have talked to a rubber duck. Hence the one on the front cover - feel free to use it! It is the process of organising your thoughts out loud that brings the answer.

Experimentation is another good approach. Try a little spike. I often do this inside a unit test harness, just so it is easy to run. That often shows you 'how a solution might feel'.

Always be ready to simplify an idea for now if it is just too hard. Get the simple one working first, then add extra features later. That stops 'analysis paralysis', where you can't start because the job is too big.

Further Reading

For further reading, here are my favourite software design books

Agile Software Development, Robert C Martin

This book covers it all. A nice intro to TDD. Detailed explanations of SOLID and other design ideas.

For me, the best part of this book is its large case studies. The book describes in detail how a number of realistic systems are designed, including a weather station with a GUI. The book takes you through the twists and turns of design and refactoring as you go, arriving at a good solution after the journey of discovering problems.

ISBN 978-1292025940
https://www.amazon.co.uk/gp/product/B00IZ0G6YG

Growing Object Oriented Software Guided By Tests, Freeman and Pryce

My favourite book on TDD. Often referred to as the 'London School' or the 'Mockist' approach (but not by the authors), this book is all about designing a large chat messaging system, outside-in, test-first.

It shows how you can start with basic, not particularly clean code that completely meets the early requirement, then use tests to drive you to better designs. But only if and when you need them. Covers a huge amount of ground in test-first design thinking and practice.

ISBN 978-0321503626
https://www.amazon.co.uk/dp/0321503627

Refactoring, Martin Fowler

The opening case study is absolutely wonderful. You'll want to buy the first edition to get Java examples.

It starts with some fairly average procedural code and splits it apart one step at a time. We end up with elegant object oriented code. This chapter alone can teach you a lot about how to do OOP properly.

The rest of the book is in 'cookbook' style, with detailed descriptions of individual refactor steps you can make. Interestingly, most of them have a reverse, which clearly shows that there is no 'one size fits all' with Refactoring. You need different tools to take you to the code you want.

ISBN 978-0201485677 (2nd edition with Java examples)
https://www.amazon.co.uk/gp/product/B007WTFWJ6

Design Patterns Helm, Johnson, Richards, Vlissides

When we say 'Design Patterns', this was the book that started it off. It is a catalogue of 23 patterns, with examples of how and where they are used.

As it has a long title and four authors with long names, you'll hear this book talked about as the "Gang of Four" book (often abbreviated GoF).

ISBN 978-0201633610
https://www.amazon.co.uk/dp/0201633612

Domain Driven Design, Eric Evans

Designing software around the problem domain. This book provides several techniques for keeping the domain knowledge up front and central in your code.

ISBN 978-0321125217
https://www.amazon.co.uk/dp/0321125215

Applying UML with Patterns, Craig Larman

This book should get more love. It is excellent. Craig Larman shows an approach to designing software based on his GRASP principles. It is a really well-written book, full of great OOP design advice.

ISBN 978-0131489066
https://www.amazon.co.uk/dp/0131489062

Home page for this book

https://www.viewfromthecodeface.com/javaoopdoneright

Pointing you to code samples, videos, extra stuff.

My Blog

My writings on software. You'll find tutorials and stories from my experience.
https://www.viewfromthecodeface.com

My Quora Space

Join 27,000+ others to see my answers to whatever random questions about software get asked on Quora!
https://www.quora.com/q/alanmelloronsoftware

LinkedIn

Me: https://www.linkedin.com/in/alan-mellor-15177927/

LeanPub page

Where the book was originally written. It's an excellent platform, I can recommend it. You should definitely try writing a short book - leanpub takes care of the practicalities
https://leanpub.com/javaoopdoneright

Cheat Sheet

My mental model of how I work with OOP code:

Behaviours First

Design in this order:

- public methods
- supporting private fields
- constructor
- supporting private methods
- getters where sensible
- setters (rarely)

Design Principles

- DRY - Don't Repeat Yourself. Eliminate duplication in code and tests
- KISS - Keep It Simple
- Tell Don't Ask - Do work inside the object. Do not drag out data.
- YAGNI You Ain't Gonna Need It (YAYA unless yes, you are)
- SOLID - Single responsibility. Invert/Inject dependencies. Swappable.
- Prefer Composition over Inheritance - nearly always
- Prefer interface/implements over Inheritance trees
- Test First - drive out the design with feedback, regression is a bonus
- FIRST tests - Fast, Independent, Repeatable, Self Validating, Timely
- Collaboration - which object should be doing this job?
- Hexagonal Architecture - Decouple external systems from the domain code
- Domain over Mechanism - Emphasis what is being done; hide how

Clean Code

- Say what you mean, mean what you say
- Method names describe outcomes
- Variable names describe contents

General Code Review Points

- Can a colleague skim read this?
- Can a colleague safely modify this? Or are there ‘traps’?
- Is the code the most straightforward way of implementing the method?
- Is the code easy to work with?
- Are the names as clean as they can be?
- Do all tests pass?
- Do we have enough unit test coverage of the core domain code?
- Could we replace an integration test (or higher) with a unit test?
- Should we refactor
- Is it OPTIMISED FOR CLARITY?

About the Author

Starting from age 12, Alan Mellor has four decades of experience developing software, for various companies, startups and as a freelancer.

From a humble Sinclair ZX81 home computer with 1k of RAM, Alan has progressed to creating systems for industrial automation, defence, e-commerce, games and mobile phones.

Some you may have heard of: Nokia Bounce, The Ericsson R380s smartphone, The Red Arrows flight simulator from 1985 and Fun School 2. All had Alan's code in them. Other code sits there quietly, doing its thing unnoticed. Yet more has been consigned to the great `/dev/null` of history.

More recently, Alan has been involved with training UK Level 4 Apprentices. He has designed and delivered content that hopefully helps 'switch the light on' about programming.

Alan also enjoys dabbling variously with guitars, electronics, videography and cheeseboards. You just can't beat a great Roquefort with Rioja.

Thanks

BJSS Limited and Manchester Digital for opportunities to use and teach this stuff.

Steven Taylor - great suggestions on the first draft (despite more work!)

My Mum. That ZX81 didn't buy itself. You made my career happen.

Stephanie, Katy, Jake. Who would have guessed a 1980s computer nerd would end up surrounded by amazing humans he can call 'family'. What a privilege.

Katy for front cover art https://www.redbubble.com/people/kath-ryn/shop