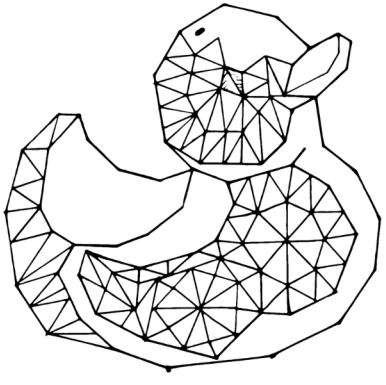
JAVA OOP
DONE RIGHT

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 Many of them thought OOP in Java was 'verbose' or 'complex' or 'outdated'.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

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Good Names

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Method names: Tell me the outcome

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Variable names: Tell me the contents

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Design methods around behaviours, not data

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Hidden data - No getters, no setters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Aggregates: More than one

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Greeting more than one user

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Type safe collections, not raw arrays

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Using forEach - not a loop

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Aggregate methods work on all the things

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Common themes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Collaboration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Basic Mechanics

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

How to decide: field or parameter?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Example: Simple Point of Sale

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Review of design so far

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Exercise: Total Amount

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Test Driven Development

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

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:

```
class BillCalculator {
}
```

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.

```
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.

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

Arrange, Act, Assert - a rhythm inside each test

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Red, Green, Refactor - a rhythm in between tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Second test: Adding an item gives us the right total

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Designing the second feature

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

TDD Steps - Too much? Too little?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

YAGNI - You Ain't Gonna Need It

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

YAYA - Yes, You Are

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Optimise for Clarity with well-named tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

TDD and OOP - A natural fit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

FIRST Tests are usable tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Real-world TDD

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

The Good

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

The Bad

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

The Ugly

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Polymorphism - The Jewel in the OOP Crown

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Classic example: Shape.draw()

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

The Shape Interface

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Tell Don’t Ask - the key to OOP

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

The SOLID Principles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

The five SOLID principles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

SRP Single Responsibility - do one thing well

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

What is 'one thing', anyway?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

DIP Dependency Inversion: Bring out the Big Picture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

What is an 'inverted dependency'?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Why is 'new' such a problem?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Inverting the input Dependency

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Making a concrete KeyboardInput class

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Dependency Injection - using our inverted dependency

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Swappable input sources

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Inverting the output to display

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Inversion - Injection: two sides of the same coin

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

LSP Liskov Substitution Principle - Making things swappable

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

When Shapes go Bad

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Substitutability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

OCP Open/Closed Principle - adding without change

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Strategy Pattern: Externalising behaviour

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

ISP Interface Segregation Principle - honest interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Bad Example: TV Controls

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Fixing our ISP violation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Redesigning to Command objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Pragmatics: I would choose to do it wrong

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

TDD and Test Doubles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Test Doubles - Stubs and Mocks

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

DIP for Unit Tests - Stubs and Mocks

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Mocking libraries

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Self-Shunt mocks and stubs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Refactoring

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

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

```
class User {
    private final String text ;

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

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

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);
```

```
10      }
11  }
```

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

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Change Method Signature

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Extract Parameter Object

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Can we refactor anything into anything else?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Hexagonal Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

The problems of external systems

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

The Test Pyramid

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Removing external systems

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

The Hexagonal Model

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Inversion / Injection: Two sides of the same coin

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Handling Errors

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Three kinds of errors

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

The null reference

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Null object pattern

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Zombie object

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Exceptions - a quick primer

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Design By Contract, Bertrand Meyer style

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Fatal errors: Stop the world!

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Combined approach: Fixable and non-fixable errors

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Which approach is best?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

NullPointerException

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Application Specific Exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Error object

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Optionals - Java 8 streams approach

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Review: Which approach to use?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Design Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Mechanism and Domain

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Patterns: Not libraries, not frameworks

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

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

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Adapter

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Command

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Composite

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Facade

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Builder

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Repository

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Query

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Simple Query Object

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

CollectingParameter

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Item-Item Description

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Moment-Interval

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

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
    }
}
```

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)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Aggregate

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Cache

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Decorator

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

External System (Proxy)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Order-OrderLineItem

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Request-Service-Response

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Anti-Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Design Pattern Soup

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Unneeded Flexibility

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Mechanism Madness

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

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

Bird extends Animal

Square extends Rectangle

Inheriting implementation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Broken Shared State

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Ordinary Bad Code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Data Structures and Pure Functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

System Boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Fixed Data, Changing Functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Algorithms and Data Structures

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Putting It All Together

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

No step-by-step plans

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Getting Started

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Perfection and Pragmatism

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Getting Past Stuck

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Further Reading

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Agile Software Development, Robert C Martin

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Growing Object Oriented Software Guided By Tests, Freeman and Pryce

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Refactoring, Martin Fowler

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Design Patterns Helm, Johnson, Richards, Vlissides

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Domain Driven Design, Eric Evans

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Applying UML with Patterns, Craig Larman

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Home page for this book

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

My Blog

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

My Quora Space

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

LinkedIn

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

LeanPub page

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Cheat Sheet

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Behaviours First

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Design Principles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

Clean Code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

General Code Review Points

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/javaoopdoneright.

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

Buy the book!

What you read was just a sample of various chapters. If you like the style, why not buy the full book? Go on - treat yourself!