

SED

# Refactoring

Dr Robert Chatley - [rbc@imperial.ac.uk](mailto:rbc@imperial.ac.uk)



## Refactoring

*"By continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code."*

Joshua Kerievsky defines refactoring as being a process of improving the design of a piece of code, without changing its behaviour. We can change the way that something is implemented in order to improve the design, but to its clients, it should behave in the same way.

When we do TDD, we should only refactor when we are in a green state. Then we can use our tests to check that behaviour is preserved before and after our refactoring - we only changed the structure without affecting the behaviour as observed through the public API. In a red state (when any test is failing), we do not have this guarantee, so refactoring is risky. Refactor on green.



## Technical Debt

Photo by Robert Scoble

It is all too common to see features that have been added quickly to a system, put in in an inelegant way. Perhaps copying and pasting previous similar code. We often know we should fix this, but we put it off. This way *technical debt* builds up, and if we don't pay it off then our system becomes harder and harder to work with over time.

Sometimes we consciously take on technical debt to meet a deadline, but we must be aware of the costs of leaving the repayment until later.



Refactoring is a technical practice that should be applied little and often to continuously improve the quality of our design. There are various books by, notably, Martin Fowler and Joshua Kerievsky that detail individual techniques that may be applied.

Modern development tools incorporate powerful refactoring tools that support performing many refactorings automatically. More powerful tools are available for statically-typed languages, as the type system gives the more tools more information with which they can analyse the code.

## Mechanical Transformation

Refactoring should be code transformation

An application of tools

Combine steps - compound refactorings

Do not alter behaviour

Photo by Reilly Butler

We can combine sets of small transformations to achieve larger refactorings. By using modern development tools we can automate many of these transformations and perform them more quickly and more reliably than we could by editing program text manually.

Different refactorings have names that, like with design patterns, give us a vocabulary to discuss proposed design changes with fellow engineers.

## Hygiene

Continuous small-scale refactoring keeps your codebase easy to work with

Avoids major surgery

Photo by Stuart Pilbrow

By continually applying these techniques to make small improvements to our design we keep our system malleable. Ideally we can continue to add features at a constant rate. It is difficult to justify or explain refactoring tasks that are done as big design changes - it also takes longer to achieve them as technical debt builds up.

# A Catalogue of Refactorings

## What would you do with this code?

```
class Invoice {
    ...
    public void print(PrintStream stream) {
        stream.println("ACME Ltd");
        stream.println("123 High St");
        stream.println("London");
        stream.println("SW12 1AA");

        int totalCost = 0;

        for(ListItem item : items) {
            StringBuffer line = new StringBuffer();
            line.append(item.name);
            line.append("\t");
            line.append(item.price);
            line.append("\t");
            line.append(item.quantity);
            line.append("\t");
            line.append(item.quantity * item.price);
            totalCost += item.quantity * item.price;
            stream.println(line.toString());
        }

        stream.println("Total" + "\t\t\t" + totalCost);
    }
}
```

Let's look at how we could improve the design of even this relatively short code extract.

Over the next few slides we will look at a number of possible refactorings that we could do, and show how we can use our IDE to perform these automatically in a reliable way. Lots of common refactorings are implemented as automatic transformations by modern tools, and act more as transformations of the syntax tree, than textual edits.

## Compose Method

```
class Invoice {
    ...
    public void print(PrintStream stream) {
        stream.println("ACME Ltd");
        stream.println("123 High St");
        stream.println("London");
        stream.println("SW12 1AA");

        int totalCost = 0;

        for(ListItem item : items) {
            StringBuffer line = new StringBuffer();
            line.append(item.name);
            line.append("\t");
            line.append(item.price);
            line.append("\t");
            line.append(item.quantity);
            line.append("\t");
            line.append(item.quantity * item.price);
            totalCost += item.quantity * item.price;
            stream.println(line.toString());
        }

        stream.println("Total" + "\t\t\t" + totalCost);
    }
}
```

```
class Invoice {
    ...
    public void print(PrintStream stream) {
        printAddress(stream);

        int totalCost = 0;

        for (ListItem item : items) {
            StringBuffer line = new StringBuffer();
            line.append(item.name);
            line.append("\t");
            line.append(item.price);
            line.append("\t");
            line.append(item.quantity);
            line.append("\t");
            line.append(item.quantity * item.price);
            totalCost += item.quantity * item.price;
            stream.println(line.toString());
        }

        stream.println("Total" + "\t\t\t" + totalCost);
    }

    private void printAddress(PrintStream stream) {
        stream.println("ACME Ltd");
        stream.println("123 High St");
        stream.println("London");
        stream.println("SW12 1AA");
    }
}
```

extract method



We can break down the long method to make it shorter. By composing it into chunks we give ourselves the opportunity to introduce a name for a concept, and to raise the level of abstraction. This helps to improve clarity. Good IDE tools, especially for statically typed languages can help us perform these transformations.

## Compose Method

```
class Invoice {
    ....
    public void print(PrintStream stream) {
        printAddress(stream);
        int totalCost = 0;
        for (LineItem item : items) {
            StringBuffer line = new StringBuffer();
            line.append(item.name);
            line.append("\t");
            line.append(item.price);
            line.append("\t");
            line.append(item.quantity);
            line.append("\t");
            totalCost += item.quantity * item.price;
            stream.println(line.toString());
        }
        stream.println("Total" + "\t\t\t" + totalCost);
    }
    private void printAddress(PrintStream stream) {
        stream.println("ACME Ltd");
        stream.println("123 High St");
        stream.println("London");
        stream.println("SW12 1AA");
    }
}

class Invoice {
    ....
    public void print(PrintStream stream) {
        printAddress(stream);
        int totalCost = 0;
        for (LineItem item : items) {
            printLineItem(stream, item);
            totalCost += item.quantity * item.price;
        }
        stream.println("Total" + "\t\t\t" + totalCost);
    }
    private void printLineItem(PrintStream stream, LineItem item) {
        StringBuffer line = new StringBuffer();
        line.append(item.name);
        line.append("\t");
        line.append(item.price);
        line.append("\t");
        line.append(item.quantity);
        line.append("\t");
        stream.println(line.toString());
    }
    private void printAddress(PrintStream stream) {
        stream.println("ACME Ltd");
        stream.println("123 High St");
        stream.println("London");
        stream.println("SW12 1AA");
    }
}
```

Sometimes we would like to extract a method, but we need to do another transformation first before this is possible. In this example we reorder a couple of the lines to make the lines we want to extract consecutive. This could be risky - ideally we'd like to run a test to check that behaviour is preserved.

## Separate Responsibility

```
class Invoice {
    ....
    public void print(PrintStream stream) {
        printAddress(stream);
        int totalCost = 0;
        for (LineItem item : items) {
            printLineItem(stream, item);
            totalCost += item.quantity * item.price;
        }
        stream.println("Total" + "\t\t\t" + totalCost);
    }
    private void printLineItem(PrintStream stream, LineItem item) {
        StringBuffer line = ....
    }
}

class Invoice {
    ....
    public void print(PrintStream stream) {
        printAddress(stream);
        for (LineItem item : items) {
            printLineItem(stream, item);
        }
        int totalCost = 0;
        for (LineItem item : items) {
            totalCost += item.quantity * item.price;
        }
        stream.println("Total" + "\t\t\t" + totalCost);
    }
    private void printLineItem(PrintStream stream, LineItem item) {
        StringBuffer line = ....
    }
}
```

Looking at this code, it's looking better, but we have things going on at different levels of abstract. First of all we print the address, but then we have a for loop that does two things in the body. We can separate this into two simpler loops, which may give us more options or where to go next.

## Compose Method

```
class Invoice {
    ....
    public void print(PrintStream stream) {
        printAddress(stream);
        for (LineItem item : items) {
            printLineItem(stream, item);
        }
        int totalCost = 0;
        for (LineItem item : items) {
            totalCost += item.quantity * item.price;
        }
        stream.println("Total" + "\t\t\t" + totalCost);
    }
    private void printLineItem(PrintStream stream, LineItem item) {
        StringBuffer line = ....
    }
}

class Invoice {
    ....
    public void print(PrintStream stream) {
        printAddress(stream);
        for (LineItem item : items) {
            printLineItem(stream, item);
        }
        int totalCost = totalCost();
        stream.println("Total" + "\t\t\t" + totalCost);
    }
    private int totalCost() {
        int totalCost = 0;
        for (LineItem item : items) {
            totalCost += item.quantity * item.price;
        }
        return totalCost;
    }
    private void printLineItem(PrintStream stream, LineItem item) {
        StringBuffer line = ....
    }
}
```

Now we can extract the lines that are only about calculating the total into a smaller, more cohesive method, and introduce a name raising the level of abstraction in the caller.

## Inline Variable

```
class Invoice {
    ....
    public void print(PrintStream stream) {
        printAddress(stream);
        for (LineItem item : items) {
            printLineItem(stream, item);
        }
        int totalCost = totalCost();
        stream.println("Total" + "\t\t\t" + totalCost);
    }
    private int totalCost() {
        int totalCost = 0;
        for (LineItem item : items) {
            totalCost += item.quantity * item.price;
        }
        return totalCost;
    }
    private void printLineItem(PrintStream stream, LineItem item) {
        StringBuffer line = ....
    }
}
```

```
class Invoice {
    ....
    public void print(PrintStream stream) {
        printAddress(stream);
        for (LineItem item : items) {
            printLineItem(stream, item);
        }
        stream.println("Total" + "\t\t\t" + totalCost());
    }
    private int totalCost() {
        int totalCost = 0;
        for (LineItem item : items) {
            totalCost += item.quantity * item.price;
        }
        return totalCost;
    }
    private void printLineItem(PrintStream stream, LineItem item) {
        StringBuffer line = ....
    }
}
```

inline



We now no-longer need a temporary variable for the total cost, and can use an automated refactoring to inline its usages, reducing the number of elements we have in the method.

## Compose Method

```
class Invoice {
    ....
    public void print(PrintStream stream) {
        printAddress(stream);
        for (LineItem item : items) {
            printLineItem(stream, item);
        }
        stream.println("Total" + "\t\t\t" + totalCost());
    }
    private int totalCost() {
        int totalCost = 0;
        for (LineItem item : items) {
            totalCost += item.quantity * item.price;
        }
        return totalCost;
    }
    private void printLineItem(PrintStream stream, LineItem item) {
        StringBuffer line = ....
    }
}
```

```
class Invoice {
    ....
    public void print(PrintStream stream) {
        printAddress(stream);
        for (LineItem item : items) {
            printLineItem(stream, item);
        }
        printTotal(stream);
    }
    private void printTotal(PrintStream stream) {
        stream.println("Total" + "\t\t\t" + totalCost());
    }
    private int totalCost() {
        int totalCost = 0;
        for (LineItem item : items) {
            totalCost += item.quantity * item.price;
        }
        return totalCost;
    }
    private void printLineItem(PrintStream stream, LineItem item) {
        StringBuffer line = ....
    }
}
```

One of our goals is that the entire content of any method should be at the same level of abstraction. We're not quite there yet, as we print an address, and then each lineitem, but at the end, we talk about strings and tabs. Composing another method gives us a constant level of abstraction within the method body.

## Duplication between classes

```
class Invoice {
    public void print(PrintStream p) {
        p.println("ACME Ltd");
        p.println("123 High St");
        p.println("London");
        p.println("SW12 1AA");
        int totalCost = 0;
        for (LineItem item : items) {
            StringBuffer line = new StringBuffer();
            buffer.append(item.name);
            buffer.append("\t");
            buffer.append(item.price);
            buffer.append("\t");
            buffer.append(item.quantity);
            buffer.append("\t");
            buffer.append(item.quantity * item.price);
            totalCost += item.quantity * item.price;
            p.println(line.toString());
        }
        p.println("Total" + "\t\t\t" + totalCost);
    }
}
```

```
class HeadedLetter {
    public void print(PrintStream stream) {
        stream.println("ACME Ltd");
        stream.println("123 High St");
        stream.println("London");
        stream.println("SW12 1AA");
        // ...
    }
}
```

extract to common class

One of the most common things we want to refactor is when there is duplication in the code. Sometimes things are similar, and we can first work to make them exactly the same, exposing the duplication, and then refactor it away, perhaps into another object.

## Extract Class

1) extract method

2) create (empty) class

```
class HeadedLetter {
    public void print(PrintStream stream) {
        stream.println("ACME Ltd");
        stream.println("123 High St");
        stream.println("London");
        stream.println("SW12 1AA");
        // ...
    }
}

class HeadedLetter {
    public void print(PrintStream stream) {
        printAddress(stream);
        // ...
    }

    private void printAddress(PrintStream stream) {
        stream.println("ACME Ltd");
        stream.println("123 High St");
        stream.println("London");
        stream.println("SW12 1AA");
    }
}

class Address {
}
```

Here we show several small transformations that go together to allow introduction of a new class. Following these steps allows us to use the IDE to apply automated transformations, rather than entering text manually which can be error prone.

## Extract Class

3) add (unused) parameter

4) move method onto parameter



```
class HeadedLetter {
    public void print(PrintStream stream) {
        printAddress(stream, new Address());
        // ...
    }

    private void printAddress(PrintStream stream, Address address) {
        stream.println("ACME Ltd");
        stream.println("123 High St");
        stream.println("London");
        stream.println("SW12 1AA");
    }
}

class Address {
}
```

```
class HeadedLetter {
    public void print(PrintStream stream) {
        new Address().printAddress(stream);
        // ...
    }
}

class Address {
    void printAddress(PrintStream stream) {
        stream.println("ACME Ltd");
        stream.println("123 High St");
        stream.println("London");
        stream.println("SW12 1AA");
    }
}
```

Once we have created an instance of our new empty class, we can move methods on to it and the IDE will update the caller.

## Duplication between classes

use extracted class

```
class Invoice {
    public void print(PrintStream p) {
        new Address().printAddress(p);
        int totalCost = 0;

        for(LineItem item : items) {
            StringBuffer line = new StringBuffer();
            buffer.append(item.name);
            buffer.append("\t");
            buffer.append(item.price);
            buffer.append("\t");
            buffer.append(item.quantity);
            buffer.append("\t");
            buffer.append(item.quantity * item.price);
            totalCost += item.quantity * item.price;
            p.println(line.toString());
        }
        p.println("Total" + "\t\t\t" + totalCost);
    }
}

class HeadedLetter {
    public void print(PrintStream stream) {
        new Address().printAddress(stream);
        // ...
    }
}
```

Then we can use our new class in place of the code that was previously cloned.


## Rename to Tidy Up

```
class Invoice {
    public void print(PrintStream p) {
        new Address().printTo(p);
        int totalCost = 0;

        for(LineItem item : items) {
            StringBuffer line = new StringBuffer();
            buffer.append(item.name);
            buffer.append("\t");
            buffer.append(item.price);
            buffer.append("\t");
            buffer.append(item.quantity);
            buffer.append("\t");
            buffer.append(item.quantity * item.price);
            totalCost += item.quantity * item.price;
            p.println(line.toString());
        }
        p.println("Total" + "\t\t\t" + totalCost);
    }
}

class HeadedLetter {
    public void print(PrintStream stream) {
        new Address().printTo(stream);
        // ...
    }
}
```

rename



Now that we have removed that duplication, we can see if there is anything that we can do to add to the clarity of the code. We see there is a little bit of duplication in the name of the call

`new Address().printAddress(stream)`, so we rename the method to make the call read better.

## Replace Conditional with Polymorphism

```
HeadedLetter letter = new HeadedLetter();
if (letter.isImportant()) {
    letter.sendByCourierTo(recipient, address);
} else {
    letter.sendByStandardMailTo(recipient, address);
}
```

1) extract & move method

```
HeadedLetter letter = new HeadedLetter();
letter.sendTo(recipient, address);

class HeadedLetter implements Correspondence {
    public void sendTo(Person recipient, Address address) {
        if (isImportant()) {
            sendByCourierTo(recipient, address);
        } else {
            sendByStandardMailTo(recipient, address);
        }
    }
}
```

2) extract interface

```
public interface Correspondence {
    void sendTo(Person recipient, Address address);
}
```

In object-oriented programming we do not like conditional statements. As a caller I do not want to make a decision based on information I query from my collaborator - I would rather they made the decision and I did not have to know. We can replace conditionals with polymorphism to achieve this.

## Replace Conditional with Polymorphism

```
class ConfidentialLetter implements Correspondence {
    ...
    public void sendTo(Person recipient, Address address) {
        sendByCourierTo(recipient, address);
    }
    private void sendByCourierTo(Person recipient, Address address) {
        ...
    }
}

class GeneralCircularLetter implements Correspondence {
    ...
    public void sendTo(Person recipient, Address address) {
        sendByStandardMailTo(recipient, address);
    }
    private void sendByStandardMailTo(Person recipient, Address address) {
        ...
    }
}
```

3) create two variants that implement the same interface

4) use polymorphically

```
for (Correspondence letter : postBag) {
    letter.sendTo(recipient, address);
}
```

We extract an interface, and implement two different versions that implement the two behaviours, and just use the interface type in the caller.