

TDD and Refactoring

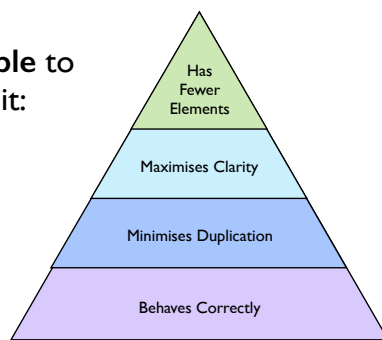
Dr Robert Chatley - rbc@imperial.ac.uk



@rchatley #doc220

Four Elements of Simple Design

A design is simple to the extent that it:

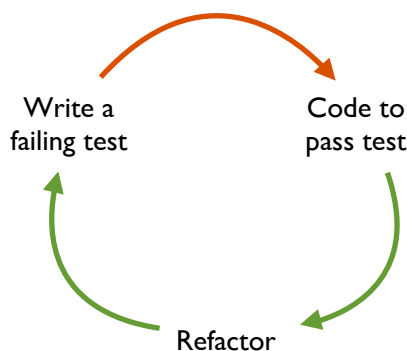


<http://www.jbrains.ca/permalink/the-four-elements-of-simple-design>

#doc220

Looking back at J.B. Rainsberger's four elements of simple design, we see that the fundamental element is that the software behaves correctly. It's no good having a very elegant design that does the wrong thing. In this lecture we look at applying Test-Driven Development (TDD) as a technique for ensuring correctness, and then *refactoring* as a technique to iteratively improve the design of our code. Minimising duplication helps to make the design more maintainable and hence more robust. If code is duplicated, then making a change to it may mean making the same change in multiple places, which is more work than we would like. By aiming to maximise clarity, we work to improve comprehension of the design by humans. Rainsberger's fourth principle - to reduce the number of elements - aims to make the overall size of the system smaller, making it simpler and easier to comprehend.

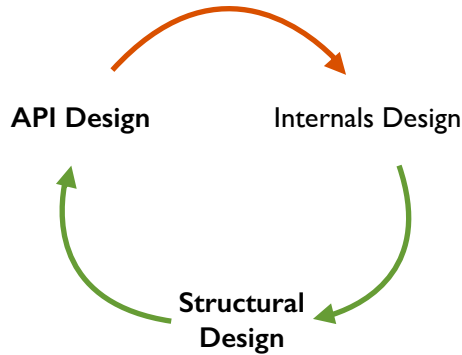
TDD Cycle



#doc220

When following the Test-Driven Development (TDD) practice, we work in a cycle. We start by writing a test. We write the test first, before writing the implementation. This helps us to specify what we want the code we are about to write to do. How do we expect it will behave when it works properly? Once we have written a test, we watch it fail. This is expected, as we haven't implemented the feature yet. Then we write the simplest possible piece of code we can to make the test pass. After this we *refactor* our design to clean up, remove any duplication, improve clarity etc etc. Then we begin the cycle again. When working with unit tests this cycle should be short, and provide us very rapid feedback about our code.

TDD is a Design Activity



#doc220

TDD is not only about testing. It is a design process. Whenever we write a new test, we are writing an executable specification of how our code should behave. We are designing its public API. Next, when we make a new test pass we know that we have software that behaves correctly. To get to this point we have to do some design of the internal logic of the object. Once we get to a green state, with tests passing, we have the opportunity to improve and simplify our design, perhaps improving structure, removing duplication or increasing the clarity of the code. The tests give us a safety net to make changes without breaking functionality.

Behaviour Driven Development (BDD)

```
CustomerLookup
- finds customer by id
- fails for duplicate customers
- ...
```

```
public class CustomerLookupTest {
    @Test
    findsCustomerById() {
        ...
    }
    @Test
    failsForDuplicateCustomers() {
        ...
    }
}
```



<http://dannorth.net/introducing-bdd/>

#doc220

We want to use tests to verify the behaviour of our system. A well-written set of tests acts as an executable specification for the software, and documents the behaviour. Dan North coined the term Behaviour Driven Development to encourage people to think about the required behaviour (rather than implementation detail) when doing TDD. We can write our specification first in natural language, and then use that to form our test cases and test names.

Developing an Object

```
FibonacciSequence
- defines the first two terms to be one
- has each term equal to the sum of the previous two
- is not defined for negative terms
```

#doc220

We start by writing down a few behavioural properties of the object we want to create - an informal specification. You can see how this can act as documentation for how the object behaves. Next we will translate these into automated tests.

Developing an Object

```
FibonacciSequence
- defines the first two terms to be one
- has each term equal to the sum of the previous two
- ...
```

```
public class FibonacciSequenceTest {

    @Test public void
    definesTheFirstTwoTermsToBeOne() {
        ...
    }

    @Test public void
    hasEachTermEqualToTheSumOfThePreviousTwo() {
        ...
    }
}
```

#doc220

This example is in Java using JUnit as a test framework. We create a test class `FibonacciSequenceTest` to accompany our (to be written) `FibonacciSequence` class. We define the first test methods one at a time with names matching the textual specification. The `@Test` annotation tells JUnit to treat this method as a test.

We will probably want to include some assertions in the bodies of our tests. In JUnit we can write these in the form

```
assertTrue(x);
assertFalse(x);
assertThat(x, is(y));
```

Developing an Object

```
FibonacciSequence
- defines the first two terms to be one
- has each term equal to the sum of the previous two
- is not defined for negative terms
- can be iterated through
```

#doc220

We step through the specification on line at a time. For each line we add a test, fill it in to make appropriate assertions on the object under test, and then complete the implementation to make it pass. We then assess our current design to see if we can make it simpler or cleaner and tidy it up. Then we repeat the cycle with the next requirement. By following this cycle strictly we make sure that we never add code that isn't being tested.

RSpec - TDD in Ruby

```
FibonacciSequence
- defines the first two terms to be one
- has each term equal to the sum of the previous two
- ...
```

```
describe FibonacciSequence do

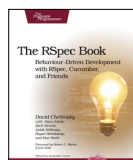
  fib = FibonacciSequence.new

  it "defines the first two terms to be 1" do
    fib.term(0).should == 1
    fib.term(1).should == 1
  end

  it "has each term equal to the sum of the previous two" do
    fib.term(2).should == 2
    fib.term(3).should == 3
  end
  ...
end
```

#doc220

<http://pragprog.com/book/achdb/the-rspec-book>



TDD is practised widely in current software development. It isn't specific to Java. There are unit testing tools for almost every language, and we can apply the principles of TDD to help ensure quality regardless of the language we are working in. Here we show an example in Ruby using RSpec.

Jasmine - TDD in JavaScript

FibonacciSequence

- defines the first two terms to be one
- has each term equal to the sum of the previous two
- ...

```
describe("Fibonacci Sequence", function() {  
  var fib = new FibonacciSequence();  
  
  it("defines the first two terms to be 1", function() {  
    expect(fib.term(0)).toEqual(1);  
    expect(fib.term(1)).toEqual(1);  
  });  
  
  it("has each term equal to the sum of the previous two", function() {  
    expect(fib.term(2)).toEqual(2);  
    expect(fib.term(3)).toEqual(3);  
    expect(fib.term(5)).toEqual(8);  
  });  
});
```



<http://javascript.crockford.com>
<http://pivotal.github.com/jasmine>

#doc220

Here is another example of developing the FibonacciSequence, this time in JavaScript and using the unit testing tool Jasmine. You can see that the pattern is the same, even though the way that the tests are expressed varies slightly.



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."

#doc220

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.



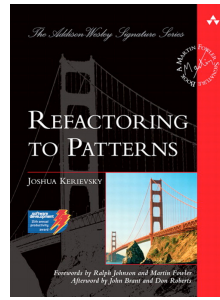
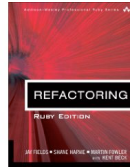
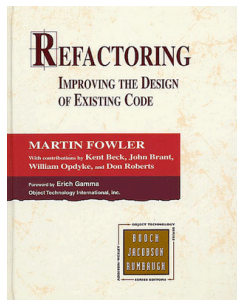
Photo by Robert Scoble

Technical Debt

#doc220

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.



#doc220

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 (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");
            line.append(item.quantity * item.price);
            totalCost += item.quantity * item.price;
            stream.println(line.toString());
        }
        stream.println("Total" + "\t\t\t" + totalCost);
    }
}
```

#doc220

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 refactoring 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 (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");
            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 (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");
            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



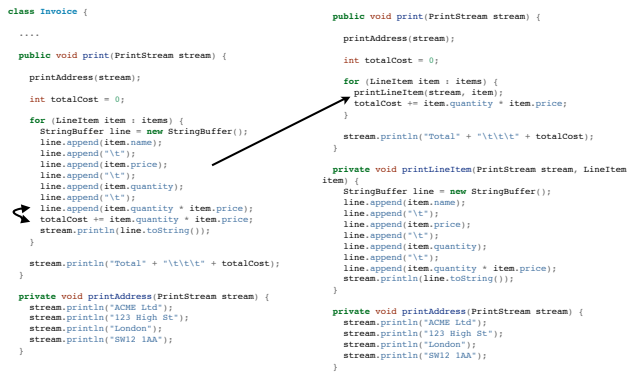
#doc220

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");
            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");
    }
}
```



```
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");
        line.append(item.quantity * item.price);
        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");
    }
}
```

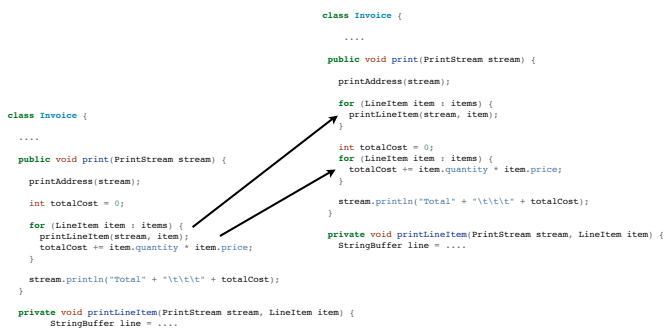
#doc220

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 = ....
    }
}
```

#doc220

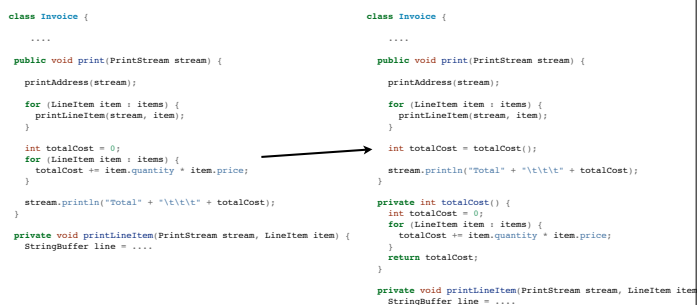
Looking at this code, it's looking better, but we have things going on at different levels of abstract. First of all we print the address, but then we have a for loop that does two things in the body. We can separate this into two simpler loops, which may give us more options or where to go next.

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 = ....
    }
}
```

#doc220

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



#doc220

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 = ....
    }
}
```

#doc220

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

#doc220

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 {
}
```

#doc220

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



move: F6

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

#doc220

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);
        // ...
    }
}
```

#doc220


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



#doc220

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);
}

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);
        }
    }
}

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

1) extract & move method

2) extract interface

#doc220

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) {
        ...
    }
}

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

3) create two variants that implement the same interface

4) use polymorphically

#doc220

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