220

Re-use and Extensibility

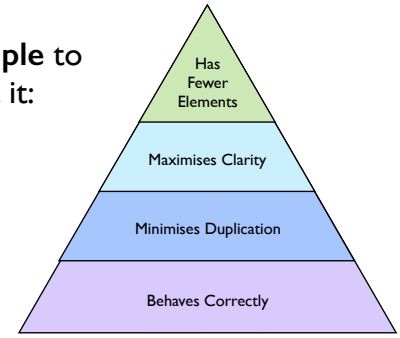
Dr Robert Chatley - rbc@imperial.ac.uk

@rchatley **#doc220**

This section of the course is about writing our code in a way that promotes reusable components, and allows us to build systems that are extensible, so that we can change them over time to keep up with changing requirements.

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

In previous classes we looked at using Test-Driven Development to provide a level of assurance that the behaviour of our software is correct. We can now focus on higher levels of the design pyramid. We can change the code however we like to improve its design, as long as our tests still pass. We will look at different ways that we can structure our code, to help us to remove duplication, and to make maintenance, change and evolution easier.

Code Example: Encoder for "Encryption"

```
ReverseEncoder
- reverses each word in the input, e.g.
  "abc 123" => "cba 321"
```

#doc220

Here we'll develop a new example, using TDD to write the executable specification. This example is an "encryption" program that encodes strings by reversing each word individually.

A Different Encoder

```
DoublingEncoder
- repeats each word in the input, e.g.
  "abc 123" => "abcabc 123123"
```

#doc220

Now let us look at developing a second, related, class. Here we decide to develop a new flavour of encoder, this time repeating each word so that it appears twice in the output. We can write the specification in just the same way that we did before. On implementing the second version, we may well notice some obvious duplication in the code between our two classes. Thinking back to our design principles, we remember that we want to try to remove duplication to improve the design and reduce maintenance costs. Let's think how we might do this.

Template Method

Requirements change over time - we need to adapt to them. When you need to vary a small part of an algorithm.

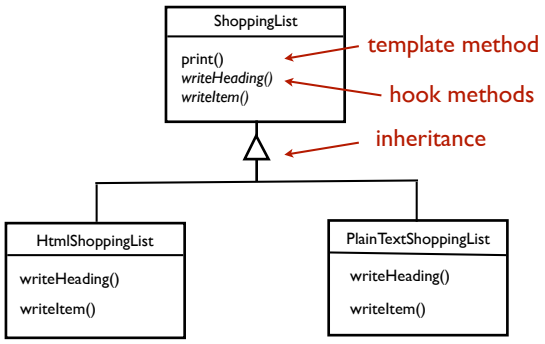
Define a skeleton and defer the varying steps to subclasses.

Separate things that change from things that stay the same

#doc220

Looking at our two different encryption classes, how might we remove the duplication between our two implementations? One possibility is to extract a superclass to deal with the common code, and just to vary a small part of the algorithm - the part that calculates a particular term - in the subclass. This is known as the Template Method pattern.

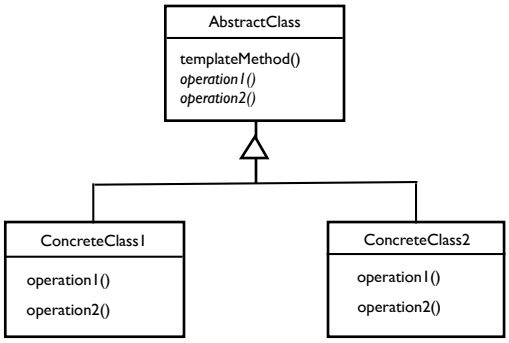
Template Method : Example



#doc220

We define the main part of the algorithm in the superclass, but call out to some abstract hook methods, which will be overridden in subclasses to specify particular behaviour for the different variants. Here we separate code that stays the same for all number sequences (in the superclass) from code that varies between the different types, which we put into the subclasses. If we need to update the core algorithm, we can do it in one place, and it applies to all of specialisations.

Template Method : pattern



#doc220

This is an example of what is sometimes called the Hollywood Principle. Don't call us, we'll call you. The concrete classes don't call up into the superclass, they just define methods that will be called when the superclass needs them. We can add new subclasses with new specialisations of the behaviour without having to make changes to the superclass, and without duplicating the core of the algorithm,

---

Open-Closed Principle

You should be able to extend a class's behaviour, without modifying it.

#doc220

The implementation using the Template Method pattern exemplifies the Open-Closed Principle, which is one of Robert Martins "SOLID" principles, a set of five principles for good object-oriented design. You can read all about the five principles in the following article.

http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

---

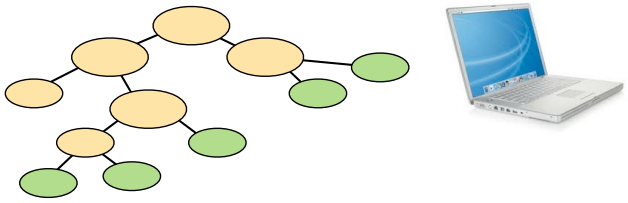
"Modules should be open for extension but closed to modification"

Bertrand Meyer, ETH

#doc220

The Open-Closed principle means that we should aim to produce re-usable, extensible modules that are flexible, but can be specialised without changing their original source code. It comes from the work of the Swiss professor Bertrand Meyer. In Martin's article, he writes "We should write our modules so that they can be extended, without requiring them to be modified. In other words we want to be able to change what the modules do, without changing the source code of the modules."

Separate out the **things that change** from those that **stay the same**

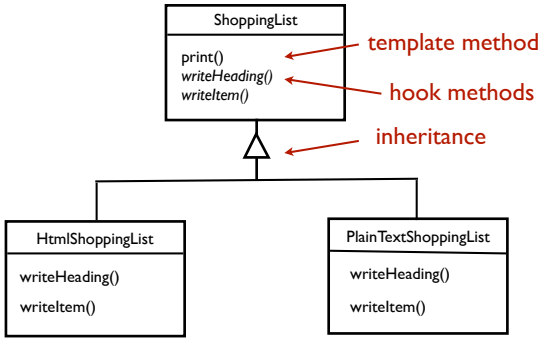


Change behaviour by **adding new code**, not by changing **existing working code**

#doc220

In our object-oriented design, we aim to separate the parts of the system that remain the same across variations, and pull out the specialisations into separate objects. This way we can build many different specialisations around the same core. This a core principle of designing maintainable systems, especially when working on large projects with large teams. Separate out the things that change from those that stay the same.

Template Method : Example



#doc220

The Template Method pattern follows the Open-Closed principle in that the core of the algorithm is in the superclass, and does not change. The hook methods allow us to override and specialise a particular part of the algorithm in each of the subclasses, but without having to alter or recompile the superclass.

What is **Bad** Design?

Rigidity

Fragility

Immobility

#doc220

But the Template Method pattern does have a problem. Let's think back to the possible design problems that Robert Martin identified. Does our Template Method design exhibit any of these?

Rigidity - where software is hard to change
Fragility - where when we change one part, other parts break unexpectedly
Immobility - where it is hard to reuse elements of the code in other applications

Coupling and Cohesion

Aim for **low coupling** between classes

Aim for **high cohesion** within each class

#doc220

In the Template Method pattern, we have a tight coupling between the subclasses and the superclass. We cannot use the subclasses independently, or use them in a different system, because they depend on the named superclass. We would have to take the superclass along with us in order to reuse the subclasses. This is a problem of *immobility* caused by *coupling*.

Coupling Metrics

Afferent coupling (Ca) : A class's **afferent** couplings is a measure of how many other classes use the specific class - a measure of the class's **responsibility**.

Efferent coupling (Ce) : A class's **efferent** couplings is a measure of how many different classes are used by the specific class - a measure of the class's **independence**.

#doc220

There are two different types of coupling: afferent end efferent. Both will appear to different degrees in different parts of any system. A system with no coupling at all would not do anything, as it would just be a collection of independent, unconnected parts.

Afferent coupling measures the number of dependencies that *arrive* at a particular class (or module).
Efferent coupling measures the number of dependencies that leave (or *exit*) from a particular class (or module).

What are **Ca** and **Ce** for collaborators in the **Template Method Pattern?**

#doc220

Strategy

An alternative to the Template Method for varying a particular behaviour.

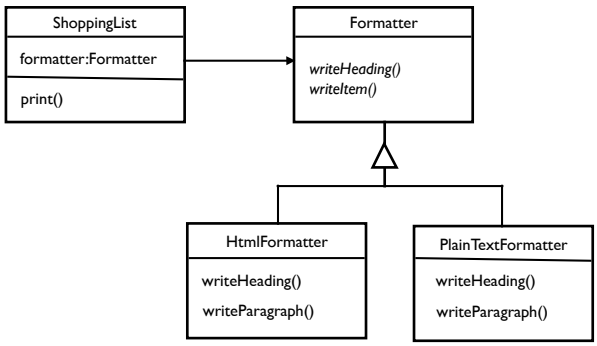
Delegate to a collaborator (rather than a subclass). Pull the algorithm into a separate object.

Separate things that change from things that stay the same

#doc220

An alternative way of dealing with this duplication, is rather than using inheritance, to use peers that collaborate. We again pull the code that calculates a particular term into a separate object, but just have a "has a" relationship between the sequence and it's term calculator. We can configure which variant we want at construction time, by passing the particular Strategy as a constructor parameter to the sequence object.

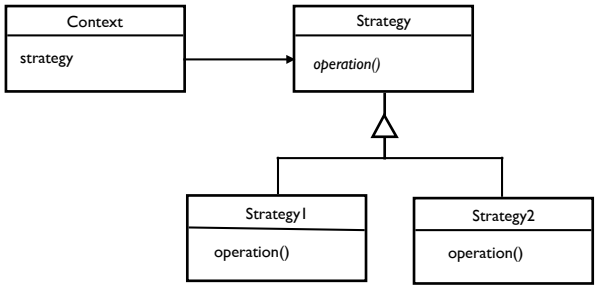
Strategy : Example



#doc220

We can see that we still have an inheritance tree, but this is just between the different strategies (in the example above for formatting a shopping list), not between different types of shopping list itself. This makes it much easier to, in future, apply these different types of formatter to other objects, such as Emails, Newsletters, Reports etc etc.

Strategy : pattern



Favour object composition over class inheritance

#doc220

The inheritance solution we saw with the Template Method pattern, keeps a fairly tight coupling between the sub and superclasses. We can use one without the other. Although we have removed some duplication, we don't have a set of independently reusable components.

The Strategy pattern gives us more flexibility here as we compose the objects together, which is a looser coupling than using inheritance.

What are **Ca** and **Ce** for collaborators in the **Strategy Pattern?**

#doc220

(In)Stability

I = Ce / (Ce+Ca)

More usefully applied to packages than to individual classes. Instable modules can be changed often; changing stable modules can have wide-ranging effects.

#doc220

One further metric that we can calculate from the two coupling metrics, is a module's Instability (or conversely, Stability). We want the core parts of our system to be stable - they should not be changed much. Parts at the edges, maybe to do with the user interface, or presenting data, might change much more often. These parts of the code should have lower stability (or rather, higher instability) measures.

Homework

Watch this video:

https://www.youtube.com/watch?v=8bZh5LMaSmE

#doc220

Once you have considered these ideas, and have completed the lab exercise for this week, watch this video of a conference talk by Sandi Metz. It gives some very good insights into ways of understanding unfamiliar code, and refactoring it to make change easier.