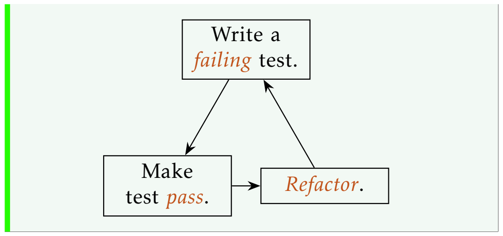
1 Test Driven Development

Waterfall development involves many phases of software development which are carried out sequentially. Design is an early phase, before coding and testing. Only one attempt is given to "get things right". Today, most teams use *agile*, iterative methods. **Cost of Change** describes how new features become more costly to implement as a project is developed (as it must work with all existing features). A good code design:

1. *Behaves Correctly* (passes tests).
2. *Minimises Duplication*.
3. *Maximises Clarity*.
4. *Has fewer elements*.

In **test driven development**, we work in a cycle: writing a test before the feature is implemented. Here, we are designing an API as we are writing tests.



Behaviour driven development means that a set of well written tests acts as an executable specification for the software, and documents its behaviour. Here, we think about *behaviour rather than implementation detail*, as in TDD. To design an object this way:

1. Write down *behavioural properties* of the object.
2. Create the actual *tests*.
3. *Implement* the code. Repeat.

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

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

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

2 Refactoring

Refactoring improves code's design without changing its behaviour. In TDD, we should only refactor in *green state* (we guarantee if we break anything its our fault). **Technical debt** is the implied cost of choosing an easy or quick solution now instead of using a better, more sustainable approach that would take longer.

3 Objects & Mock Objects

Inter-object communication can be thought of as *messages* instead of method calling. This is visualized in a *UML Sequence Diagram*. Methods fall into two categories: **commands** (no return) and **queries** (return).

An object graph contains **value objects** which usually contain data and little functionality, and interaction objects. When writing tests, we *focus on a single object*, whose collaborators have a particular responsibility which is seen by the caller object (*interfaces*). Inward messages to our test object are provided by the test, but outward messages must be *sensed* by a **mock object**. For example with JMock:

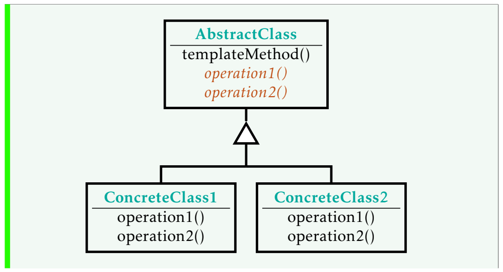
```
@Rule public JUnitRuleMockery context = new
    JUnitRuleMockery();
Chef pastryChef = context.mock(Chef.class);
HeadChef headChef = new HeadChef(pastryChef);

@Test
public void delegatesPuddingsToPastryChef() {
  // Expectation is set
  context.checking(new Expectations() {{
      exactly(1).of(pastryChef).order(PUDDING);
  }});

  // Expectation should be met
  headChef.order(CHICKEN, PUDDING);
}
```

We can also do:

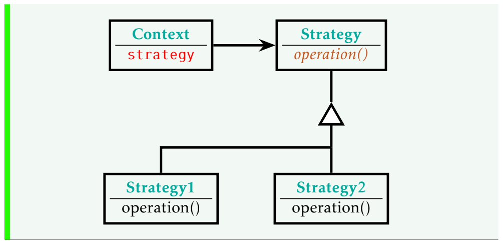
```
context.checking(new Expectations() {{
    // Any value
    oneOf(obj).method(with(any(Clazz.class)));

    // Specified Return
    allowing(summariser).summarise(a);
    will(returnValue(List.of(topic1, topic2)));

    // Counting
    oneOf(alice).alert(a);
    never(bob).alert(a);
}});
```

4 Reuse & Extensibility

Programming **patterns** are solutions to common problems - ways to do stuff in a language. The **template method pattern** means extracting a superclass to minimise duplicated code. *Modules should be open for extention but closed for modification* (Open-closed principle).



We aim for a low **coupling** between classes. **Afferent coupling** is how many other classes use it (*responsibility*). **Efferent coupling** is how many other classes it uses (*independence*). The **strategy pattern** delegates to a collaborator rather than subclass:



5 Designing for Flexibility

Stability depends on a balance of independence and responsibility. A bad design is:

- ***Rigid*** - hard to understand / change.
- ***Fragile*** - unstable.
- ***Immobile*** - hard to reuse the code.

Encapsulation reduces *fragility*, making code more stable; a method of **information hiding**. The **law of demeter** states:

- Each unit has limited knowledge about others.
- Each unit should only talk to its friends.
- Only talk to your immediate friends.1

6 Object Creation & Dependencies

Factory Methods create objects. Unlike constructors, they have a descriptive name. (e.g. `VirtualMachine.optimisedForHighCpu();`). A *private constructor* can force this. Additionally, these may delegate to runtime allowing for additional functionality.

Abstract Factory Pattern: we can define an interface for a factory method called a factory pattern. Useful when we want many factory patterns for similar objects.

Builders are separate objects that first gather parameters, then produce a valid object.

Singletons ensure a class only has ***one*** instance, and provide a global point of access to it. *This should be used when absolutely required!*

We can avoid unecessary dependencies by creating an interface. For example, if `Switch` activates `Light`, instead of referencing directly, we can pass in a `Device` into the consturctor of `Switch`. *Singletons create a dependency!*

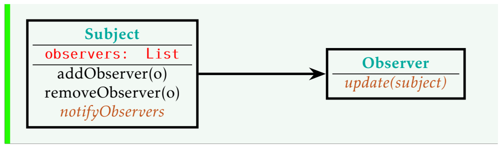
7 Code Metrics

We can use graphs or matrices to visualize dependencies. This is useful as formations such as *cyclic dependencies* lead to tight coupling and immobility of code. **Cyclomatic complexity** counts nodes and edges in CFG of a program, counting possible executions as a measure of complexity. In general, if it looks bad, *it is*.

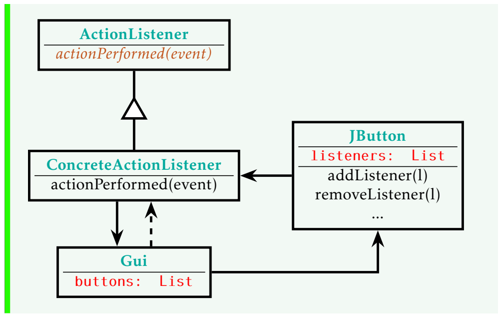
An **ABC metric** counts assignments, branches and conditions. We can use version control to see changes in metrics over time. We can also see turbulence of files: how often they get changed. Plotting turbulence against complexity can identify problem points. We can also look at which files were changed in the same commit.

8 Interactive Applications

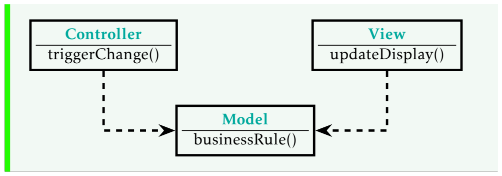
To give control to the user we use **events**. An **observer pattern** registers an object to be notified when an event occurs.



An example of this is:



The **model-view-controller** pattern splits apps into *data*, *display* and *user input* respectively:

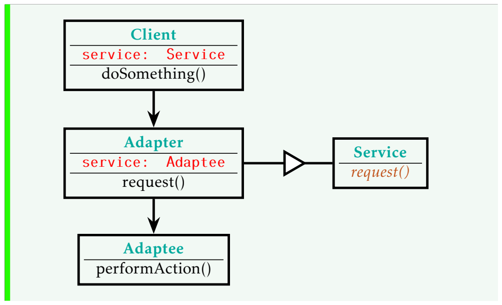


In this model:

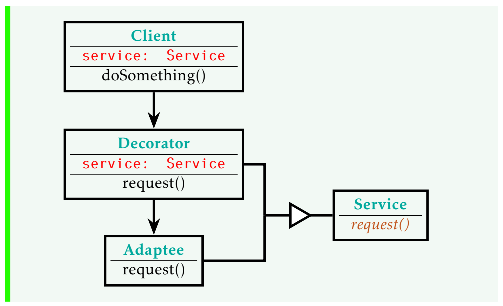
- **Strategy Pattern** - *view* uses *controller*, to which it delegates decisions and input handling. Controller the controls interaction with the *model*.
- **Composite Pattern** - *view* contains composite UI components. The *controller* updates the root, and this propogates through the tree.
- **Observer Pattern** - *model* is observable and *views* are observers.

9 Systems Integration

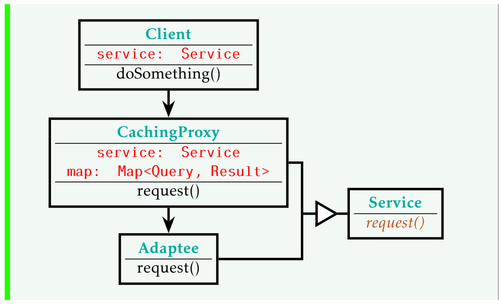
An **adapter pattern** is a wrapper around a class to change its interface to an expected one: delegation. A client requests a service, but despite implementing service the adapter has no behaviour of its own:



A **decorator** adds additional functionality:



A **facade** does the opposite, removes functionality. A **proxy** has the same structure as a decorator, but instead the functionality it adds is protection to the Adaptee. A **caching** pattern reduces latency:



10 Legacy Code

A **legacy** system is *important* software you have *inherited*. One of the first things to do is to make a graph of the code's dependencies (and their versions).

When changing the code, we want to preserve existing behaviour as much as possible. To give us confidence, we unit test at the micro level, and system test at the macro level.

- A **seam** is a place where you can alter behavior in your program without editing in that place. Every seam has an **enabling point**, a place where you can make the decision to use one behavior or another.
- We break dependencies to **sense** when we cant access values our code computes. This involves replacing a dependency at a seam with a test implementaiton.

11 Distribution & Remoting

We split an app into components to split up computation when running, and development into teams. In this case, it is common to have *client* components and *service* components. In recent years, apps have been using *microservice* archs, a large number of small services that cooperate to provide the full system.

To communicate, we usually use HTTP and REST, and pass data as XML or JSON.

The **Richardson Maturity Model** describes & categorises webservices based on how much they take advantage of URIs, HTTP and hypermedia.

0. A **Level 0** service uses HTTP, without using URIs to identify resources, different HTTP methods to describe actions or hypermedia. They normally use a single URI to identify the service *endpoint*, to which requests are posted. Each request consists of a document or set of parameters to describe the request, but go to the same URI by the same method. An example is **SOAP (Simple Object Access Protocol)**, which wraps a XML document describing a request.
1. A **Level 1** service makes more use of URIs, but does not take advantage of all HTTP methods. Here, verbs appear in URIs (e.g. *GET* `https://tennis-club.com/bookings/create?member=tim`).
2. A **Level 2** service uses different URIs and responds to different HTTP methods. They also send appropriate HTTP status codes.
3. A **Level 3** service is a fully RESTful service. It builds on level 2, but also contain hyperlinks to other resources a client can follow.

12 Agile

Agile is an alternative to *waterfall development*, an iterative method to revise and refine the design as new features are added. Work is done in short iterations, and changing plans and reprioritising requirements is common. We release the first, simple version as quickly as possible, and then improve it.

Continuous Integration keeps the master branch at a state where it could be deployed any time, by frequently merging into master. This is often automated with *automated builds*, that compile, run tests and package for release. This is often done on a dedicated **CI server**. Often, the builds themselves will be further tested in a production-like environments before they are shipped.