Object Oriented Software Engineering (COMP2003)

Lecture 6: Dependencies

Updated: 30th April, 2019

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

Dependencies

- Coupling/dependencies between classes affects:

  Maintainability: the more coupling, the harder it is to understand and modify code.

  Testability: the more coupling, the more complex the test cases, and the harder to interpret the results.
- Many design patterns are intended to *decouple* parts of a system.
- But coupling is not quite the same as links between objects.
  - We've seen that objects can be linked without their classes knowing about each other.
- We'll examine the decisions in code that leads to object creation.
  - How to link objects without coupling classes.

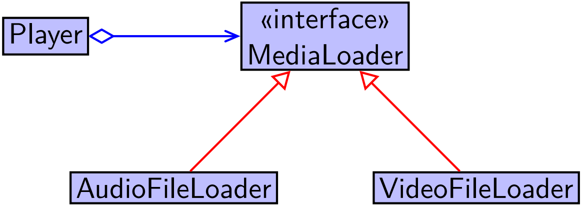
Factories

- Consider the Template Method, Strategy, Decorator and Composite patterns.
    - Several classes, with a common interface/superclass.
- Who decides *which* subclasses are used?
- One approach is to use a "factory":
    - A method, or an entire class, whose job is to make objects.
    - Not restricted to the above patterns.
- Factories are really the decision makers.
    - They choose between subclasses, so nobody else has to.

Factory Methods

- A method that creates and returns an object.
- It does this by calling the constructor (obviously).
- Why? Because this allows us to separate/decouple:
    1. The code that will use the new object, from
    2. The class being instantiated.
- If a class calls another's constructor, that's a hard-coded dependency.
- A factory method removes that hard-coding.
- (There is also a thing called the "Factory Method Pattern", which uses factory methods, but in a more complex way.)

Factory Method – Example



- The Strategy Pattern.
- A factory can decide which subclass to create – `AudioFileLoader` or `VideoFileLoader`.

Factory Method – Example

```
public static MediaLoader makeLoader(String file)
{                                             // Factory
    MediaLoader loader = null;
    if(file.endsWith(".mp3") || file.endsWith(".flac"))
    {
        loader = new AudioFileLoader();
    }
    else if(file.endsWith(".mp4") || file.endsWith(".avi"))
    {
        loader = new VideoFileLoader();
    }
    return loader;
}
```

- The system needs a `MediaLoader` object.
- The factory decides *which one* to make, and makes it.

A Warning About Silly Factories

```
public static MediaLoader sillyFactory(int which)
{
    MediaLoader loader = null;
    if(which == 1)
    {
        loader = new AudioFileLoader();
    }
    else if(which == 2)
    {
        loader = new VideoFileLoader();
    }
    return loader;
}
```

- To use this factory, you must *already know* what class you need.
  - Defeats the whole purpose.
  - The factory is supposed to make that decision.

Factory Methods – Location

- *Where* is a factory method located?
- Sometimes, in the superclass:

```
public abstract class MediaLoader
{
    public static MediaLoader makeLoader(...)
    {
        ... // Create and return a subclass object
    }
    ...
}
```

- A (sort-of) polymorphic constructor.
  - You call `MediaLoader.makeLoader()` to get a new instance.
  - But you don't know which subclass you'll get.
- Java will not allow this for interfaces, though. In that case, the factory method will need to be somewhere else.

Factory Classes

- Sometimes a factory method needs its own class.
  - (Because it doesn't really belong anywhere else.)
- Sometimes you might have several related factory methods.

```
public class MediaFactory
{
    public static MediaLoader makeLoader(...) { ... }
    public static MediaWriter makeWriter(...) { ... }
    public static MediaConverter makeConverter(...)
        { ... }
}
```

- `MediaLoader`, `MediaWriter` and `MediaConverter` are all abstract classes or interfaces.
- You may also create factory *objects*, with non-static factory methods. (Discussed later in the lecture.)

Factories and Testability

- Say we want to test this code:

```
public void play(String filename, Audio audio)
{
    MediaLoader loader;
    Media media;
    loader = MediaFactory.makeLoader(filename);
    media = loader.load(filename);
    audio.play(media);
}
```

- In testing, we need to control all the inputs.
  - Unit test code must examine specific, isolated situations.
  - Unit tests often use "mock" objects to do this. Here, we could pass in a mock instance of `Audio`.
- But we can't control the media loader.

Factories and Testability – Example

- We can design factories with testing in mind.

```
public class MediaFactory {
    private static MediaLoader testLoader = null;
    static void setTestLoader(MediaLoader tl) {
        testLoader = tl;
    }
    public static MediaLoader makeLoader(...) {
        if(testLoader == null) {
            /* --- Normal factory stuff --- */
            return ...;
        }
        else {
            return testLoader;
        }
    }
}
```

Factories and Testability – Discussion

In the previous example:

- As long as `testLoader == null`, `MediaFactory` works as before.
- However, test code can now force `makeLoader()` to return a different object:

```
@Test public void testSomething()
{
    MediaLoader mockLdr = mock(MediaLoader.class);
    MediaFactor.setTestLoader(mockLdr);
    ... // [Test something that uses MediaFactory]
    MediaFactory.setTestLoader(null);
}
```

- Between the two calls to `setTestLoader()`, `MediaFactory.makeLoader()` will return a special test object instead.

Singletons

- A class that restricts itself to having one instance.
- Typically used to control a resource, to avoid conflicts.
  - e.g. configuration files, log files, database connections.

```
public class MySingleton {
    private static MySingleton instance = null;

    public static MySingleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    private MySingleton() { /* Initialisation */ }
    // Other fields and methods
}
```

Singleton Use

- Once defined, you typically access a singleton from multiple different parts of the program:

```
public void method1() // ClassX
{
    MySingleton s = MySingleton.getInstance();
    ...
}
```

```
public void method2() // ClassY
{
    MySingleton s = MySingleton.getInstance();
    ...
}
```

- Both methods above will retrieve the same object.
- Whichever runs first will trigger the initialisation.

Global State

- Singletons use "global state".
  - Practically any non-constant static fields create "global state".
  - No possibility for a second copy of the same variable.
  - (FYI, a "global variable" is global state *without* any encapsulation.)
- Sometimes necessary.
  - e.g. In Android development, there's no way to directly pass parameter data from one "Activity" to another.
  - So, you can create a singleton, accessible by all activities.
- Often harmful!
  - Global state shares some of the same problems with global variables.

Singleton – Example

We might have a singleton to represent the configuration file:

```
public class Configuration {
    private static Configuration instance = null;
    public static Configuration getInstance() {
        if(instance == null)
            instance = new Configuration();
        return instance;
    }
    private Configuration() { /* Read config file */ }

    private String option1;
    private int option2;
    public String getOption1() { return option1; }
    public int getOption2()    { return option2; }
}
```

Singleton Structure

A singleton class (traditionally) has the following:

- A `private` constructor (and no other constructors).
  - Prevents any outside classes from creating an instance.
  - Also prevents any subclasses from being created at all.
- A `static` field whose type is the singleton class itself.
  - Typically called "`instance`".
  - Contains the one-and-only instance of the class.
- A `static` method whose return type is the singleton class.
  - Typically "`getInstance()`".
  - Allows the outside world to retrieve the instance.
  - Creates it if it does not already exist.
- Other fields and methods necessary for whatever the class does.

Singletons vs Static Fields/Methods (1)

- While singletons have a static field and method, they store all their important data in *object* fields.
- Why not just store everything in static fields to begin with, and get rid of the object altogether?
- Isn't this basically the same as having exactly one object?

Reason 1: Refactorability.

- In the future, you may discover you need more than one object.
- Singletons don't exactly help, but they're *closer* to what you want than a bundle of static fields/methods.
- You will still have to dismantle the singleton itself, but at least you're already using an object.

Singletons vs Static Fields/Methods (2)

Reason 2: "Lazy initialisation".

- This is a technique that can save processing resources.
- In lazy initialisation, we only initialise an object when it's about to be used:

```
if(instance == null)
{
    instance = new MySingleton();
}
```

- If the object is never used, then it is never created.
- Very awkward to do without having a proper object to create.
  - Not technically impossible, but we'd end up a very large number of `if` statements all over the place.

Singletons vs Static Fields/Methods (3)

Reason 3: Inheritance.

- A singleton may inherit from other classes and interfaces.
  - e.g., it could be an observer, receiving events from somewhere:

```
public class MySingleton implements MyObserver
{
    ...
    @Override
    public void update(MySubject s) {...}
}
```

- Static fields/methods can't achieve this, because they are not polymorphic (you can't override them).

The Singleton *Antipattern?* (1)

- Many people regard Singleton as (mostly) an antipattern.
  - An antipattern is like a pattern, but turns out to have more downsides than upsides. i.e., it's a trap!
- ✗ Singletons tend to break *testability*.
  - For a class that uses a traditional singleton, it's almost impossible to disentangle the two for unit testing.
- ✗ Singletons break dependency injection (more on that later).
- ✗ Singletons break serialisation.
  - (De-)serialisation needs to create new instances of a class.
- ✗ Singletons break design extensibility.
  - A common maintenance technique is to `extend` an existing class rather than modifying it.
  - Minimises the chances of introducing bugs.
  - Singletons cannot be extended, due to the private constructor.

The Singleton *Antipattern?* (2)

✗ Singletons can often resemble global variables.

- Singletons have accessors and mutators, but they only really protect the singleton itself, not *other* classes that use it:

```
public void method1() // ClassX
{
    MySingleton.getInstance().setVal("Hello world");
}
```

```
public void method2() // ClassY
{
    String v = MySingleton.getInstance().getVal();
    ...
}
```

- The above code shows a global variable "with extra steps".
- ClassX and ClassY are coupled to each other, via the singleton, and this coupling relationship will be difficult to see.

“Looser” Singletons

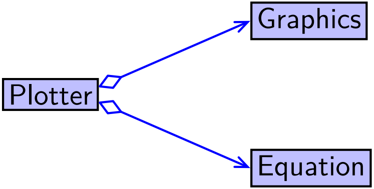
- Singletons may occasionally be necessary (e.g. in Android).
- There are ways to reduce the issues with them.
- Have a normal *public* constructor.
    - Keep the static `instance` field and `getInstance()` method.
    - Remove the restriction on creating more instances/subclasses.
    - Is *one-instance-only* important? Or do you just need *an* instance to be globally accessible?
- Have a package-private, static `setInstance()` method.
    - This would allow unit test code to replace the real singleton with a mock version.
    - Similar to what we did for factories.
- Avoid defining *multiple* separate singleton classes.
    - If needed, have a single singleton that *aggregates* other classes.
    - Other classes then don’t need to be singletons themselves.

Dependency Injection

- Intended to improve both maintainability and testability.
- A class should not hard-code its dependencies.
  - No direct object creation (“`new`” in Java).
  - No static method calls to other classes.
  - No raw objects (in C++).
  - References to any required objects should be *imported*.
    - Possibly by the constructor, and stored in fields.
    - Possibly by a method.
  - The constructor should be trivial(ish).
    - All significant code should be in methods.
  - Sound radical?
- A piece of code – the *injector* – connects objects together.
  - Creates the actual objects, and supplies the dependencies.
  - Possibly done in `main`, or close to `main`.

Dependency Injection – Example

- Consider a system to graph equations like $y = 2x + 3$.
    - Parse an equation entered by the user (as in DSA).
    - Iterate over values of x, calculating y for each x.
    - Display the points on the screen.
- In designing this system, we might have these classes:
    - `Equation` – to calculate values of y given x;
    - `Graphics` – to display points on the screen;
    - `Plotter` – to perform the algorithm.
- Say `Plotter` depends on the other two classes:



Naïve Implementation of Plotter

```
public class Plotter
{
    private Equation eq;
    private Graphics gfx;

    public Plotter()
    {
        eq = new Equation(); // Hard-coded dependency
        gfx = new Graphics();
    }
    public void plot() { /* algorithm */ }
}
```

- Coupled directly to `Equation` and `Graphics`.
- How do you test `Plotter` *by itself*?

Plotter with Dependency Injection

```
public class Plotter
{
    private Equation eq;
    private Graphics gfx;

    public Plotter(Equation inEq, Graphics inGfx)
    {
        eq = inEq;             // Injected dependency
        gfx = inGfx;
    }
    public void plot() { /* algorithm */ }
}
```

- Simple change – parameters instead of new.
- Now the test code can control eq and gfx.

The Injector

- Obviously, we must still create `Equation` and `Graphics` objects.
- This is done "out of the way" of `Plotter`'s logic.
- Below, `main()` is injecting dependencies into `Plotter`:

```
public static void main(String[] args)
{
    ...
    Equation eq = new Equation(...);
    Graphics gfx = new Graphics(...);

    Plotter plotter = new Plotter(eq, gfx);
    plotter.plot();
    ...
}
```

What about "new LinkedList()" and "Math.abs()"?

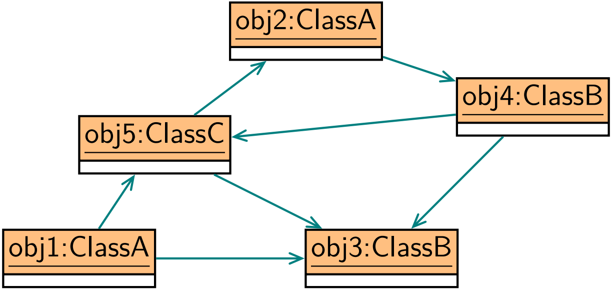
- Dependency Injection does not apply to standard API classes.
- This is fine:

```
public class ShoppingCart
{
    private List<Product> products;
    public ShoppingCart()
    {
        products = new LinkedList<Product>();
    }
    ...
}
```

- Standard classes like `LinkedList` and `Math` are not expected to have defects, nor expected to change.
- You can *depend* on them!

The Goal of Dependency Injection

- DI allows easy replacement of classes.
  - Replace any one without the others knowing.
  - Only the injector needs to be updated.
- A running system has many interconnected objects.
  - If the injector is responsible for the interconnections, nothing else has to be.



Testing Dependency-Injected Code – Example

```
@Test public void testPlotter()
{
    // Create special "mock" objects
    Equation eq = mock(Equation.class);
    Graphics gfx = mock(Graphics.class);

    when(eq. ...).thenReturn(...); // Define mock behaviour
    ...

    // Call the code you want to test
    Plotter plotter = new Plotter(eq, gfx);
    plotter.plot();

    verify(eq). ...(...); // Check what happened.
    ...
}
```

Testing Dependency-Injected Code – Discussion

- The test code on the previous slide relies on this critical line:

```
Plotter plotter = new Plotter(eq, gfx);
```

- Without this kind of constructor, it's pointless to mock `Equation` or `Graphics`.
- Dependency injection makes this kind of testing possible.

Other Things to Avoid

- Static method calls (hard-coded dependencies, like “new”):

```
public void plot()
{
    int x, y;
    ...
    Graphics.setPixel(x, y);
} // We should have a Graphics *object* here.
```

- Indirect dependencies (increases the amount of setting up required in test code):

```
public Plotter(Equation inEq, UserInterface ui)
{
    eq = inEq;
    gfx = ui.getGraphics();
} // We don't actually need UserInterface.
```

Factories and Dependency Injection

- In previous examples, our factories used static methods.
- But factories can be objects too:

```
public class MediaLoaderFactory()
{
    public MediaLoaderFactory() {}
    public MediaLoader makeLoader(String filename)
    {
        ...
    }
}
```

- The injector can pass the factory object to whoever needs it:

```
MediaLoaderFactory tlf = new MediaLoaderFactory();
Controller cont = new Controller(tlf, ...);
```

Injection Frameworks

- Take dependency injection to its ultimate end:
  - The (almost) complete elimination of new.
  - Impossible?
- 3rd-party frameworks actually allow this – like “Guice” for Java.
- Guice creates objects for you:
  - Using an auto-generated factory object, that is also an injector.
  - When you want an object, it works out what dependencies need to be created.
  - You give it some hints up-front on which subclasses to use.
- GUI construction takes a different approach.
  - You’ll find in worksheet 5 that GUIs are made of many, many interconnected objects.
  - You can create these manually, in source code.
  - OR you can load the entire object structure from a data file.
  - The data file can be created by a WYSIWYG GUI editor.