Worksheet 6: Dependencies

Updated: 7th October, 2014

1. Factories

Here are some factory-related conundrums for you to consider:

(a) Your program needs to read a file and create an object to represent the file's contents. You want to do this with a factory. There are many kinds of files you might need to read, and a different way to read each type.

You can usually use a file's extension (e.g. ".txt", ".png", ".so", etc.) to determine the file type, and thus how to read the file. However, sometimes the extension is incorrect. For instance, you can actually store a PNG image in a .txt file; the only thing stopping you is common sense.

Imagine that you don't want to rely on people's common sense. So, without checking filename extentions, how would you design the factory?

(b) You're writing a very old-fashioned video game, where the player controls a 2D spacecraft and must shoot asteroids and two kinds of aliens. These appear on the right-hand-side of the screen, randomly, and move left in some pre-determined fashion.

The game creates a new object every few seconds, and chooses its type at random. However, the probabilities are weighted depending on the difficulty, which increases as the game goes on. At first, the player faces only asteroids, then faces a combination of asteroids and type-1 aliens, then faces a combination of asteroids, and both types of aliens.

You have classes called `Asteroid`, `Alien1` and `Alien2` to represent the different obstacles the player faces. These all inherit from a common superclass called `SpaceObject`. How would you design a factory for creating `SpaceObject` objects?

(There are multiple solutions to this problem – see if you can think of more than one.)

2. Dependency Injection – Refactoring

Consider the following class:

```java
public class SecuritySystem implements SensorObserver
{
    private MotionSensor motionSensor;
    private HeatSensor heatSensor
    private Alarm alarm;
    private boolean armed;

    public SecuritySystem(Hardware hw)
    {
        SensorBundle sens = hw.getSensors();
        motionSensor = sens.getMotionSensor();
        heatSensor = sens.getHeatSensor();
        motionSensor.addSensorObserver(this);
        heatSensor.addSensorObserver(this);
        alarm = new Alarm();
        armed = false;
    }

    public void setArmed(boolean newArmed)
    {
        armed = newArmed;
        EmailSystem.sendMessage("Armed: " + newArmed);
    }

    @Override
    public void sensorDetection(Sensor s)
    {
        if(armed)
        {
            alarm.ring();
            EmailSystem.sendMessage("Sensor detection for " +
                                    s.toString());
        }
    }
}
```

(a) Refactor the `SecuritySystem` class to conform to the Dependency Injection pattern, without altering its basic responsibilities.

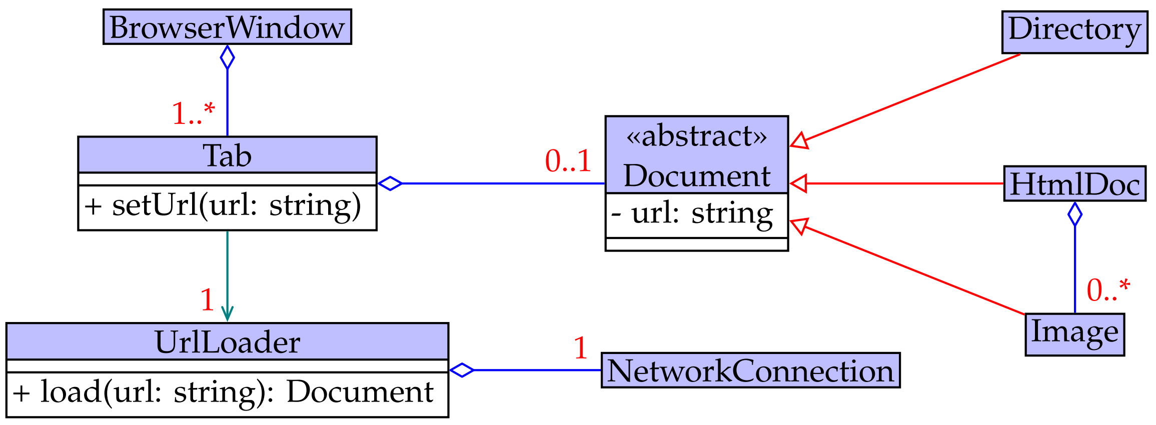
What other classes might need to be refactored as well?

(b) Provide an example of the injector code, showing how to create an instance of `SecuritySystem` from scratch.

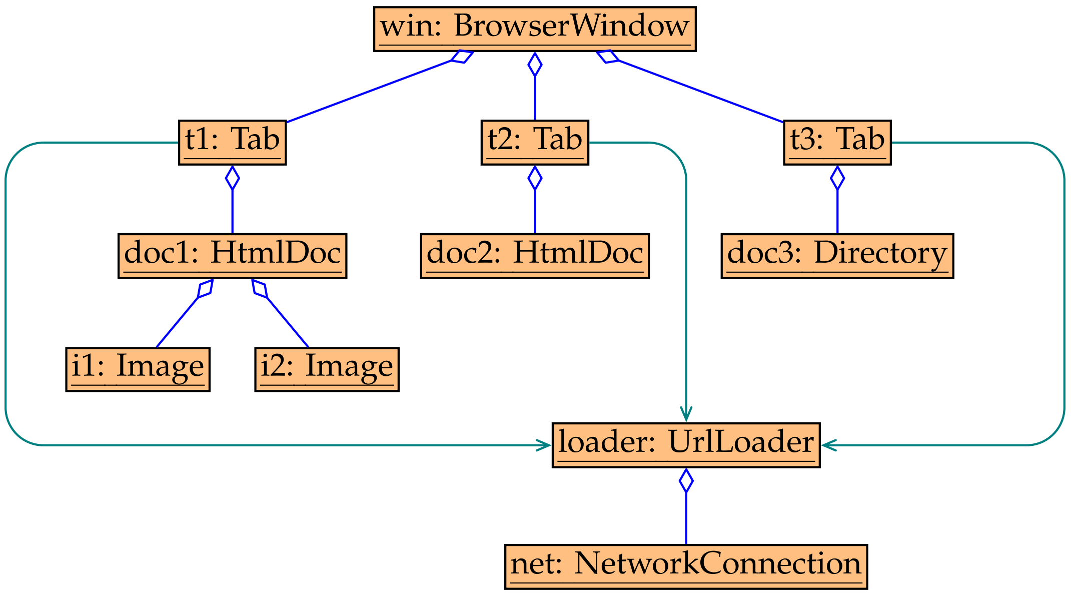
3. Dependency Injection – Design

Consider the following class and object diagrams, representing a web browser:

Class diagram



Object diagram



Based on the Dependency Injection pattern:

(a) What are *two ways* you could design the BrowserWindow class, such that a BrowserWindow object can have many Tab objects?

(b) What is the nature of the relationship between Tab, UrlLoader and Document?

(c) Write the simplest injector code needed to create the object structure shown above. (Don't worry about user input for this purpose.)

Assume that Tab's constructor does import a Document object.

End of Worksheet