Module 11
Design Patterns

CS W169A: Software Engineering

1 Overview

The first part of this worksheet is meant to introduce the reader to some practical code examples of some of the design patterns we've discussed in class. This tutorial is largely based off the Design Patterns in Ruby documentation. You are definitely encouraged to read about the other design patterns with code examples, but we will go over the patterns we deem more relevant to the class in this worksheet. The reference code is available here.

The second section of this worksheet will ask you to fill in some design pattern definitions and think about what kind of problems or contexts warrant what kinds of design patterns.

2 Design Pattern Examples

2.1 Observer

Let's consider an `Employee` object that has a `salary` property. We'd like to be able to change their salary and keep the payroll system informed about any modifications. The simplest way to achieve this is passing a reference to payroll and inform it whenever we modify the employee `salary`:

```
class Employee
  attr_reader :name, :title
  attr_reader :salary

  def initialize(name, title, salary, payroll)
    @name = name
    @title = title
    @salary = salary
    @payroll = payroll
  end

  def salary(new_salary)
    @salary = new_salary
    @payroll.update(self)
  end
end
```

2.2 Decorator

Here is an implementation of an object that simply writes a text line to a file.

At some point, we might need to print the line number before each one, or a timestamp or a checksum. We could achieve this by adding new methods to the class that performs exactly what we want, or by creating a new subclass for each use case. However, none of these solutions is optimal.

```
class SimpleWriter
  def initialize(path)
    @file = File.open(path, 'w')
  end

  def write_line(line)
    @file.print(line)
    @file.print("\n")
  end

  def close
    @file.close
  end
end
```

2.3 Factory

Imagine that you are asked to build a simulation of life in a pond that has plenty of ducks. But how would we model our `Pond` if we wanted to have frogs instead of ducks? In the implementation above, we are specifying in the `Pond`'s initializer that it should be filled up with ducks.

```
class Pond
  def initialize(number_ducks)
    @ducks = number_ducks.times.inject([]) do |ducks, i|
      ducks << Duck.new("Duck#{i}")
      ducks
    end
  end

  def simulate_one_day
    @ducks.each {|duck| duck.speak}
    @ducks.each {|duck| duck.eat}
    @ducks.each {|duck| duck.sleep}
  end
end
```

2.4 Singleton

Let's consider the implementation of a logger class. Logging is a feature used across the whole application, so it makes sense that there should only be a single instance of the logger.

```
class SimpleLogger
  attr_accessor :level
  ERROR,WARNING,INFO = 1,2,3

  def initialize
    @log = File.open("log.txt", "w")
```

```
    @level = WARNING
  end

  def error(msg)
    ..
  end

  def warning(msg)
    ..
  end

  def info(msg)
    ..
  end
end
```

3 Problems

3.1 Definitions

Fill in the definitions for the following design patterns. Recall that design patterns help us solve recurring software engineering design and architecture problems with flexible and reusable object-oriented software.

- Observer: The Observer pattern is appropriate when there exists a one-to-many relationship, and dependent objects should be notified when a single object is modified. This pattern is mainly used to implement distributed event handling systems.

- Decorator: Add functionality to an object without changing it. The general idea is that a decorator "wraps" the original class, such that additional, new methods will not alter the original class definition. This pattern provides flexible alternative to sub-classing.

- Factory: Abstract creation of family of objects. We create objects without exposing the creation logic to the user. In other words, how the code works is a black box to the client. All the client sees is a common interface that he/she/they can refer to using a common interface.

- Composite: Composites are best used when you'd like to represent a group of objects as the same type of object, i.e. a "composite" of multiple objects! The problems this design pattern solve are situations where a you'd like a client to ignore differences between compositions of objects and individual objects.

- Iterator: This pattern is motivated by the need to allow clients to access each item in a collection without exposing details of the container.

- Visitor: This pattern applies type-specific operations to elements in a container without changing the objects' code. In other words, if you'd like the behavior specific to different kinds of clients in the same application, the visitor pattern serves as an intermediary that handles this exchange.

- Singleton: As evident in the name, a singleton pattern is where only a single instance of a single class is created. The class allows one to access its only object, but does not allow instantiation of multiple instances of the class. This design pattern would be useful for resources that are shared by an entire application (i.e. logging, caches, configuration settings).

3.2 Name that Pattern

For each problem below, you're presented with a situation or system that we'd like to implement. Decide which design pattern is best suited to or most reflective of each context or behavior and justify why.

- Armando's theater management site wants to send emails to every patron announcing the new season. However, donors (a type of patron) should get a special email welcoming them to the opening gala.
  Visitor: Emails should be specific to the type of patron. Therefore, we use the visitor pattern to tailor our responses to whichever guest we're interacting with.
  Iterator: We want to send emails to the exhaustive list of all patrons. Therefore, our application should implement an iterator pattern that abstracts away the behavior of traversing the list of all theatergoers.

- Sometimes a theater has to reschedule a show. When that occurs, all tickets should be updated to reflect the new show time.
  Observer: The main concept here we'd like you to notice is that one entity's accuracy relies on another entity's state. In this case, the tickets' displays are dependent upon what the show time is. Therefore, the ticket should be an observer of the show's state, and updates automatically upon changes to the show.

- The site allows patrons to purchase tickets through Visa. For security reasons, however, the site can only maintain one connection to Visa's purchasing service.
  The "one connection" requirement should be a conspicuous giveaway that we're looking for the behavior provided by the Singleton design pattern. All transactions processed by the entire application use this one shared resource, so the corresponding connection class should create and manage one instance of itself.

- Sometimes people buy a single ticket, but most of the time people buy tickets in pairs, or for the whole family. The theater wants to be able to print these tickets in the same way.
  This one's a bit tricky. The main behavior we'd like to capture is that printing tickets works the same way no matter whether a single ticket or a pair of tickets are being bought. The Composite behavior will allow us to ensure any number of tickets will be treated the same way in the eyes of the printing functionality.

- The theater recently decided that VIP ticket should, in addition to showing the show name, now also show the director's name to make it more informative for the audience.
  The VIP ticket class already has an existing set of functionality, and we'd like to augment it with this new feature without affecting the original implementation. In this case, we'd be wise to make use of a Decorator to make this happen.

- The theater used to write each ticket on a post-it note and give that to the recipient. Having decided that their employees have better things to do than write on post-its, the theater bought a printer for tickets to hide the complexity of actually printing tickets.
  The desired behavior here is a bit subtle. The main idea here is that we'd like to hide the complexity of creating objects of a class. The Factory pattern is defined to abstract creation of families of objects, which makes it the most appropriate choice here.