SWEN20003
Object Oriented Software Development
Workshop 9

Eleanor McMurtry

Semester 2, 2020

Workshop

This week, we are reviewing **design patterns**.

1. Design patterns give you a recipe for solving common problems.
2. They are designed for a specific situation (the **motivation**), come with a specific **structure** (typically a UML diagram), and have **consequences**.
3. The **singleton** design pattern is used to ensure there is a single global instance of an object. Like global variables in C, it is usually best to avoid this pattern.
4. The **factory method** design pattern allows more flexible control over which specific instances are created e.g. of an abstract parent class or interface.
5. The **template method** design pattern allows an implementation of an algorithm to vary in its details using **inheritance**.
6. The **strategy** design pattern allows an implementation of an algorithm to vary in its details using **delegation**.
7. The **observer** design pattern decouples class interactions by notifying an observer when a subject's state changes.

Questions

1. Despite many people telling you it's a bad idea, you have decided to write your own encryption algorithm. You have decided to use the **factory method** design pattern to do this.

   (a) Create a class `SecretKey` with a constructor `SecretKey(int n)` and an attribute `byte[] key`, initialised with `n` random values. Write a method `byte[] encrypt(byte[] message)` implementing the following algorithm. ("xor" means bitwise exclusive-or, represented by ^ in Java.)

```
$m \leftarrow message.length$
$k \leftarrow key.length$
$j \leftarrow 0$
for $i = 0$ to $m$ do
    $result[i] \leftarrow message[i]$ xor $key[j]$
    $j \leftarrow j + 1$
    if $j = k$ then
        $j \leftarrow 0$
return $result$
```

   (b) Create a method `String decrypt(byte[] message)` that does the same thing, but uses

   ```
   new String(bytes, StandardCharsets.UTF_8)
   ```

   to convert the result back to a `String`. (For this algorithm, encryption and decryption are the same thing!)

(c) Make `SecretKey` abstract, and create two subclasses `ShortSecretKey` and `LongSecretKey` that create keys of length 16 and 32 respectively.

(d) Create an abstract `Encryptor` class with a `SecretKey` attribute and two methods:

- `byte[] encrypt(List<String> lines)` which should take a list of strings, encrypt each string, and concatenate the encrypted arrays.
- `abstract SecretKey createKey()`

(e) Create two subclasses: `InsecureEncryptor` that creates a `ShortSecretKey`, and `SecureEncryptor` that creates a `LongSecretKey`. [1]

2. We are opening a bar to profit off the influx of people going out when the lockdown ends. People who attend the bar can be relatives of the owners, members of the bar club, or just a regular customer. Relatives get a 99% discount, members get a 10% discount, and regular customers pay full price.

   Using the **strategy pattern**, design a simple system to handle these discount variations. Once you've done this, implement a simple bar simulation in your code. The bar class should contain a method that accepts customer and drink names as an argument, and returns the appropriate price.

   Use a `Map<String, Double>` to store the drink names and their costs. Use a `Map<String, DiscountStrategy>` to decide which strategy to use for a given customer name. You can hard-code entries for these maps. Test out your discounting system with different names.

3. Copy your solution to Question 2. This time, using the **template method pattern**, use an abstract method to create the drink map. Create subclasses `CasualBar` and `ExclusiveBar` with different menus.

4. To see the **singleton pattern** in practice, read `https://gitlab.eng.unimelb.edu.au/obrienad/bagel-public/blob/master/src/main/java/bagel/Window.java`. Why is this a good use for a singleton?

[1] It goes without saying that neither of these classes are actually secure!