SWEN20003
Object Oriented Software Development
Workshop 9 (Solutions)

Eleanor McMurtry

Semester 2, 2020

Workshop

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.)

```
1: $m \leftarrow message.length$
2: $k \leftarrow key.length$
3: $j \leftarrow 0$
4: for $i = 0$ to $m$ do
5:     $result[i] \leftarrow message[i]$ xor $key[j]$
6:     $j \leftarrow j + 1$
7:     if $j = k$ then
8:         $j \leftarrow 0$
9: 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.

   Solution:

```java
public abstract class SecretKey {
    private final byte[] key;

    public SecretKey(int n) {
        key = new byte[n];
        Random rand = new Random();
        rand.nextBytes(key);
    }

    public byte[] encrypt(byte[] message) {
        // Uses the modulo operation instead to make it neater
        byte[] result = new byte[message.length];
        for (int i = 0; i < message.length; ++i) {
            result[i] = (byte) (message[i] ^ key[i % key.length]);
        }
```

```
            return result;
        }

        public String decrypt(byte[] message) {
            return new String(encrypt(message), StandardCharsets.UTF_8);
        }
    }

    public class ShortSecretKey extends SecretKey {
        public ShortSecretKey() {
            super(16);
        }
    }

    public class LongSecretKey extends SecretKey {
        public LongSecretKey() {
            super(32);
        }
    }
```

(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]

Solution:

```
abstract class Encryptor {
    private final SecretKey key;

    public Encryptor() {
        key = createKey();
    }

    public byte[] encrypt(List<String> lines) {
        int length = 0;
        for (String str : lines) {
            length += str.length();
        }

        byte[] result = new byte[length];
        int i = 0;
        for (String str : lines) {
            byte[] ct = key.encrypt(str.getBytes());
            for (byte b : ct) {
                result[i++] = b;
            }
        }
        return result;
    }

    public abstract SecretKey createKey();
}

class InsecureEncryptor extends Encryptor {
    @Override
    public SecretKey createKey() {
        return new ShortSecretKey();
```

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

```java
    }
}

class SecureEncryptor extends Encryptor {
    @Override
    public SecretKey createKey() {
        return new LongSecretKey();
    }
}
```

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 using an interface `DiscountStrategy`. 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. (If there is no such drink, return 0.)

   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.

   Solution:

```java
public interface DiscountStrategy {
    double applyDiscount(double price);
}

public class MemberDiscountStrategy implements DiscountStrategy {
    @Override
    public double applyDiscount(double price) {
        return price * 0.90;
    }
}

public class RelativeDiscountStrategy implements DiscountStrategy {
    @Override
    public double applyDiscount(double price) {
        return price * 0.01;
    }
}

public class DefaultDiscountStrategy implements DiscountStrategy {
    @Override
    public double applyDiscount(double price) {
        return price;
    }
}

public class Bar {
    private final Map<String, Double> menu = new HashMap<>();
    private final Map<String, DiscountStrategy> people = new HashMap<>();

    public Bar() {
        menu.put("Orange juice", 1.00);
        menu.put("Water", 0.00);
        menu.put("Lemon squash", 2.00);

        people.put("Alice", new RelativeDiscountStrategy());
        people.put("Bob", new MemberDiscountStrategy());
    }
    public double getPrice(String customer, String drink) {
        if (menu.containsKey(drink)) {
            return people.getOrDefault(customer, new DefaultDiscountStrategy())
                         .applyDiscount(menu.get(drink));
        } else {
            return 0;
        }
    }
}
```

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.

   Solution:

```
public abstract class Bar {
    private final Map<String, Double> menu;
    private final Map<String, DiscountStrategy> people = new HashMap<>();

    public Bar() {
        menu = getMenu();

        people.put("Alice", new RelativeDiscountStrategy());
        people.put("Bob", new MemberDiscountStrategy());
    }

    public abstract Map<String, Double> getMenu();

    public double getPrice(String customer, String drink) {
        if (menu.containsKey(drink)) {
            return people.getOrDefault(customer, new DefaultDiscountStrategy())
                         .applyDiscount(menu.get(drink));
        } else {
            return 0;
        }
    }
}

public class CasualBar extends Bar {
    @Override
    public Map<String, Double> getMenu() {
        Map<String, Double> menu = new HashMap<>();
        menu.put("Orange juice", 1.00);
        menu.put("Water", 0.00);
        menu.put("Lemon squash", 2.00);
        return menu;
    }
}

public class ExclusiveBar extends Bar {
    @Override
    public Map<String, Double> getMenu() {
        Map<String, Double> menu = new HashMap<>();
        menu.put("Rosewater", 9.00);
        menu.put("Water", 0.00);
        menu.put("Affogato", 11.00);
        return menu;
    }
}
```