Design patterns lab 1

Generalities

Def1: Reusable solutions to commonly occurring problems.

Def2: OO design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

Each pattern has:

- name
- problem – when to apply the problem
- solution – elements that make up the design, relationships, responsibilities and collaborations
- consequences – results and trade -offs of applying the pattern

Reference:

For the first lab read descriptions for **Factory**, **Factory Method**, **Builder** and **Decorator**

https://refactoring.guru/design-patterns/factory-method

https://refactoring.guru/design-patterns/builder

https://refactoring.guru/design-patterns/decorator

https://www.oodesign.com/

https://medium.com/@ronnieschaniel/object-oriented-design-patterns-explained-using-practical-examples-84807445b092

Additional practical usages (from Effective Java)

Static Factory Method

-use it instead of a constructor, eg.

public static Boolean valueOf(boolean b) {

return b ? Boolean.TRUE : Boolean.FALSE;

}

Advantages

-they have names. eg. BigInteger(int, int, Random), which returns a BigInteger that is probably prime, would have been better expressed as a static factory method named BigInteger.probablePrime.

-unlike constructors, they are not required to create a new object each time they're invoked. This allows immutable classes to use preconstructed instances, or to cache instances as they're constructed, and dispense them repeatedly to avoid creating unnecessary duplicate objects.

-unlike constructors, they can return an object of any subtype of their return type.

-the class of the returned object can vary from call to call as a function of the input parameters.

-the class of the returned object need not exist when the class containing the method is written.

Examples from the library:

```
List<Integer> list = Arrays.asList(9,7,5,3);
List<Complaint> litany = Collections.list(legacyLitany);
Date d = Date.from(instant);
Instant now = Instant.now();
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
LocalTime time2 = LocalTime.of(6, 15, 30);
BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
```

Exercise 1

Interface MyList (with methods add (adds at the end of the list) and get(int index) is implemented by three classes MyArrayList, MyLinkedList and MySynchronizedList. MyArrayList is backed by an array, MyLinkedList by a linkedlist and MySynchronizedList has synchronized add and remove methods. Interface MyList has a static method getList(enum ListType) which returns the appropriate implementation. For the exercise work only with integers.

```
MyList arrayList =MyList.getList(ListType.Array);
arrayList.add(5);
System.out.print(arrayList.get(0));
MyList linkedList =MyList.getList(ListType.LinkedList);
linkedList.add(7);
System.out.print(linkedList.get(0));
MyList syncList =MyList.getList(ListType.SyncList);
syncList.add(9);
System.out.print(syncList.get(0));
```

End of exercise 1

Builder (from Effective Java)

Static factories and constructors share a limitation: they do not scale well to large numbers of optional parameters. Consider the case of a class representing the Nutrition Facts label that appears on packaged foods. These labels have a few required fields—serving size, servings per container, and calories per serving— and more than twenty optional fields—total fat, saturated fat, trans fat, cholesterol, sodium, and so on. Most products have nonzero values for only a few of these optional fields.

What sort of constructors or static factories should you write for such a class?

Traditionally, programmers have used the telescoping constructor pattern, in which you provide a constructor with only the required parameters, another with a single optional parameter, a third with two optional parameters, and so on, culminating in a constructor with all the optional parameters.

```
public class NutritionFacts {
private final int servingSize; // (mL) required
private final int servings; // (per container) required
```

```
private final int calories; // (per serving) optional
private final int fat; // (g/serving) optional
private final int sodium; // (mg/serving) optional
private final int carbohydrate; // (g/serving) optional
public NutritionFacts(int servingSize, int servings) {
this(servingSize,servings, 0);
}
public NutritionFacts(int servingSize, int servings, int calories) {
this(servingSize, servings, calories, 0);
}
public NutritionFacts(int servingSize, int servings, int calories, int fat) {
this(servingSize, servings, calories, fat, 0);
}
public NutritionFacts(int servingSize, int servings, int calories, int fat, int
sodium) {
this(servingSize, servings, calories, fat, sodium, 0);
}
public NutritionFacts(int servingSize, int servings,
int calories, int fat, int sodium, int carbohydrate) {
this.servingSize= servingSize;
this.servings= servings;
this.calories= calories;
this.fat= fat;
this.sodium= sodium;
this.carbohydrate= carbohydrate;
}
}
```

When you want to create an instance, you use the constructor with the shortest parameter list containing all the parameters you want to set:

```
NutritionFacts cocaCola =new NutritionFacts(240, 8, 100, 0, 35, 27);
```

the telescoping constructor pattern works, but it is hard to write client code when there are many parameters, and harder still to read it. The reader is left wondering what all those values mean and must carefully count parameters to find out. Long sequences of identically typed parameters can cause subtle bugs. If the client accidentally reverses two such parameters, the compiler won't complain, but the program will misbehave at runtime.

An alternative is the JavaBeans pattern, in which you call a parameterless constructor to create the object and then call setter methods to set each required parameter and each optional parameter of interest.

```
NutritionFacts
cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

Disadvantage

-a JavaBean may be in an inconsistent state partway through its construction.

-JavaBeans pattern precludes the possibility of making a class immutable.

Builder pattern:

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)
.calories(100).sodium(35).carbohydrate(27).build();
```

(optional parameters are initialized to default values).

Exercise 2

A car has the following features:

-brand (required)

-production year(required)

-engine power(required)

-fuel type(required)

-chassis number(required)

-sound system (optional, default is Sound.RadioCD)

-navigation (optional, default is Navigation.None)

-air conditioning (optional, default is Air.MANUAL)

Use a Builder pattern so that different combinations of parameters are used to construct car objects. The object has to be immutable.

eg.

```
Car fordTrend=new Car.Builder(“Ford”,2009,87,”diesel”,”XYZ”).build();
Car fordTitanium=new
Car.Builder(“Ford”,2018,125,”diesel”,”WWW”).sound(Sound.MP3).navigation(Navigation.SM
ALL).build();
Car fordEco = new Car.Builder(“Ford”,2019,100,”gas”,”YHD”).air(Air.AUTO).build();
```

End of exercise 2

Exercise 3

A decorator can be used to add additional behavior to objects at runtime. Reuse the MyList interface and classes from the first exercise. Implement a LoggingDecorator so that each time add method is called a message is displayed on the console detailing what element was added.

```
LoggingDecorator loggedList = new LoggingDecorator(MyList.getList(ListType.Array);
loggedList.add(5);
```

// Default output displays “5 was added to the list”