CS613 Assignment 1

Deirdre Hegarty

December 2018

Contents

Directory Structure

```
Submitted_Folder

   Assignment

       bin
           behaviours
           factory
           strategy
           subclasses
       docs
           behaviours
           factory
           jquery
               external
                   jquery
               images
               jszip
                   dist
               jszip-utils
                    dist
           resources
           strategy
           subclasses
       src
            behaviours
            factory
            strategy
            subclasses

   Code_Evolution
        1_Strategy_pattern
            bin
            src
        2_Factory_Method
            bin
            src
        3_Abstract_Factory
            bin
            src
        4_Singleton
             bin
             src
```

`Assignment` contains version 1.0 of my project. This folder contains 3 subfolders: * src : All source-code(.java files). * docs : Documentation generated from javadoc - to view open `index.html` in browser. * bin : All byte-code (.class files).

`Code_Evolution` contains folders that reflect how the project evolved over time; building each design pattern on top of the previous.

To Run (UNIX)

```
# navigate to src folder
cd src/

# compile
javac -d ../bin $(find ./* | grep .java)

# run
java -cp ../bin Runner

# generate javadoc
javadoc -d docs/ $(find ./src/ -name *.java)
```

How it Works

There are 4 different collections: `behaviours`, `factory`, `strategy` and `subclasses`. `behaviours` contains interfaces for behaviours in Chromosomes and Populations, which are located in the `strategy` collection. The `factory` collection contains code that utilises factory method and abstract factory pattern. `subclasses` conatins subclasses of `behaviours` and `strategy`.

complete output

```
$ javac -d ../bin $(find ./* | grep .java)
$ java -cp ../bin Runner

New Chromosome created
New Individual
New Chromosome created
New Individual
****************** Factory Method ****************************
Inside abstract Chrome_Factory
Inside IndividualGetter
New Chromosome created
New Individual
1111111111111111
1111111110111111
****************** Abstract Factory ****************************
New Chromosome created
New Individual
1101100100110110
I am NOT a mutant
1101000100110110
New Chromosome created
New Individual
New Chromosome created
New Individual
Offspring 1 : 1101111000011110
```

```
Offspring 1 : 1101100100110110
******************** Abstract Factory Singleton ****************************
LAZY INITIALIZATION WITH DOUBLE CHECK LOCKING
SINGLETON CONSTRUCTOR
New Chromosome created
New Individual
1101100100110110
1101111100110110
```

output breakdown

```

New Chromosome created
New Individual
New Chromosome created
New Individual
```

This demonstrates that Individual is a subclass fo Chromosome. Chromosome is an abstract class, and the base of a strategy pattern (Eric Freeman, n.d.a). Inside Chromosome behaviours such as mutate and crossover can be specified.

```
ChromeFactory factoryMethod = new IndividualGetter();
Chromosome factoryChromosome = factoryMethod.getChrome("111111111111111");
System.out.println(factoryChromosome.getName());
factoryChromosome.mutate();
System.out.println(factoryChromosome.getName());
```

```
Inside abstract Chrome_Factory
Inside IndividualGetter
New Chromosome created
New Individual
1111111111111111
1111111110111111
```

This demonstrates the use of a simple factory (Eric Freeman, n.d.b). IndividualGetter is a subclass of the abstract class ChromeFactory. Inside ChromeFactory is an abstract method `getChromeSubclass` that is accessable through `getChrome`.

When an Individual is retrived using IndividualGetter, name (encoding) is the same as the arguement passes by getChrome. When mutate is called, the name of the Individual object is modified and saved back to the object.

```
// Abstract factory
BigGAFactory bigFactory = new BigGAFactory();
Mutate m = bigFactory.createMutatorGA(checkMutant.NOTMUTANT);
Chromosome bigFChrome = new Individual("1101100100110110");
System.out.println(bigFChrome.getName());
bigFChrome.setMutation(m);
bigFChrome.mutate();
```

```
m = bigFactory.createMutatorGA(checkMutant.ISMUTANT);
bigFChrome.setMutation(m);
bigFChrome.mutate();
System.out.println(bigFChrome.getName());

Chromosome [] i = {chromosome1, chromosome2};
Population pop = new Population(i);
Selection s = bigFactory.createSelectGA(checkFitness.BIGGEST);
pop.setSelection(s);
Chromosome [] selected = pop.select();
for (int j = 0; j<2; j++){
    System.out.println("POPULATION SELECTED ["+j+"]"+selected[j].getName());
}


Couple c1 = new Couple(chromosome1, chromosome2);
c1.crossover();
```

```
New Chromosome created
New Individual
1101100100110110
I am NOT a mutant
1101100100100110
POPULATION SELECTED [0]1101111000011110
POPULATION SELECTED [1]1101100100110110
New Chromosome created
New Individual
New Chromosome created
New Individual
Offspring 1 : 1101111000011110
Offspring 1 : 1101100100110110
```

This demonstrates abstract factory pattern (Eric Freeman, n.d.c).

I could have made BigGAFactory into a singleton, but did not for the purposes of demonstrating different design patterns. Inside BigGAFactory behaviours for mutation and selection can be decided and saved into their respective Mutate and Selection type variables. These variables can then be passed the strategy pattern(s), where they are set to the behaviours of their respective object(s).

Population is the base of a second implementation of a strategy pattern. Inside it, selection behaviours can be set and changed at runtime. Couple is composed of two Individuals, and can crossover to ouput 2 offspring. Couple returns an array of 2 Individuals.

```
RandomGAFactory randomFactorySingleton = RandomGAFactory.createRGF();
// RandomGAFactory g2 = RandomGAFactory.createRGF();
// RandomGAFactory g3 = RandomGAFactory.createRGF();
Mutate m1 = randomFactorySingleton.createMutatorGA(checkMutant.ISMUTANT);
Chromosome randFChrome = new Individual("1101100100110110");
System.out.println(randFChrome.getName());
randFChrome.setMutation(m1);
```

```
randFChrome.mutate();
System.out.println(randFChrome.getName());
```

```
LAZY INITIALIZATION WITH DOUBLE CHECK LOCKING
SINGLETON CONSTRUCTOR
New Chromosome created
New Individual
1101100100110110
1101111100110110
```

This demonstrates abstract factory as a singleton.

(Geeks, n.d.)

Lazy initialization with Double check locking: In this mechanism, we overcome the overhead problem of synchronized code. In this method, getInstance is not synchronized but the block which creates instance is synchronized so that minimum number of threads have to wait and that's only for first time.

Pros:

- Lazy initialization is possible.
- It is also thread safe.
- Performance reduced because of synchronized keyword is overcome.

Cons:

- First time, it can affect performance.

References

Eric Freeman, Elisabeth Robson. n.d.a. In *Head First Design Patterns*, by Eric FreemanElisabeth Robson, pp1–24. O'Reilly.

———. n.d.b. In *Head First Design Patterns*, by Eric FreemanElisabeth Robson, pp1199–134. O'Reilly.

———. n.d.c. In *Head First Design Patterns*, by Eric FreemanElisabeth Robson, pp158–163. O'Reilly.

Geeks, Geeks for. n.d. "Java Singleton Design Pattern Practices with Examples." https://www.geeksforgeeks.org/java-singleton-design-pattern-practices-examples/.