Assignment 3

Generic Rubric

- 50% task completion: all requirements implemented
- 20% code correctness: code is correct, demonstrated with unit tests or user tests
- 20% style: code is well-organized, follows coding conventions, and is easy to read
- 10% presentation: code is adequately documented, UML class diagrams are formatted well

Singleton (50 points)

Implement 2 singleton classes, `EagerSingleton` and `LazySingleton`. Each of these singletons will create a maximum of 3 instances (instead of 1 instance as in the original pattern). The `EagerSingleton` creates all 3 instances immediately on program startup, whereas the `LazySingleton` creates the instances only when needed. As an additional variation, we change the signature of `getInstance` to take an `int` argument that indicates which of the 3 instances we want to access. The singletons do not have to be thread-safe.

Below is a Java code snippet with client code that uses your Singleton implementations. The client sleeps for 2s on every iteration in order to better observe how the eager and lazy singleton implementations differ in instantiating the singleton. Your implementation must work with this client code.

```
import java.util.Random;

public class Assignment3Singleton {
  public static void main(String[] args) {
    Random r = new Random();
    for (int i = 0; i < 10; i++) {
      EagerSingleton s = EagerSingleton.getInstance(r.nextInt(3));
      System.out.println("Retrieved eager singleton " + s.getId());
      try {
        Thread.sleep(2000);
      } catch (InterruptedException e) {}
    }
    for (int i = 0; i < 10; i++) {
      LazySingleton s = LazySingleton.getInstance(r.nextInt(3));
      System.out.println("Retrieved lazy singleton " + s.getId());
      try {
        Thread.sleep(2000);
      } catch (InterruptedException e) {}
    }
  }
}
```

An expected example output is provided below:

```
EagerSingleton 0 instantiated
EagerSingleton 1 instantiated
EagerSingleton 2 instantiated
Retrieved eager singleton 1
Retrieved eager singleton 2
Retrieved eager singleton 2
Retrieved eager singleton 1
Retrieved eager singleton 0
Retrieved eager singleton 0
Retrieved eager singleton 1
Retrieved eager singleton 2
Retrieved eager singleton 2
Retrieved eager singleton 0
LazySingleton 0 instantiated
Retrieved lazy singleton 0
Retrieved lazy singleton 0
LazySingleton 1 instantiated
Retrieved lazy singleton 1
Retrieved lazy singleton 0
Retrieved lazy singleton 0
Retrieved lazy singleton 0
Retrieved lazy singleton 1
LazySingleton 2 instantiated
Retrieved lazy singleton 2
Retrieved lazy singleton 1
Retrieved lazy singleton 2
```

Assignment steps:

1. Copy the client code `Assignment3Singleton` into your Java project
2. Implement the `EagerSingleton` class
3. Implement the `LazySingleton` class

Factory Method and Abstract Factory (50 points)

You are working at a software company that develops a presentation software for Natural History Museums. The museums want to organize their fossil exhibits consistently by era (e.g., Jurassic, Triassic, Cenozoic). In each era, there are exhibits of land, sea, or sky animal fossils: use interfaces/abstract classes and concrete classes to categorize the animal fossils (e.g., a `Pterodactylus` is a Jurassic sky animal, a `Plesiosaurus` is a Jurassic sea animal, a `Mammoth` is a Cenozoic land animal). `Animal`s have a `name` (e.g., `Mammoth`) and a value indicating how they walk (e.g., `stomp`, `run`, use your imagination), swim, or fly.

Implement the abstract factory pattern to easily switch out exhibitions in a museum. Every factory knows its own `era` and provides methods to create the exhibition's land (`createLandAnimals`), sea (`createSeaAnimals`), and sky animals (`createSkyAnimals`). Below is a museum implementation that uses the abstract factory. Your implementation of the abstract factory design pattern must work with this museum implementation.

```
import java.util.List;

public class NaturalHistoryMuseum {
  public static void main(String[] args) {
    // set up the exhibit
    AnimalAbstractFactory f = new CenozoicAnimalFactory();
    System.out.println("You are in the " + f.getEra() + " exhibition");
    System.out.println("===== " + f.getEra() + " land animals section
=====");
    List<LandAnimal> landAnimals = f.createLandAnimals();
    for (LandAnimal a : landAnimals) {
      System.out.println("A " + a.getName() + " " + a.getWalking());
    }
    System.out.println("===== " + f.getEra() + " sea animals section
=====");
    List<SeaAnimal> seaAnimals = f.createSeaAnimals();
    for (SeaAnimal a : seaAnimals) {
      System.out.println("A " + a.getName() + " " + a.getSwimming());
    }
    System.out.println("===== " + f.getEra() + " sky animals section
=====");
    List<SkyAnimal> skyAnimals = f.createSkyAnimals();
    for (SkyAnimal a : skyAnimals) {
      System.out.println("A " + a.getName() + " " + a.getFlying());
    }
    System.out.println("Thank you for visiting the " + f.getEra() + "
exhibition");
  }
}
```

Below is an example output from the museum exhibition:

```
You are in the Cenozoic exhibition
===== Cenozoic land animals section =====
A Mammoth stomping
A Cave lion walking
A Woolly rhino running
===== Cenozoic sea animals section =====
A Otodus hunting
A Whale swimming
===== Cenozoic sky animals section =====
A Argentavis flapping
Thank you for visiting the Cenozoic exhibition
```

Assignment steps:

1. Copy the `NaturalHistoryMuseum` class into your Java project
2. Create a UML class diagram documenting your design
3. Implement at least two abstract factories (eras of your choice)

Bonus Question (10 points)

Use a Java properties file to configure which factory to use in the Natural History Museum exhibition application. Implement a Singleton class `AnimalFactory` that reads the configuration file and instantiates the configured factory in a `static` initializer block.

A Java properties file is a text file that contains entries of the format `key=value` and has Java library support for reading and retrieving those entries (Java Properties documentation). Below is an example of a configuration file `museum.properties` (name can be hardcoded in the singleton):

```
factory=CenozoicAnimalFactory
```

After reading in the properties file, use reflection to load and instantiate the configured class. Useful methods are `Class.forName` to load a class, and `Class.getDeclaredConstructor().newInstance()` to instantiate the class using its default constructor.

Then replace the line `AnimalAbstractFactory f = new CenozoicAnimalFactory();` in the `NaturalHistoryMuseum` class with code using the Singleton as follows:

```
// set up the exhibit
AnimalAbstractFactory f = AnimalFactory.getInstance();
```