CHRISTOPHER NEWPORT UNIVERSITY

DEPARTMENT OF PHYSICS, COMPUTER SCIENCE & ENGINEERING

CPSC 280: INTRODUCTION TO SOFTWARE ENGINEERING

* Assignment 3: Design Patterns *

Instructor: Dr. Roberto A. Flores

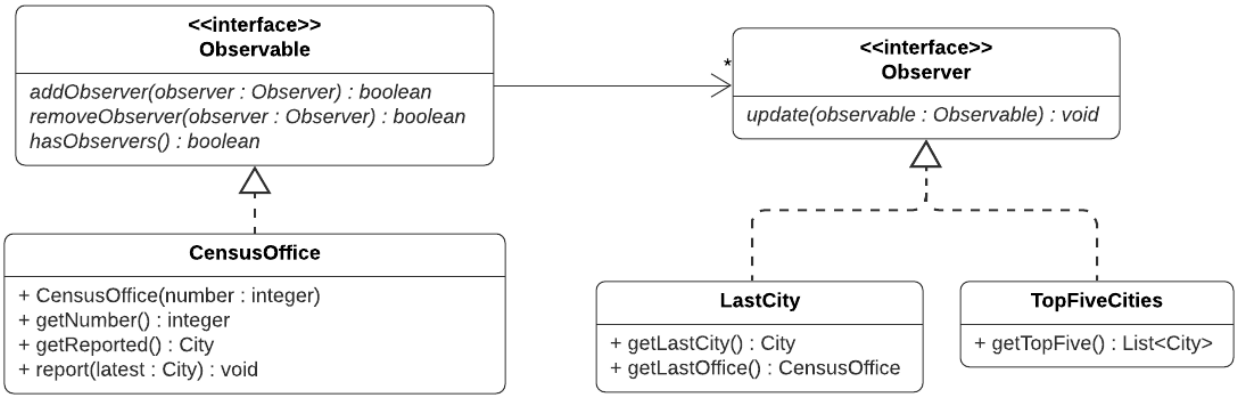
Goal

Practice selected design patterns.

1. Observer

Initial files: *Observable.java*, *Observer.java*, *City.java*, *CensusOfficeTest.java*

Write classes *CensusOffice* (implementing interface *Observable*) and *LastCity* and *TopFiveCities* (which implement interface *Observer*). Each census office has a number (greater than 0) and receives city population reports (using objects of class *City*). Census offices notify registered observers every time city data is reported (which happens when their *report()* method is called). *LastCity* observers keep track of the last city reported and *TopFiveCities* keep track of the five cities with the highest population counts.



In addition to the inherited method *update()*, **LastCity** has getter methods *getLastCity()* returning the last city data received and *getLastOffice()* returning the office that notified the last city data received.

In addition to the inherited method *update()*, **TopFiveCities** implements a getter method *getTopFive()* returning a list with the five top cities (in terms of population) received. The list is sorted from higher to lower numbers. The returned list must be a copy of the list kept by the observer.

In addition to inherited methods, **CensusOffice** has a constructor receiving its number - an illegal argument exception is thrown if the number is 0 or lower (message "office number must greater that 0 [*number*]" where *number* is the number received); getter methods *getNumber()* to retrieve the office's number and *getReported()* to retrieve the most recently reported city; and a setter method *report()* that receives a new reported *City* object.

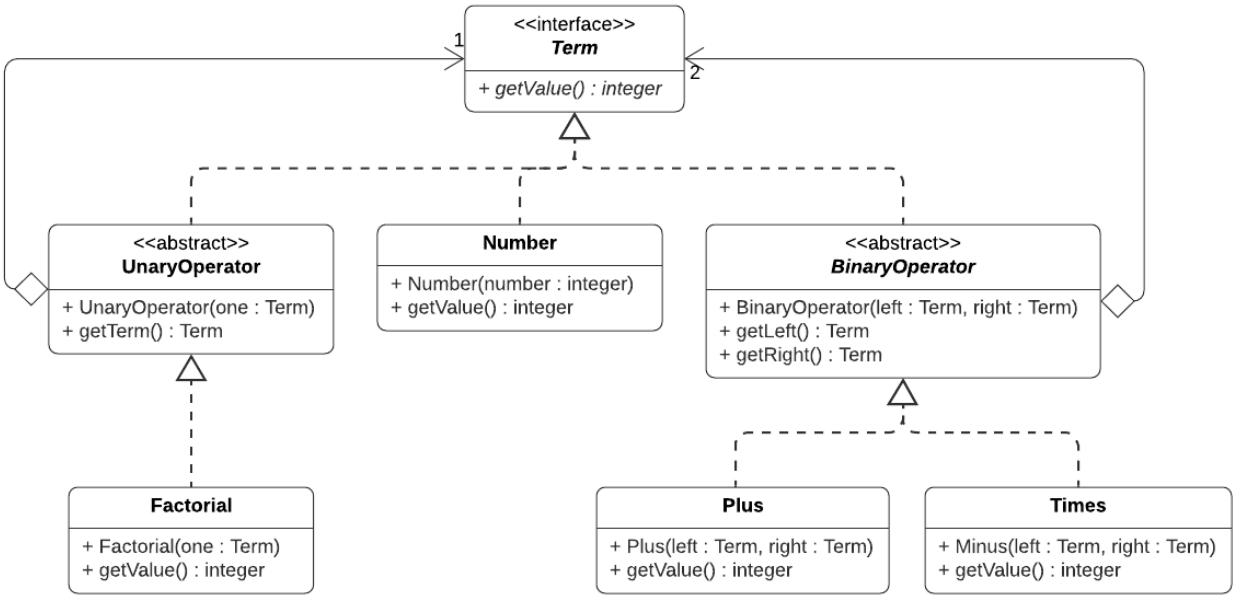
All fields are private and non-static. No static analysis warnings should exist (suppression not allowed). Other private methods in these classes are permitted.

Use *CensusOfficeTest.java* to validate your implementation.

2. Composite

Initial files: *Term.java*, *TermTest.java*

Write the following *Composite* class structure:



Term is the super-type of leaf class *Number* (which holds an integer number) and composite classes *UnaryOperator* and *BinaryOperator*, which hold one and two *Term* objects, respectively. *Factorial* is a concrete *UnaryOperator* class; and *Times* and *Plus* are concrete *BinaryOperator* classes.

Constructors in these classes receive the following values:

- *UnaryOperator*: receives a *Term* object. If the term were null, the constructor throws an illegal argument exception with message "Term cannot be null".
- *BinaryOperator*: receives two *Term* objects. If either term were null, the constructor throws an illegal argument exception with message "*n* term cannot be null" where *n* is either left or right as appropriate.
- *Factorial*: receives a *Term* object. If the value of the term is negative, the constructor throws an illegal argument exception (message "negative value: *value*", where *value* is the value of the term).
- *Number*: receives an integer number.
- *Plus* & *Times*: receive two *Term* objects.

Method *getValue()* in these classes return the following values:

- *Number*: returns the value of the number it holds.
- *Factorial*: returns the factorial of the *Component* object it holds.
- *Plus* & *Times*: return the addition & multiplication (respectively) of the *Component* objects they hold.

Classes *UnaryOperator*, *Number* and *BinaryOperator* are the only classes holding any fields. Classes *Factorial*, *Plus* and *Times* must use the fields in their corresponding abstract classes.

Use *TermTest.java* to validate your implementation.

3. Iterator

Initial files: *FibonacciTest.java*

Write an iterable class *Fibonacci* that generates the Fibonacci sequence (i.e., 1,1,2,3,5,7,...*ad infinitum*). The interface of this class is given below.

```
public class Fibonacci implements Iterable<Long> {
    @Override
    public Iterator<Long> iterator();
}
```

The class implements one method *iterator()* returning an iterator object that provides the Fibonacci sequence when requested through its method *next()*.

Fibonacci numbers should not be precalculated. This means that your implementation must **not** use any arrays or collections (e.g., *ArrayList*) to store Fibonacci numbers; rather your program should calculate the next result only when the method *next()* is invoked.

Use *FibonacciTest.java* to validate your implementation.

Groups

This assignment is done individually or in teams of two as assigned by the instructor.

You must follow the "Empty Hands" policy described in the syllabus. Review the class syllabus or contact your instructor if you're unsure about this policy.

Submission & Grading

Your work must be submitted by the due date to the **Gitlab repository** (https://gitlab.pcs.cnu.edu/) **given to you by the instructor**.

Important notes about the git repository:

- It must be formatted as a **Gradle** project.
- Gradle's **src/main/java** folder must contain your *solution files*.
- Use the *build.gradle* file given to you on scholar.
- Use a *.gitignore* file generated by <http://gitignore.io>. Generate a file customized for OSX, Linux, Windows, Git, Gradle, Eclipse and Java. Make sure it is effectively used by your repository.

Grades for this assignment are based on the test cases given to you.

Up to 20 points could be deducted for non-compliance with the assignment description, e.g., static analysis warnings (Eclipse), failed Gitlab submission, Gradle misconfiguration, incorrect *.gitignore*.

Javadoc is not required.

Solutions to test cases must not be hardcoded.

•••