CS5010, Fall 2019

Refactoring Assignment

Tamara Bonaci and Maria Zontak[1]

t.bonaci@northeastern.edu

- **This assignment is due by 11:59pm on Tuesday, October 15. Please push your individual solutions to your designated repo within the CS5010 GitHub organization, tagged as A2-refactoring-final.**

1. Resources

You might find the following online resource useful while tackling this assignment:

- Java 8 Online Documentation (https://docs.oracle.com/javase/8/docs/api/)
- Java Tutorial: Collections (https://docs.oracle.com/javase/tutorial/collections/index.html)
- Java Tutorial: Basic I/O (https://docs.oracle.com/javase/tutorial/essential/io/index.html)
- Java Tutorial: The Platform Environment (https://docs.oracle.com/javase/tutorial/essential/environment/index.html)
- Java Tutorial: Regular Expressions (https://docs.oracle.com/javase/tutorial/essential/regex/index.html)

2. Submission Requirements

2.1. Gradle Software Management Tool

In this assignment, we will continue using Gradle software management tool, introduced in Lab 2. Below is the reminder on how to setup a Gradle project in IntelliJ.

2.2.1. Gradle and IntelliJ

IntelliJ has a special project type for Gradle projects.

Please follow these steps to setup the project layout for Gradle projects.

1. Start IntelliJ and create a new project.
2. In the selection window, in the left-hand pane, **instead of selecting Java, select Gradle**.
3. Once you select Gradle in the left-hand pane, the right-hand pane populates with additional libraries and frameworks. Make sure to select "**Java**", and click "`Next`".

[1] Acknowledgement: assignment based on the original assignment prepared by Dr. Maria Zontak.

4. You will see a dialog box, asking you to enter "**GroupId**" and "**ArtifactId**".
    1. **ArtifactID**: enter the project name (such as "Assignment3")
    2. Ignore the **GroupId** box.
5. On the next screen, make sure that the selected "Gradle JVM" is Java 1.8, and accept the defaults for everything else.
6. Accept the defaults on the next screen too, and click "`Finish`".

At this point, you should have a Gradle project created, but you are not quite done yet.

The last step is to configure your **`build.gradle`** file so that it does everything we want it to do in this course. The easiest way to do this is to open the "**`build.gradle`**" file in your project directory, and copy/paste the file below into that file (the text with the dark background) (this **`build.gradle`** file is also available on Piazza and the course website for your convenience).

```
plugins {
    id 'java'
    id 'pmd' // PMD: source code analyzer to find common programming flaws
    id 'jacoco' // Code coverage
}

defaultTasks 'clean', 'build', 'javadoc', 'check', 'test'

apply plugin: 'java'

group = 'edu.neu.khoury.pdp'
version = '1.0-SNAPSHOT'
description = 'PDP Fall 2019 Seattle'
sourceCompatibility = '8'
sourceCompatibility = 1.8

repositories {
    jcenter()
    mavenLocal()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
    testImplementation 'junit:junit:4.12'

}

jacoco {
    toolVersion = "0.8.4"
    reportsDir = file("$buildDir/customJacocoReportDir")
}

jacocoTestReport {
    reports {
        xml.enabled false
        csv.enabled false
        html.destination file("${buildDir}/jacocoHtml")
    }
}

jacocoTestCoverageVerification {
    violationRules {
        rule {
            limit {
```

```
                minimum = 0.3
            }
        }

        rule {
            enabled = false
            element = 'CLASS'
            includes = ['org.gradle.*']

            limit {
                counter = 'LINE'
                value = 'TOTALCOUNT'
                maximum = 0.3
            }
        }
    }
}



test {
    useJUnit()

    maxHeapSize = '1G'
}

tasks.withType(Pmd){
    reports{
        xml.enabled=true
        html.enabled=true
    }
}
pmd {
    ignoreFailures = true
    pmdTest.enabled=true
    ruleSets = [
            "category/java/bestpractices.xml",
            "category/java/errorprone.xml",
            "category/java/codestyle.xml"
    ]
}

tasks.withType(JavaCompile) {
    options.encoding = 'UTF-8'
}

task docs(type: Javadoc) {
    source = sourceSets.main.allJava

}

check.dependsOn jacocoTestCoverageVerification
jacocoTestReport.mustRunAfter test

task doAll{
    dependsOn test
    dependsOn check
    dependsOn javadoc
    dependsOn build

    doLast {
```

```
        println 'all done!'
    }
}

jacocoTestReport {
    doLast {
        println "file://$buildDir/jacocoHtml/index.html"
    }
}

javadoc {
    doLast {
        println "file://$buildDir/docs/javadoc/index.html"
    }
}

test {
    finalizedBy jacocoTestReport
}
```

2.2. Code Criteria

- **Naming convention:** Your package name should follow this naming convention assignment2_refactored.
- **Gradle built:** Your project should successfully build using the provided **`build.gradle`** file, and it should generate all the default reports.
- **Javadoc generation:** Your Javadoc generation should complete with no errors or warnings.
- **Checkstyle report:** Your Checkstyle report should have no violations.
- **Code coverage report:** Your JaCoCo report should indicate 60% or more code coverage per package for "Branches" **and** "Instructions".

- **Methods hashCode(), equals(), toString():** all of your classes *have to* provide appropriate implementations for methods:

    - `boolean equals(Object o)`
    - `int hashCode()`
    - `String toString()`

(appropriate means that it is sufficient to autogenerate these methods, as long as autogenerated methods suffice for your specific implementation). Please don't forget to autogenerate your methods in an appropriate order - starting from the ancestor classes, towards the concrete classes.

- **UML diagrams:** Please include UML diagrams for the final versions of your designs for every problem. In doing so, please note that you do not have to hand-draw your UML diagrams anymore. Auto-generating them from your code will be sufficient.

3. Your Goal

Your goal in this assignment is to refactor your Assignment 2 code so that it becomes a **clean code**.

What is a clean code?

Let's see some suggestions from two books recommended for this course:

- **Clean code**, by Robert C. Martin, and
- **Effective Java**, Second Edition, by Joshua Bloch

*Clean code is **simple** and direct. Clean code **reads** like **well**-written prose. Clean code never obscures the designer's intent but rather is **full of** crisp **abstractions** and **straightforward lines** of control.* [Grady Booch, author of *Object Oriented Analysis and Design with Applications*]

*I like my code to be **elegant and efficient**. The logic should be straightforward to make it **hard for bugs to hide**, the **dependencies minimal** to ease maintenance, **error handling complete** according to an articulated strategy, and performance close to optimal... Clean code **does one thing well**.* [Bjarne Stroustrup, inventor of C++]

*Clean code **can be read, and enhanced by a developer other than its original author**. It **has unit and acceptance tests**. It has **meaningful names**. It has **minimal dependencies**, which are **explicitly defined**, and **provides a clear and minimal API**...* [Dave Thomas, founder of OTI, godfather of the Eclipse strategy]

3.1. Some Suggestions

Below are some refactoring suggestions, but these are just that – suggestions - they are not limiting, and they may not cover everything discussed in lectures/codewalks. Please feel free to incorporate all the knowledge and insights that you have gained from the course so far.

3.1.1. Gradle reports

Reports generated as a part of your Gradle project are a great way to start improving/refactoring your code. If your Gradle project fails to build, your correctness will not be graded (that is 0 for correctness), so you want to make sure that is not the case.

So, to start, you may want to bringing your Gradle reports up to the following standards:

- JaCoCo:

i. Code coverage should be above 70%
ii. Methods declared in interfaces, and their implementations should be *green*
iii. Overridden equals and hashCode should have most of it *green*

If you cannot improve code coverage by adding additional tests, that likely means that you have too much logic in static methods, dead code etc. → refactor your code to improve coverage

Recommended reading on proper test design: Clean Code, Chapter 9

- PMD - should have no violations (0 errors and 0 warnings).

3.1.2. Codewalk Comments

During Codewalk week 2, you were asked several questions about your A2 design and code, you were given comments, and you had a chance to see your classmates approach.

You want to consider how to incorporate some of the given comments (and seen ideas) into a refactored version of your Assignment 2.

3.1.3. General Design

Some suggestions for general refactoring:

1. Your code should follow **proper naming conventions** (i.e., see Clean Code, Chapter 2, "Meaningful names")
2. Your code should have **proper programming style**:
    1. **No magic numbers/strings (any hardcoded information inside code)** (i.e., see Item 30 in Effective Java, or Chapters 6 and 17, J3 in Clean Code)
    2. No **dead code**, such as:
        i. unnecessary if statements
        ii. unnecessary nulls and casting
        iii. code/method that will never run (i.e., Clean Code, Chapters 17, G9)
    3. **No copy-pasted or duplicate code** → you should abstract out simple copy/pasted code (this might sometimes require minor modifications (e.g., needs an argument when abstracted))
    4. **No Boolean flags in methods** → Boolean flags mean that your method is doing more than one thing, and that violates our rule **one method – one task.**
3. Your code should **properly capture information** (i.e., Clean Code, Chapter 6, "Object and Data Structures")
    1. Do **not heavily rely on basic types** → not everything should be an array, a string, an int, etc., when a new type might improve encapsulation/reusability/robustness to errors.
    For example: if a problem had to capture the State, i.e., WA, NY etc., an expected solution should have a class State that captures a US state acronym, and where the constructor checks that the String passed as argument has two valid characters.
    2. **Choose data collections that serve design well** (do not put everything is an array, a string, an int etc.). Try to maintain reasonable complexity (reasonable trade-off between clean design, and a bit higher complexity is fine)
4. Please **design your classes carefully** (i.e., Clean Code, Chapter 10, Effective Java, Chapter 4)
    1. Every major data type should have **well defined** interface (i.e., Clean Code, Chapters 17, G8)

    2. **Avoid "God" class anti-pattern** – your code should not have one complex class to rule them all, or one static method to rule them all
    3. Do not miss opportunities for **helper methods**
    4. Set **appropriate visibility of methods/fields**
    5. Maintain **proper cohesion within the class** (a simple way to check this is to see how many fields you defined that are used in every non-static method. Ideally every non-static method uses all the non-static fields.)
5. Use JDK APIs consistently (i.e., Effective Java, Item 47)
    1. Use appropriate types from the JDK (e.g., List) along with their methods
    2. Do not re-implement existing methods
    3. Do not re-implement JDK types if you are NOT asked to.

6. Misc

1. **Do not rely on concrete types, instead of abstract** (interfaces or abstract class) (e.g., Effective Java, Item 52)
    1. Compile time type should and can be an abstract type (unless you are using methods only available in concrete classes), e.g., List<Integer> l = new ArrayList<> and not ArrayList<Integer> l = new ArrayList<>()
2. **Maintain proper Javadocs, including invariants, pre- and post-conditions** (please note that @postcondition/@precondition /@requires are not recognized by Maven and standard Javadocs, instead use @param/@throws/@return)
3. **Do not create silent code** (like aborting your method with return, instead of throwing an exception, or catching exceptions and doing nothing with them, while the code silently continues).
4. Do not **overuse static methods/fields** (they are better than magic numbers, but not in hundreds, and not when they do not have direct relation to the class they are in).
5. **Rethink your design to minimize coupling between your classes.**
6. **Verify completeness and consistency of code of your design.**
7. **Verify modularity of your design** (dividing a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system).
8. Verify that every **major** component of your problem is captured by a certain module/class.

Your tasks:

1. Please refactor the code you developed in Assignment 2 so that it follows the suggestions you may have gotten during your codewalk, as well as with the suggestions above.
2. Please attach a short writeup to summarize all of the changes you have made to your A2 code, and explain why did you do that.

Grading for the Refactoring Assignment:

Your refactored Assignment 2 will be graded separately from your original A2 grade, and it will be graded on the scale from 0-10.

The refactoring grade will be counted towards the course participation component of this course grade.