Guidelines for Refactoring Assignments 2 & 3

YOUR GOAL: Refactor your code for Assignments 2&3 to become a CLEAN CODE.

What is a clean code?

*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]

All the above citations are taken from Chapter 1 "Clean Code" of a VERY recommended book (generally written to serve MANY languages, but with convenient examples in Java):
[**CC**] Clean Code, Robert C. Martin

Another book that you should use this week is
[**EJ**] Effective Java, Joshua Bloch, Second Edition

Below are suggestions for refactoring, however they are NOT limiting and may NOT cover everything discussed in lectures/codewalks. Incorporate all the insights you have gained till now from the course!

Maven

Maven Reports are a GREAT way to start improving/refactoring your code.

If Maven fails to build (unsuccessfully exits) the correctness will not be graded (that is 0 for correctness).

Start from bringing Maven Reports to the following standards:

- JaCoCo:
    i. Code coverage should be above 90%
    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 additional tests that means you have too much logic in static methods, dead code etc. → refactor your code to improve coverage

Recommended reading on proper test design: Chapter 9 in [CC]

- Checkstyle - should have NO violations (0 errors and 0 warnings).
- FindBugs - should have NO violations (0 errors and 0 warnings).
  - Ignore security and performance violations (especially if they suggest you changing code to something you do NOT understand, like lambda functions)
- PMD - should have NO violations (0 errors and 0 warnings).

Points will be reduced if your Maven reports do not compile with the above restrictions after code refactoring!

General Design

Requirements for general refactoring:

1. Your code should follow proper naming conventions: **Chapter 2, "Meaningful names" in [CC]**

2. Your code should have proper programming style:
   a. NO magic numbers/strings (any hardcoded information inside code) – **Item 30 in [EJ], Chapter 6, Chapter 17, J3 in [CC]**
   b. NO dead code, including **Chapter 17, G9 in [CC]**
      i. unnecessary if statements
      ii. unnecessary nulls and casting
      iii. code/method that will never run
   c. NO copy-pasted or duplicate code → you should abstract simple copy pasted code, this might sometimes require minor modifications (needs an argument when abstracted)
   d. NO boolean flags in methods → boolean flags means that your method is doing more than one thing → violation one method – one task.
   e. Maintain same level of abstraction per specific instructional line/method (Do NOT mix important concepts with tedious implementation details) – **Chapters 6 & 17 in [CC]**

3. Your code should properly capture information (**Chapter 6 in [CC] – "Object and Data Structures"**)
   a. Do NOT rely heavily on basic types (everything is an array, a string, an int etc), when a new type might improve encapsulation/reusability/robustness to errors.
      For example: If the problem had to capture the State, i.e., WA, NY etc. I expect the solution to have a class `State` that captures a US state acronym and that the constructor checks that the String passed as argument has 2 characters.
      OR: In assignment 2 – most of the students (even strong students) set patient urgency as int. Instead they should have had a proper class (that can encapsulate any changes in urgency representation or calculation, or allows some meaningful constants like MAX_URGENCY, which should be hold within `Urgency` class)
   b. Choose data structures 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. You should design classes carefully (**Chapter 10 in [CC] – "Classes", Chapter 4 in [EJ] – "Classes and Interfaces"**)

a. **Avoid** "God" class anti-pattern - one complex class to rule them all or one static method to rule them all
b. Do NOT miss opportunities for helper methods → is also helpful to maintain the same level of abstraction (see 2 e. above)
c. Set appropriate visibility of methods/fields → every major type should have **well defined** interface (avoid too much information **Chapter 17, G8 in [CC])**
d. Maintain proper cohesion within the class (a simple way to check is to see how many fields you defined are used in every NON-static method, ideally every non-static method uses all the non-static fields)

5. Use JDK APIs consistently– **Item 47 in [EJ]**
    a. Use appropriate types from the JDK (e.g., List) along with their methods
        i. do NOT re-implement methods
        ii. do NOT re-implement JDK types if you are NOT asked to.

6. Misc
    a. Do NOT rely on concrete types instead of abstract (interfaces or abstract class) - compile time type should be an abstract type (unless they are using methods only available in concrete classes), e.g., `List<Integer> l = new ArrayList<>` and **NOT** `ArrayList<Integer> l = new ArrayList<>()` (**Item 52 in [EJ]**)
    b. Maintain proper Javadocs, including invariants, pre- and post-conditions (please note that @postcondition/@precondition /@requires are NOT recognized by Maven and standard JavaDoc, instead use @param/@throws/@return)
    c. 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).
    d. 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).
    e. **Rethink your design to minimize coupling of between your classes (**see Lecture 5**)**
    f. **Verify completeness and consistency of code of your design (**see Lecture 5)
    g. **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). Verify that every **major** component of your problem is captured by a certain module/class (that is in Assignment 3 you might want to have CommandLineHandler, InputOutputHandler, Template, Database, Binder/PlaceholderEvaluator) and that you define/operate through proper interfaces.

WHAT TO SUBMIT?

1. **Push your new code by next Monday, October 16th 6:00 p.m.**
2. **Attach a writeup – summary of ALL the changes you have incorporated and WHY you did so** (provide a reason for every change).
3. **This assignment is an opportunity to improve grade for previous two!**

GOOD LUCK!!!