Assignment 3 - Fall 2021
CSCI 5408 - Advance Topics in Software Development

Assignment by:

Full Name: Adesh Nalpet Adimurthy

Banner Number: B00886154

Dal FCS CSID: adimurthy

Gitlab Profile: https://git.cs.dal.ca/adimurthy

Assignment Repository:
https://git.cs.dal.ca/courses/2021-fall/csci-5308/assignments/adesh_n/-/tree/master/A3

1. Cohesion and Coupling:

Although the convention is to fix coupling across modules and cohesion among the packages, given the simple use case, all the classes are in the same module (A3 is a single module maven project).

Following are the packages created to increase cohesion and reduce coupling:

- **Models:** Employee, Driver, ForkLiftOperator, Pricker, and Instruction (Enum). The purpose of having a separate package for models is for extensibility, which makes it easier to share models across modules.
- **Services:** Receiving and Shipping are the services classes that understand the usage of human resources and warehousing.
- **Storage:** HumanResource and Warehouse belong to the storage layer; in the current implementation, the data is stored in memory, but extending it to use other storage technologies such as MySQL and MongoDB should not affect the application classes. Hence, HumanResource and Warehouse are the interfaces; InMemoryHumanResource and InMemoryWarehouse are the implementations representing the in-memory storage. Thereby reducing the coupling between the business and storage layer.

2. Code Layers:

For better separation of concerns, the code layers or boundaries are as follows:

- **Persistence layer:** storage package, responsible for managing/storing human resources and warehouse.
- **Business Logic Layer:** services packages, responsible for receiving and shipping items.

- **Presentation Layer:** the root directory of the warehouse package, where the input is a file and output is on the console.

3. Clean Code:

Without specifying the exact set of modified lines, the steps taken to write clean code are as follows:

1. Followed camel casing for attributes, functions, and local variables.
2. Renamed local variables into meaningful names. For example, "p" is renamed to "picker" and "w" to "warehouse."
3. Removed comments and made changes to the code to make it evident. For example, the comment for "c" representing the count is fixed by renaming "c" to "count."
4. Used System.out.printf with the correct use of string replacement instead of string concatenation across the codebase. Example: System.*out*.printf("Received %d %s", count, item);
5. Removed unused local variables. For example, Driver and Picker in the Shipping class are not used. However, having a local variable is not wrong, even for something as simple as a null check. Assuming that "getDriver" and "getPicker" do not return null, the change is valid.
6. Followed naming consistency throughout the codebase. For example, shipping and ship are used interchangeably; however, shipping is the service, and ship is the method. Refer for naming conventions: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html
7. Code indentation and Spacing convention followed from: https://www.oracle.com/java/technologies/javase/codeconventions-whitespace.html

4. Refactoring:

While there are tons of refactoring techniques, some of the refactoring done are as follows:

1. **Primitive Obsession:** Going by the principles of OOPs, everything is an Object. Although Java is not purely object-oriented, a convention is to use wrapper classes for primitive types. For example, int is replaced by Integer.
2. **Switch Statements:** The if-else conditions for the instruction are replaced by a visitor pattern; while this isn't a complex switch/complex if-else condition, it's always better to enforce the use of visitor pattern when all instructions are to be considered.
3. **Comments:** As mentioned above, comments explaining the code are removed throughout the code base, and code readability has been improved to accommodate the removal.

4. **Duplicate Code:** An example for duplicate code is the "getDriver," "getPicker" and "getForkLiftOperator," which is refactored by introducing a private generic method "getEmployee."
5. **Dead code & Speculative Generality:** the Warehouse class is not used in the Shipping and Receiving class, although present in the constructor, with the two options of either removing the Warehouse class or implementing the add and remove methods. The latter has been implemented.
6. **Long Method:** Decomposing into smaller methods improves clarity. The main method is decomposed into smaller methods to separate the functionality and enhance readability without overdoing it.

Apart from these, refactoring reduces coupling and increases cohesion significantly, contributing to better extensibility, readability, and easy codebase maintenance.

5. Design Patterns:

Builder Pattern:

The HumanResource no-args constructor initializes the private fields of the class for instantiation. Instead, using the Builder Pattern, the need for a no-args constructor is eliminated, and instantiating an object is also cleaner in the Main class. The InMemoryHumanResourceBuilder is the builder for the InMemoryHumanResource, which implements the interface HumanResource.

Visitor Pattern:

From the input file "input.txt," it is that there are two instructions "RECEIVE" and "SHIP." While a string comparison works well, using a combination of Enum and Visitor design patterns ensures that all the instructions are handled.

Iterator Pattern:

Using the Iterator pattern to calculate the total minutes worked is faulty. The Iterator is mutated; furthermore, the whole point of using an iterator is lost and can be replaced by a simple for loop. The Iterator pattern is ideally used when the list size changes or handles extensive data in the memory in batches. Therefore, the Iterator is replaced by the for-loop.

6. Other Refactoring:

1. The minutes taken for each work are hardcoded in the employee subclasses. The models should not have such constants. The better place to get these values would be from the service classes (Receiving and Shipping).
2. Furthermore, having all the hard-coded values in one place, either in a configuration file (if the values change often) or in a Constant class with a private constructor, is preferred. For the sake of simplicity, the Constants class in application packages holds the minutes' values for employee subclasses.
3. The input to the application can be from different sources, and input from the file system is one such use-case. Hence, having an abstract class for different input types may be better. However, keeping extensibility in mind and not over refactoring, having an Input class to parse the input file into makes it reusable to extension. The merge request for this: https://git.cs.dal.ca/courses/2021-fall/csci-5308/assignments/adesh_n/-/merge_requests/18

7. General References:

"Code Conventions - Whitespace," Oracle [Online]. Available: https://www.oracle.com/java/technologies/javase/codeconventions-whitespace.html [Accessed: November 30, 2021].

"Code Conventions - Naming Conventions," Oracle [Online]. Available: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html [Accessed: November 30, 2021].

"Refactoring," Refactoring Guru [Online]. Available: https://refactoring.guru/refactoring [Accessed: November 30, 2021].