Milestone 3 Final Assessment Written Analysis

Name: Prashant Sanjay Sarvi
Banner: B00870599
Group # of assessed project: 11

You have assessed another group's work and completed the assessment spreadsheet. You must now perform the equivalent of a take home final exam. You will now provide further details on the group's work, as related to the major course learning outcomes.

In each section below you must explain **in your own words, with no assistance from anyone else**:

- Why the code is correct, and a short analysis and explanation of the best practices the team followed to achieve their result.

  AND/OR

- Why the code fails to achieve a best practice, or avoid a bad practice or anti-pattern, **and which refactoring techniques you would use to correct the code**.

When you highlight an issue with their code it must be accompanied with an explanation of which refactoring technique you would use to correct the code. Do not forget this vital information, it is a major portion of your grade for this assessment. When listing the refactoring techniques, list one of Martin Fowler's techniques and then explain what changes these techniques would have on the code (what classes/methods/abstractions would be renamed, added, deleted or modified).

Clean Code:

The project assessed adhere to the some of the clean coding standards:

- Group displays good use of configuration
- All most no usage of control flags
- There we no usage of negative conditions

But the group has violated certain clean coding practises which needs to be addressed:

a) **Issue**: Unused variables or imports

Example:

- In src/test/java/group11/Hockey/BusinessLogic/TeamCreation/CreateTeamTest.java at line number: 38, 39 and 40, the objects created are unused.

- In src/main/java/group11/Hockey/db/Constants.java, at line number: 29 the constant field is never used.
  Solution: The unused imports can be removed from the code that will lead to more readability.

b) **Issue:** Comment noise (unnecessary comments cluttering files)

Example:

- In src/main/java/group11/Hockey/BusinessLogic/LeagueSimulation/Schedule.java, at line number: 111, 122, 149, 194, 209, 236, 294 and 321, there are unnecessary comments found cluttering the file.

- In src/main/java/group11/Hockey/InputOutput/JsonParsing/ParseRootcoaches.java, at line number: 37, 39, 41, 43 and 45, there are unnecessary comments found cluttering the file.

Solution:

The comments aren't required if proper meaningful names are given to the classes, methods and variables. They can be removed as they create noise and confusion to the programmers.

c) **Issues**: Consistency issues

Example:

- In src/main/java/group11/Hockey/BusinessLogic/Trading/TradeRunner.java, at line number: 53, there is no space after "if" but in src/main/java/group11/

- 2. In src/main/java/group11/Hockey/BusinessLogic/models/Roster/Roster.java, at line number: 37, there is no space after "if" but in src/main/java/group11/Hockey/BusinessLogic/models/Roster/Roster.java, line number: 51, there's a space after "if". So, there's consistency issues.

Solution:

The uneven spaces between the "for" and "if " can be removed by indenting in according to the group standards. The new programmer should first check the files and indentation followed by the group and follow the same. The IDE also helps to indent the code.

d) **Issue:** The pre-conditions weren't asserted in specific areas of code.

Example:

- In src/main/java/group11/Hockey/BusinessLogic/models/Roster/Roster.java, at line number: 81, the object passed through the parameters of the constructor fails to assert the pre-condition to check if the passed object of type "Player" and objects "one", "two" to check if it's null or not.

- In src/main/java/group11/Hockey/BusinessLogic/models/GeneralManager.java, at line: 52, the object passed through the parameters of the constructor fails to assert the pre-condition to check if the passed which is "managerList" of type "GeneralManager" to check if it's null or not.

Solution:

The pre-conditions can be mentioned for checking of the null value when objects are passed through the parameters. This can reduce the chances of creating bugs in the method.

Layers:

Good implementation of the code which is segregated by various layers namely Business logic layer, persistence layer and Presentation layer but there are some violations which has to be identified and refactored. Some of them are:

a) **Issue:** Usage of "System.out.println()" even in the presence of presentation layer.

Examples:

- In src/main/java/group11/Hockey/db/Team/TeamDbImpl.java at line number: 69, loggers are not used instead "System.out.println" is used to display the error message to the user.

- In src/main/java/group11/Hockey/db/League/LeagueDbMock.java, the mock data class is present in the business logic which is violation of the layer.

Solution:

Usage of presentation layer to perform all the display functionality to the user should be maintained uniformly throughout the code in all the layers which is display.ShowMessageOnConsole from the presentation layer should be used.

b) **Issue:** Usage of Mock data in the business logic layer

Examples:

- In src/main/java/group11/Hockey/db/League/LeagueDbMock.java, the mock data class is present in the business logic which is violation of the layer.

Solution:

The mock data class should be present in the Test Driven Development package.

S.O.L.I.D.:

The code has followed some of the solid principles in a good way which are:
- Open/Closed principle
- Liskov Substitution Principle

But there are few places where the violation of SOLID principles has taken place:

a) **Issue**: Class share more than one responsibility

Example:

1. In src/main/java/group11/Hockey/InputOutput/JsonParsing/JsonImport.java the class has method startState() which does not have the same responsibility as JsonImport class. So, violating the single responsibilty principle.

2. In src/main/java/group11/Hockey/BusinessLogic/models/Training.java the class contains methods relating to “Training” but at line number: 14, method getDaysUntilStatIncreaseCheck () has some other responsibility which is related to Stats and not related to the class. So, it is violating single responsibility principle.

Solution:

Few methods present in the above mentioned classes are not appropriate and the class has more than one responsibility. The methods should be moved to the appropriate classes by refactoring the code.

b) **Issue:** Violation of Interface Segregation Principle

Example:

1. In src/main/java/group11/Hockey/BusinessLogic/models/IPlayer.java the interface contains more number of methods which is not required by the class such as method getIsFreeAgent() which is not implemented by any class. The interface is bloated and has more than one reason to change. Therefore, violating the interface segregation principle.

2. In src/main/java/group11/Hockey/BusinessLogic/models/ITimeLine.java the interface contains more number of methods declared which is not required by the class more than three methods are not implemented by any class. The interface is bloated and has more than one reason to change. Therefore, violating the interface segregation principle.

Solution:

The interface should not contain unnecessary methods and should not force the concrete class to implement it. The unused methods in the interface should be removed. The bloating of the interface can be reduced by only declaring the useful methods and interface shouldn't have more than one reason to change.

Cohesion & Coupling:

Issue:

The group has done good implementation of the cohesion and coupling in the code but in the code, I have noticed that the importing packages via * makes sense but the components are not so cohesive to have everything in the project. The code also faces coupling with the packages.

Example:

In src/main/java/group11/Hockey/InputOutput/JsonParsing/JsonImport.java
The class jsonImport implementing the state machine method which is startState(). So, this will form a dependency cycle and making these two packages tightly coupled.

Solution:

The refactoring of the code by moving the startState() to the appropriate class and removing it from the JsonImport class can remove the dependency cycle making it loosely coupled.

Design Patterns:

- There is a good implementation of creational pattern and all the packages have well defined abstract factory class usage like in src/main/java/group11/Hockey/BusinessLogic/Trading/TradingFactory.java, the abstract factory is used for trading class so that we can get rid of the new key word. Default Abstract factory is also created and implemented in some classes.

- Good use of structural design pattern, the structure implemented in the project is flexible and efficient.

- The observer pattern as seen in src/main/java/group11/Hockey/BusinessLogic/Trophy/Interfaces/ITrophySubject.java is implemented in a proper way but doesn't make use of the singleton to attach and detach the observer and even to send the notify calls to the subscriber classes.