CIS573 –Software Engineering

4th Homework

Ning Xu 20139217

Yayang Tian 24963298

WorldSeriesInstance

1. Design Problem

Lack of Abstraction and Information Hiding:

In this class, there is one main design problems. Since there is only one method displaying "winner defeated the Loser" while no method displaying the complementary sentence like "loser lost to the winner". This leads to the lack of Abstraction and Information Hiding in the "showDataForTeamLosses" method of "DataProcessor" class. Since the method can only output the loser by accessing "year()", "winner()", "loser()" and "score()" methods in "WorldSeriesInstance", which exposes too much information about "How to do" instead of "What to do" and causes much coupling between these two class as well. As a result, it also decreases the testability.

2. Redesign Solution:

We modify the name of "toString()" method as "toStringWinner()" and also add a new method called "toStringLoser()", which display the winner and loser respectively. The new added method may increase the test scope of the class. However, the test complexity of the classes won't change. Besides, it can help improve the testability of the class "DataProcessor" (Assist fault localization and increase the simplicity).

Changes are backed off:

None

3. Tradeoffs:

As said, the new added method may increase the test scope of the class "WorldSeriesInstance". But it can help improve the testability of the class "DataProcessor". Therefore, we decide to keep this change.

DataStore

1. Design Problem

(1)***Modularity & Aspects***: There are two main design problems in this class. One is about the logging. Notice that there are "log()" methods in all three classes "DataStore", "DataProcessor" and "UserInterface". However, The functionality of these "log()" is the same. Therefore it violates the design concepts Modularity and Aspects, thus decreasing the Testability (Test Simplicity). Besides, it's desired to use design pattern "Singleton" to handle the logging problem.
(2)***Testability & Information Hiding***: Another problem of this class is about the HashMap variable "_resultsCache". It's defined as public, thus making it can be modified by externally, which reduces the Testability. Besides, a lot of other modules have no need to access it, which render it lack of information hiding.
(3) ***Procedure Abstraction***: The function name "readDataFileAndPopulateArrayList" tells too much about the detailed implementation about how it does instead of what it does, which obeys the rule of procedure abstraction.
(4) HardWiring & Testability
we cannot create test case concerning variable "_list" and "HashMap", because it's private. We have to modify it.

2. Redesign Solution:

(1)***Singleton:*** We handle the logging problem by using the design pattern "Singleton". We create a new class called "logger", and define the variable class "type" as protected static. Then we create a static method "setType()" to return the variable "type". We also create an unique "log()" in this class to replace the other three "log()" methods in different classes. Through using Singleton pattern, we increase the Modularity, Aspects, Abstraction as well as Information Hiding of other classes. In addition, the scope of Testability is improved, since it's only needed to test the "log()" method in one class "logger". Because we use static way to keep only one instance of "logger" class, the security is also improved.
(Factory Method): Alternatively, we can also abstract these logging to be factory method, letting subclasses decide which class to instantiate. As a result, "log" method would write in the "log.txt" in terms of "DataStore", "DataProcessor" or "UserInterface".

(2)***Singleton:*** We handle the second problem also by using singleton pattern, which is the same as the above one. We changed the type of "_ resultsCache" to be protected static. And

create a new method “resultsCache()” to return the static variable. This can improve both the Testability and security since the variable can’t be modified externally randomly.

(3)We simply change "readDataFileAndPopulateArrayList" to "loadFile".
(4)We create a class "DataStoreHelper" to see the side effect by calling it during testing.

3. Tradeoffs:

***(1)Changes are backed off:**
At first, we misunderstood the functionality of “lookup()” in the “DataStore” class, therefore we moved this method to the “logger” class. However, finally, we think it’s more appropriate to keep the “lookup()” method in the “DataStore” class, since it’s more related to the functionality of this class. Otherwise we will violate the Modularity of “logger” classes.
One tradeoff of using Singleton pattern is to “make everything static”. Indeed we have to change the type of all related methods to be static. However, since the Singleton pattern can improve the design concepts: Modularity Aspects, Abstraction and Information Hiding, we still prefer to keep these changes.

(4)One draw back is that it would reduce information hiding and other class may mistakenly calls it.

DataProcessor:

1. Design Problem:

(1)Lack of Aspects and Abstraction:

One of the problems is lack of Aspects and Abstraction. In the method “showDataForTeamAll()”, a large part of codes is similar as that in the method “showDataForWins()” and “showDataForLosses()”. This is due to that the functionality of “showDataForTeamAll()” is just the combination of the two methods “showDataForWins()” and “showDataForLosses()”. Therefore, it’s more suitable to replace the copy of codes with the two existing methods.

(2) Testability and Security:

Since there is only one instance _dataStore, and in the "UserInterface" class, the client will access this global and single instance (external database), in order to prevent wrongly initialize twice especially during testing, we have to make some restriction.

(3) Cross-cutting concern & Information Hiding:

We create "allWorldSeriesInstance()" all distributed in "DataProcessor"class. However, all the following functions(module) don't need to initialize it for so many times and get know the details.

(4) Functional Independece:

The core task of "DataProcessor" is to search data from the database, not to interact with clients. The function "displayTeams" obyess functional Independence because it overly depends on other class. That is ,printing lines to the screen is what should be done in "UserInterface".

(5) HardWiring & Testability

we cannot create test case concerning variable "_datastore", because it's private. We have to modify it.

2. Redesign Solution:

(1)Facade:

We use the Facade pattern to address this problem. We only need to replace the repeating codes in the “showDataForTeamAll()” method with the two existing methods. Therefore we don’t need to know the details about how to find the winners or losers, which improves the Abstraction. Besides, in order to keep the specification, we also add two static variables “wins” and “losses” to count the numbers of wins and losses.

Changes are backed off:

(2)Singleton:

To fix this problem, we use singleton method. That is,we change the instance member "_dataStore"to static to hold the only instance, as well as change the construction method "DataProcessor" ""to be protected to ensure single initialization. Lastly, to let it be accessible, we create a access method "getDataInstance".

(3)Singleton:
We set the list as a protected static instance, and initialize it in the constructor, which is also protected to avoid multiple initialization.

(4)we simply move this function "displayTeams"to "UserInterface", and delete the input argument ui within this method.
(5)We create a class "DataProcessorHelper" to see the side effect by calling it during testing.

3. Tradeoffs:

(1)Although we derive a unified and simple interface, It might deviate from the original specification slightly. Because it displays the results firstly for "won" then for "lost".
(2)(3)Altough we restrict instantiation to only one object., it also bring about problem that no external control over arguments to construct. But it worth it since we get a higher security.
(5) One draw back is that it would reduce information hiding and other class may mistakenly calls it.

UserInterface

1. Design Problem

(1) ***Functional independence :*** As is said in "datastore", "log" is independent of this class. To avoid coupling between classes we have to modify it.

(2) ***Functional Independence & Modularity:*** Since the function "assembleWinnersByTeam" in "UserInterface" does not demand for interaction between clients and computer, just find winning teams and add to the arraylist, which is exactly what "DataParser " does. This is much different to the functions before. So in order to keep cohesion within class and avoid coupling, we'd better put it somewhere else.

(3) Testability & Information Hiding & Abstraction & Modularity:
In the section of "DataProcessor", we moved the method "displayTeams" to this class. That is not enough. The two steps in "if-which" choice 4 expose too much details to clients, thus poses great thread to information hiding and abstraction. In addition, since it lacks cohesion between context, it further increases the complexity in testing.

(4) Abstraction & Information Hiding:
"askUserForTeamName" and "askUserForWinLossOrAll" are really bad names. Because they tell too much information concerning how the function does to the clients which is not necessary. Furthermore, the split the "searchByTeam" method into two input steps are entirely redundant and unnecessary. This is not a good abstraction at all, thus increase the complexity of testing and non-cohesion within class.

(5) HardWiring & Testability
we cannot create test case concerning variable "_datastore" and "_dataprocessor", because it's private. We have to modify it.

2. Redesign Solution:

(1)***Singleton or Factory Method:*** we simply delete "log" and modify all the sentences including "log". This is discussed in section " datastore".

(2) We put "assembleWinnersByTeam" funtion in class "UserInterface" to class "DataParser" which result in higher lever of modularity and function independence. As a result, we have to change the returned from value void to "Arraylist".

(3) To address this, we abstract the two steps "assembleWinersByTeams" and "displayTeams"into one, and modify them to becomes one method "searchAll". So the cohesion with class "interface" greatly enhances.

(4) Resemble to the method "searchByRange", which doe not expose the information like

"askUserforStartYear" and "askUserForEndYear", we encapsulate what "askUserForTeamName" and "askUserForWinLossOrAll" do into method "searchByTeam".
(6)We create a class "UserInterfaceHelper" to see the side effect by calling it during testing.

3. Tradeoff

(1)(2) It makes fault localization easier and we can change one without changing another. Besides, it reduces couping between classes. No apparent drawback.
(4)This would hide the detailed information from clients for the interaction. However, the hiding is meaningful because we have to get good abstraction and consistency within class.
(5)One draw back is that it would reduce information hiding and other class may mistakenly calls it.