Analyzing fEMR's cohesion and coupling

Team - 7

Name	NetID	Email
Md Samiul Aftad Chowdhury	mac151830	mac151830@utdallas.edu
Kenechukwu Chidozie Nwankwo	kcn062000	kcn062000@utdallas.edu

For the following classes, the cohesion metrics used by SourceMeter are

Lack of Cohesion in Methods 5(LCOM5):

This metric measures the lack of cohesion and computes into how many coherent classes the class could be split.

Classes with highest cohesion:

1. femr.ui.controllers.TriageController
2. femr.data.models.mysql.PatientEncounter

Classes with lowest cohesion:

1. femr.common.ItemModelMapper
2. femr.data.daos.Repository

Difference between highest cohesion versus lowest cohesion:

After analysis of low cohesion class "Repository" and the high cohesion class "TriageController" we found the following:

"Repository" shows lack of cohesion. This class can be split into multiple coherent classes. In that class, methods carry out a high number of related activities. It shows the lack of Functional cohesion.

On the other hand, "TriageController" shows high cohesion. This class uses functions from already split multiple coherent classes. In that class, methods carry out a low number of related activities. It shows a high Functional cohesion.

For the following, two class coupling metrics used by SourceMeter are

Coupling between Object classes (CBO):

This measures number of directly used other classes (e.g. by inheritance, function call, type reference, attribute reference). Classes using many other classes highly depend on their environment, so it is difficult to test or reuse them; furthermore, they are very sensitive to the changes in the system.

Classes with highest coupling:

1. util.dependencyinjection.modules.TestDataLayerModule
   Type of coupling: CBO

Classes with lowest coupling:

1. femr.ui.controllers.TriageController
   Type of coupling: CBO

For the following, two class coupling metrics used by SourceMeter are

Coupling between Object classes Inverse (CBOI):

This measures the number of other classes which directly use the class. Classes which are used by many other classes have a high impact on the behavior of the system, and should be modified very carefully and tested intensively.

Classes with highest coupling:

2. femr.common.dtos.ServiceResponse
   Type of coupling: CBOI

Classes with lowest coupling:

2. femr.data.DataModelMapper

   Type of coupling: CBOI

Difference between highest couplings versus lowest coupling:

After analysis of low coupling class “TriageController” and the high coupling class “TestDataLayerModule” we found the following:

TestDataLayerModule class directly used other classes by inheritance, function call, type reference, and attribute reference. This class accesses local data of another module in many places. This class shows high dependency on other modules. Other classes have a high impact on the behavior of the system on this class, and should be modified very carefully and tested intensively.

TriageController has very low knowledge of the other component classes. This Class directly instantiates very few objects of other classes, and does not go as far as accessing member variables. This makes it much less dependent on other classes. This class does not share the same global data as other classes.

Detecting code smells in fEMR

<u>**Smell:** Message Chains</u>

Method:

findTabFieldMultiMap (int encounterId, String tabFieldName, String chiefComplaintName) : ServiceResponse

findTabFieldMultiMap is affected by Message Chains because:

The method uses one object to access another object, then uses the obtained object to access another object, and so on with all objects having different types.

Method:

createMissionTripItem (IMissionTrip missionTrip) : MissionTripItem

createMissionTripItem is affected by Message Chains because:

The method uses one object to access another object, then uses the obtained object to access another object, and so on with all objects having different types.

We agree that the above methods have Message chains smells. Those two functions used chains for function calls on multiple objects. Any change to the intermediate relationships causes the client to have to change.

<u>**Smell**: Data class</u>

Class: MissionItem

MissionItem is affected by Data Class because:

The class is exposing a significant amount of data in its public interface, either via public attributes or public accessor methods

Class: CitySearch

CitySearch is affected by Data Class because:

The class is exposing a significant amount of data in its public interface, either via public attributes or public accessor methods

We agree that the above classes have Data class smells because those two classes had some unused variables and methods. Also those Classes contain only fields, and their setters and getters.

__**Smell**: Data Clumps__

Method:

createPatient (int userID, String firstName, String lastName, Date birthday, String sex, String address, String city, Integer photoID) : IPatient

createPatient is affected by Data Clumps because:

The method has a long parameter list, and its signature or a significant fragment thereof is duplicated by other methods. This is a sign that the group of parameters, being passed around collectively to multiple methods in the system, could form a new abstraction that could be extracted to a new class.

Method:

createPatientItem (int id, String firstName, String lastName, String city, String address, int userId, Date age, String sex, Integer weeksPregnant, Integer heightFeet, Integer heightInches, Float weight, String pathToPatientPhoto, Integer photoId, String ageClassification) : PatientItem

createPatientItem is affected by Data Clumps because:

The method has a long parameter list, and its signature or a significant fragment thereof is duplicated by other methods. This is a sign that the group of parameters, being passed around collectively to multiple methods in the system, could form a new abstraction that could be extracted to a new class.

We agree that the above methods have Data Clumps smell because those two methods have long parameter list. Groups of variables, parameters or fields in different parts of the code that usually appear and change together. These items should be part of a separate entity

Refactoring analysis

Automated refactoring in fEMR

1. Refactoring smell <u>Data class</u> for the class: MissionItem

Steps used for automated refactoring:

We have changed the constructor from MissionItem() to

MissionItem(String teamName, String teamLocation, String teamDescription, List<MissionTripItem> missionTrips)

We used "Changed Signature" (ctrl+F6) refactoring method for this Constructor.

We have added 4 new parameter using "Changed Signature" refactoring tool.

Later we have used "Removed unused resources" to remove some unused variable and some accessory methods to access those variable.

We did some manual refactoring to remove some previous method call which were no longer needed for this class.

Running Incode on the refactored code showed that it reduced the severity of that smell from 2 to 1 but did not completely remove the bad smell.

2. Refactoring smell <u>Data Clumps</u> for the method: createPatientItem

Steps used for automated refactoring:

By using the InCode's "Browse detected parameter clusters "it showed us which parameters in the function call causing the "Data Clumps" smell.
Then we decided to implement a new createPatientItem function with different parameters.
We found that all the values passed to createPatientItem function are already present in IPatient interface which is used to get all the values before calling that method.
We decided to pass the whole IPatient type object to creatPatientItem function and from there we will use accessor functions to get the values.

We have changed the function from **createPatientItem**(int id, String firstName, String lastName, String city, String address, int userId, Date age, String sex, Integer weeksPregnant, Integer heightFeet, Integer heightInches, Float weight, String pathToPatientPhoto, Integer photoId, String ageClassification);

Then we declared a new signature for the method createPatientItem

createPatientItem (IPatient patient, , Integer weeksPregnant, Integer heightFeet, Integer heightInches, Float weight, String pathToPatientPhoto, Integer photoId, String ageClassification)

We used “Changed Signature” (ctrl+F6) refactoring method for this Constructor.

We have added 1 new parameter using “Changed Signature” refactoring tool and removed 7 old parameters that we can directly use using IPatient object.

Later we have used “Removed unused resources” to remove some unused variable and some accessory methods to access those variable.

We did some manual refactoring match the function calls from different classes.

We rerun the InCode tool on our code after manual refactoring and found that it have reduced the “Data Clump” bad smell for the createPatientItem function.

Manual refactoring in fEMR

1. Refactoring smell Data class for the class: CitySearch

Steps used for manual refactoring:

We browsed seemingly unused data in the class CitySearch.

We used Intellij’s unused code search option along with the suggestion from InCode.

We found a variable “missionCountryId” and its setter method never used in the code.

We manually removed the variable and the setter method from that class.

Running InCode on the manually refactored code we found that “Data class” smell for CitySearch is no longer there.

2. Refactoring smell Data Clumps for the method: createPatient

Steps used for manual refactoring:
By using the InCode’s “Browse detected parameter clusters “it showed us which parameters in the function call causing the “Data Clumps” smell.
Then we decided to implement a new createPatient function with different parameters.
We found that all the values passed to createPatient function are already present in PatientItem object which is used to get all the values before calling that method.
We decided to pass the whole patientItem object to creatPatient function and from there we will use accessor functions to get the values.

Then we declared a new signature for the method createPatient
IPatient createPatient(PatientItem patient);
In the file IDataModelMapper.java
And implemented the new function to the file IDataModelMapper.java
This newly created method using the accessor functions of PatientItem class.
Which reduced the parameter list of createPatient from 8 to 1.

We reran the InCode tool on our code after manual refactoring and found that it had reduced the "Data Clump" bad smell for the createPatient function.

Comparison of manual and automated refactoring process:

Both the manual and automated refactoring processes have pros and cons.

In our case, when we did manual refactoring we found that it is very time consuming because we had to change every line of code manually and we had to search for the codes where it might create side effect for changing that particular code. On the other hand, automated refactoring gives us the opportunity to automate most for the refactoring process. But at a cost, when a method was called from different places it's quite difficult to automate the process if there are some variable declaration is involved.

In our case problem, we have faced with automated refactoring that might be due to limited knowledge of automated refactoring with IntelliJ. Before doing this assignment, our team had very little knowledge about automatic refactoring and we had to spend a lot of time to understand its process.