ASSIGNMENT2

SE-6356

Team -6
Raghu Vasantavada
Devika Chakraborty

1.Analyzing fEMR's cohesion and coupling

Cohesion

For High cohesion , we chose the following 2 classes.

1. Class Name : femr.ui.controllers.TriageController
2. Class Name : femr.business.services.system.UserService;

Per Uncle Bob's (Robert Martin)'s codecleaners.com, cohesion is high when

- Classes have a small number of instance variables.
- Each of the methods of a class manipulates one or more of those variables.
- In general, the more variables a method manipulates the more cohesive that method is to its class.
- A class in which each variable is used by each method is maximally cohesive.
- Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

1. Highly Cohesive classes/methods

1.1 Class Name: femr.ui.controllers.UserService

Sourcemeter LCOM = 0

Conclusion: Highly Cohesive

Number of instance variables: 5
Number of methods: 10
Instance variables: RoleRepository, userRepository, itemModelMapper, dataModelMapper, passwordEncryptor

The instance variables were used in majority of the methods as shown below

Sno	method Name	Instance Variables used in each method	Method uses instance variables?	Method corresponds to class's function
1	createUser	userRepository, roleRepository dataModelMapper	Y	Y
2	retreiveALllUsers	userRepository, itemModelMapper	Y	Y
3	retrieveUsersByTripId	userRepository, itemModelMapper	Y	Y
4	toggleUser	userRepository, itemModelMapper	Y	Y
5	retrieveUser	userRepository, itemModelMapper	Y	Y

6	updateUser	roleRepository, userRepository, itemModelMapper	y	Y
7	retrieveByEmail	userRepository	y	Y
8	retrieveById	userRepository	y	Y
9	retrieveRolesForUser	userRepository	y	Y
10	encryptAndSetUserPassword	passwordEncryptor	y	Y

Conclusion: In general the more instance variables a method manipulates, the more cohesive the method is to the class. The table above shows the usage of instance variables in each method and if the method corresponds to class's single responsibility principle. We conclude that this class is having **Model cohesion** while its methods are showing **functional cohesion** as all elements (methods/arguments) correspond to a single well defined task in the problem domain which is dealing with creating, viewing of user details. The class and its methods satisfy the underlying concepts of cohesion which are readable, usable/re-usable and easier to maintain.

1.2 Class Name : femr.ui.controllers.TriageController

Sourcemeter LCOM = 0

Conclusion: Highly Cohesive

Number of instance variables: 6
Number of methods: 3
Instance Variables: encounterServce, PatientService, searchService,photoService, vitalService and SessionService

The instance variable are being used in majority of the methods as shown below.

Sno	method Name	Instance variables	Method uses instance variables?	Method corresponds to class's function
1	IndexGet	VitalService, SearchService, PatientService	Y	Y
2	indexPopulatedGet	VitalService, SearchService, PatientService	Y	Y
3	IndexPost	SessionService, PatientService, PhotoService, EncounterService, VitalService	Y	Y

Conclusion: In general the more instance variables a method manipulates, the more cohesive the method is to the class and hence this class TriageController is highly cohesive.
We conclude that this class is having **Model cohesion** while its methods are showing **functional cohesion** as all its elements correspond to a single well define task which is dealing with creating, viewing and editing of patient details.

Low Cohesive classes/methods

1.3 Class **Name : femr.util.stringhelpers.StringUtils.java**
Sourcemeter LCOM =9
Conclusion: Low Cohesion
Number of instance variables: 0
Number of methods: 11

Sno	method Name	Instance variables	Method uses instance variables?	Method corresponds to class's function
1	isNotNullOrWhiteSpace	None	N	N
2	FormatDateTime	None	N	N
3	FormatTime	None	N	N
4	ToSimpleDate	None	N	N
5	splitCamelCase	None	N	N
6	outputStringOrNA	None	N	N
7	outputIntOrNA	None	N	N
8	outputHeightOrNA	None	N	N
9	outputFloatOrNA	None	N	N
10	outputBloodPressureOrNA	None	N	N
11	isNullOrWhiteSpace	None	N	N

Conclusion: In general the Util classes support DRY (don't Repeat Yourself) principles, they usually fail with SRP (Single Repository Principle) utility classes are not proper objects.Because many other classes will have dependencies on a utility class, any changes to utility classes will have significant ramifications on all of those other classes.
We conclude that this class is having **Separable cohesion** while its methods are showing **logical cohesion**.

1.4 **Class Name : femr.business.helpers.QueryHelper.java**
Sourcemeter LCOM5 =7
Conclusion: Low Cohesion
Number of instance variables: 0
Number of methods: 7

Sno	method Name	Instance variables	Method uses instance variables?	Method corresponds to class's function
1	findPatientWeight	None	N	N
2	findWeeksPregnant	None	N	N
3	findPatientHeightFeet	None	N	N
4	findPatientHeightInches	None	N	N
5	findPatients	None	N	N
6	findCities	None	N	N
7	findPatients	None	N	N

Conclusion: In general the Helper classes support DRY (don't Repeat Yourself) principles, they usually fail with SRP (Single Repository Principle) utility classes are not proper objects; therefore, they don't fit into object-oriented world. Because many other classes will have dependencies on a utility class, any changes to utility classes will have significant ramifications on all of those other classes. The methods findCities and findWeeksPregnant logically do not belong to this class. These methods have coincidental cohesion with the other methods in the class.
We conclude that this class is having **Separable cohesion** while its methods are showing **coincidental cohesion**.

Highly Coupled classes

1.5 **Class Name :** Item Model Mapper:
CBO:38
RFC: 192
Conclusion: High Coupling
The objects cityItem, MissionItem, MissionTripItem,, patientItem, patientEncounterItem, photoItem, PrescriptionItem, problemItem, settingItem, tabItem, TabFieldItem, teamItem , TripItem, UserItem, VitalItem, MedicationAdministrationItem
are tightly coupled because in place of those obects if other objects are required to be used, it would require code changes in the ItemModelMapper Class. Alternatively the ItemModelMapper can use dependency injection by following POJO/POJI model.
Conclusion: Item Model Mapper shows Stamp Coupling and Data coupling

1.6 **Class Name :** HistoryController
CBO:25
RFC: 53
Conclusion: Highly Coupled

Conclusion HistoryController shows increased coupling,icreased interclass dependancies making the code less modular and less suitable for reuse. The code becomes more difficult to maintain since an alteration to code in one area runs a higher risk of affecting code in another area. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding

HistoryController shows both Control coupling and Stamp Coupling

Low Coupled classes

1.7 **Class Name : femr.business.helpers.QueryHelper**

Sourcemeter CBO =4
Conclusion: Low Coupled
The logic is specific to the implementation of the find Patient related attributes for the class is restricted to just that class. So we could change the implementation of any of these classes without having to change the other classes hence it is low coupled
component communication is done via parameters and hence it is message coupling

1.8 **Class Name:** femr.ui.controllers.admin.AdminController
Sourcemeter CBO =3
Conclusion: Low Coupled
The AdminController though is a small class, the amount of unnecessary information it need to know about other design elements are reduced and also that component communication is done via parameters and hence it is message coupling

1.9 Differences between Highly and low Cohesive classes
The highly cohesive classes showed Model Cohesion and their methods showed functional cohesion which is a clear indication of the characteristics of how the methods are grouped to serve a single purpose. The low cohesive classes showed separable cohesion while their methods showed logical and coincidental cohesion which is an indication that the some of the code/methods are spread out into different classes at random. This would lead to code smells and have negative effects software maintenance and future enhancements.

1.10 Differences between Highly and low coupled classes
The high coupled classes showed Stamp and Data Coupling which is better because the data gets passed between classes using parameters or through a passed data structure which may or may not may contain more data than required. The low coupled classes showed message coupling because component communication is done via parameters or message passing as the

objects are decentralized. This would lead to This would lead to code smells and have negative effects software maintenance and future enhancements as it prevents the replacement or changes of components independently of the whole

2. Detecting code smells in fEMR

SNo	Smell Type	Description
2.1	**Code Duplication**	**The smell:** In the ResearchService class, in order to build Secondary Data, the three methods buildVitalResultSet,buildMedicalResultSet and buildAgeResultset used the same duplicated code inside their implementations. **The Why:** This could become messy soon because any of the variables/functions within secondary data building logic needs a change, all the three methods would call for a change too We **agree** that the detected smell is a smell for the above said reason
2.2	**Schizophrenic**	**The Smell:** The public interface of the class LocaleUnitConverter is large and used non-cohesively by client methods i.e., disjoint groups of client classes use disjoint fragments of the class interface in an exclusive fashion. **The Why:** This could become messy soon because any of the variables/functions within secondary data building logic needs a change, all the three methods would call for a change too. We **agree** that the detected smell is a smell for the above said reason
2.3	**God Class**	**The Smell:** The class ItemModelMapper uses many attributes from external classes, directly or via accessor methods and responsible for creating various Items like PatientItem, PrescriptionItem,PatientEncounterItem etc. **The Why:** The class is excessively large and complex, due to its methods having a high cyclomatic complexity and nesting level. For instance, the class manages functionality related to MissionTrip, PatientItem, PrescriptionItem, PatientEncounter etc. The class controls way too many other objects in the system We **agree** that the detected smell is a smell for the above said reason

2.4	**Feature Envy**	**The Smell:** The ResearchFilterItem heavily uses data createResearchFilterItem directly. Furthermore, in accessing external data, the method is intensively using data from at least one external capsule, FilterVieawModel**.** **The Why:** The createResearchFilterItem uses **none** of the attributes that ResearchController defines. We **agree** that the detected smell is a smell for the above said reason

3. Refactoring analysis

3.1 **Code Duplication** - **ResearchService (Automated Refactoring using IDE)**

3.1.1	Detail	In the ResearchService class, Extract refactoring was used to extract the code from the 3 functions buildVitalResultSet, buildMedicalResultSet and buildAgeResultset to build Secondary Data from a common function.
3.1.2	Rationale	As the code is duplicated, expected the IDE to automatically extract the selected code to refactor
3.1.3	Code changes (manual + Automated)	After using Refactor->Extract, the IDE did not really extract the selected code and hence performed refactoring manually

3.2 **Schezophrenic class- LocaleUnitConverter (Automated Refactoring using IDE)**

3.2.1	Detail	In the **LocaleUnitConverter** class, Move refactoring was used move the methods toMetric() and forDuaUnitDisplay() functions to SearchServiceLocaleUnitConverter.java
3.2.2	Rationale	As the move is a simple operation, expected IDE to easily move the functions to a different class.
3.2.3	Code changes (manual + Automated)	After using Refactor->Move, the IDE did not really move the selected code , may be because of ambiguity in function names and hence performed refactoring manually

3.3 **God class ItemModelMapper (Manual Refactoring)**

3.3.1	Detail	From the **ItemModelMapper** class, the methods CreatePatientItem() and CreatePatientEncounter() were moved out to PatientItemModelMapper class. • Created Interface I PatientItemModelMapper with the same method signature. • Moved the imeplementation of methods from ItemModelMapper class to PatientItemModelMapper • In the femr.util.dependencyinjection.modules.MapperModule, created bindings from IIPatientItemModelMapper.**class**) to PatientModelMapper.class
3.3.2	Rationale	The rationale was to move the Patient related functionality to a separate ItemModelMapper class so that the unit will be cohesive and less coupled.
3.3.3	Code changes (manual + Automated)	• At first, we copied the IItemModelMapper class and pasted in the same directory and renamed it to PatientItemModelMapper. • Next ItemModelMapper class was copied, pasted and renamed to PatientItemModelMapper. • Then we searched for the methods that used CreatePatientItem() and CreatePatientEncounterItem() • From the Results of the searched, we've realized that we need to make changes to the femr.business.service.system.PatientService and femr.business.service.system.PatientService • Edited both the files and injected PatientItemModelMapper in the constructor • Searched for itemModelMapper.CreatePatientItem and replaced with PatientItemModelMapper.CreatePatientItem • Searched for itemModelMapper. CreatePatientEncounterItem and replaced with PatientItemModelMapper. CreatePatientEncounterItem

3.4 **GodClass ItemModelMapper (Manual Refactoring)**

3.2.1	Detail	From the **ItemModelMapper** class, the method CreatePrescriptiontItem() was moved out to PrescriptiontItemModelMapper class. • Created Interface I PrescriptionItemModelMapper with the same method signature. • Moved the imeplementation of method from ItemModelMapper class to PrescriptionItemModelMapper • In the femr.util.dependencyinjection.modules.MapperModule, created bindings from IPrescriptionItemModelMapper.**class**) to PrescriptionItemModelMapper.class
3.2.2	Rationale	The rationale was to move the Prescription related functionality to a separate ItemModelMapper class so that the unit will be cohesive and less coupled.
3.2.3	Code changes (manual + Automated)	• At first, we copied the IItemModelMapper class and pasted in the same directory and renamed it to PrescriptiontItemModelMapper. • Next ItemModelMapper class was copied, pasted and renamed to PrescriptionItemModelMapper. • Then we searched for the methods that used CreatePrescriptiontItem() • From the Results of the searched, we've realized that we need to make changes to the femr.business.service.system.MedicationService and femr.business.service.system.SearchService • Edited both the files and injected PrescriptionItemModelMapper in the constructor • Searched for itemModelMapper.CreatePrescriptionItem and replaced with PrescriptionItemModelMapper.CreatePrescriptiontItem

Comparisons of Manual Vs Automated Refactoring

Automated refactoring

Difficulties	• When we tried to use the Refactor→extract method in the ReseachService, IntelliJ did not perform the extraction as expected. We were not very sure if it is an IDE problem or with our understanding. to finish in rush, we ended up finishing the task using manual refactoring. • When we refactor->rename a class's name, eclipse typically will change the filename as well which is not the case with intelliJ and took a while to understand that.
Advantages	• Find and Replace Code duplicates feature is very friendly. Though we did not use the Replace function, we used the find function to be very effective • Encapsulate fields is helpful • Automatically applying changes to method return types, local variables, parameters and other data-flow-dependent type entries across the entire project.
Disadvantages	• Does not support automatic refactoring for code duplication between 2 classes • Code visibility of deeper dependent classes is hard to visualize with automated refactoring