SE 6356

Software Maintenance, Evolution and Re-Engineering

# Assignment #2

# Chelsea Swan

# Leena Varghese

# EMSE - Fall 2016

# PART-1

# Select one or more coupling and cohesion metrics (used by SourceMeter) and analyze the results by following the next steps:

## Select the top two non-trivial classes with the highest cohesion and the two classes with the lowest cohesion. Explain why the selected classes have high/low cohesion. Explain the difference between the classes with highest cohesion versus the classes with lowest cohesion. Identify and explain the type of cohesion that SourceMeter is measuring.

The SourceMeter uses the LCOM1 Chidamber & Kemerer method to identify the cohesion in the class. Using this approach each method in the class is compared to identify if they contain disjoint set of code. If the 2 class access variables that are not common then the variable P is increased by one and if they share a instance variable then the Q is increased by one.

LCOM1 = P - Q , if P > Q :: Indicates a non cohesive class. (low cohesion)

LCOM1 = 0 otherwise :: indicates a cohesive class.(high cohesion)

Higher the LCOM1 value the higher the chance for the flaws in the design and functionality.

As per the tool the least cohesive class is QueryProvider (path: app/femr/business/helpers/QueryProvider.java) with the LCOM1 count =31. On inspecting the class we found that this class was responsible for creating all the queries and it is not a complex class although methods in the class are unrelated they share the common purpose of providing all logic for the search related queries. It indicates logical cohesion.

Per the results of the tool the highly cohesive class with LCOM of 0 were:

Roles(Package:: femr.data.models.mysql) - This class only contains variables but no methods.

IntegrationTest(Package: functional) - all code is commented. So these are not considered for the highest cohesion for the purpose of the assignment.

The main difference between that classes (that are chosen below) that are more cohesive than the classes that are low in cohesion is that the classes that are low in cohesion have a number of unrelated functions that are often complex and difficult to understand. The highly cohesive classes operate on common variables and are easier to understand as they represent a common functional goal.

| Cohesion Type | Class | Why? Depending on the outcome of running the Sourcemeter. | Details |
| --- | --- | --- | --- |
| Highest | EncounterService  (**package::** femr.business.services.system  ) | LCOM=1, In this class most of methods have common variables that are accessed. | This class provides the service layer implementation for the functions that deal with the patient encounter.  Most of the methods operate on the input encounterId and response returnedServiceResponse. The internal method implementation waits on response from other methods that needs to be provided as input to the next in line code. Therefore the cohesion indicated is sequential cohesion. |
|  | VitalService  (**package::** femr.business.services.system  ) | LCOM=1. In this class most of methods have common variables that are accessed. | This class provides the service layer implementation for handling the information pertaining to patient vitals collected.  Most of the methods operate on the input encounterId and response returned is of type ServiceResponse of type List<VitalItem>. The class has an implementation for isMetric method which is could have been avoided.  The internal method implementation waits on response from other methods that needs to be provided as input to the next in line code. Therefore the cohesion indicated is sequential cohesion. |
| Lowest | IndexViewModelPost  ( **package**::femr.ui.models.triage) | LCOM=24 contains only get and set methods for the attributes in the class which operate on different input and output variables. | This class operates on nearly 21 variables and contains get and set operations for these operators only. There is no implementation or modification operation performed on the operators and are grouped together only to provide a object that captures the information from the user interface.  When we view this as a set of unrelated methods we infer this as coincidental cohesion. |
|  | ItemModelMapper  (**package::** femr.common) | LCOM=16, contains methods that do not operate on the same variables. | This class is responsible for creating item objects (common/models) that are visible to the user interface and service layer. The methods are all unrelated and operate on different variables. It is a complex class and is difficult to understand and maintain.  When we view this as a set of unrelated methods we infer this as coincidental cohesion. |

## 2.Select the top two non-trivial classes with the highest coupling and the two classes with the lowest coupling. Explain why the selected classes have high/low coupling. Explain the difference between the classes with highest coupling versus the classes with lowest coupling. Identify and explain the type of coupling that SourceMeter is measuring (based on the selected metrics).

To analyze the coupling in the code we depend on the metric Coupling between objects (CBO) that is provided by the source meter. The metric CBO refers to the count of the number of classes to which the specific class is coupled to.

Another metric is RFC which stands for Response for class indicates the number of methods that can be executed as a result of the a message received by the specific class.

The metrics calculated by the Source meter depends on the Chidamber & Kemerer metric method that primarily relies on the metric CBO for inferring coupling between objects of the classes analysed.

CBO accounts for both kinds of the usages - classes that are called by and the classes that call the specific class being analyzed. The higher the count presented by CBO the more tightly coupled the classes. This means the code is more complex, not reusable and fault prone. Inheritance is also taken as a normal coupling.

For the purpose of the comparison between the classes with highest and lowest coupling presented in the table below, we consider 3 parameters that the tool generates- CBO (primary), RFC and WMC where WMC is the number of methods defined in the class. This is done to contradict the results that when considering the coupling between classes it is necessary to understand the number of methods and class complexity to conclude the coupling between the classes being used for analysis.

| Coupling Type | Class | Source meter measure | Details |
| --- | --- | --- | --- |
| Highest | ItemModelMapper  (**package::** femr.common) | CBO =39  RFC=221  WMC=85  The metric above indicate there are 39 objects that are coupled indicating strong coupling. This might affect 85 methods in all defined in the class and 221 methods could be affected by the changes in the class also indicating the complexity generated by the implemenatation. | The class contains method implementations that are strongly coupled. A number of the functions depends on other classes that directly influence the response returned by the class. The class is difficult to maintain as changes cannot be localized. Any change to the calling classes(external) directly influence the behavior of the method. Methods like ::createPatientItem calls the external PatientItem class,createTabFieldItem calls the external class TabFieldItem which indicate content coupling and createSettingItem which uses switch statements based on the ISystemSetting input to modify the output returned indicate control Coupling. |
|  | SearchService (**package**::femr.business.services.system) | CBO=39  RFC =113  WMC=55  The metric above indicate there are 39 objects that are coupled indicating strong coupling. This might affect 55 methods in all defined in the class and 113 methods could be affected by the changes in the class. | This class has complex logic that is strongly coupled. The method ServiceResponse<List<PatientItem>> retrievePatientsForSearch(Integer tripId) is an example of strongly coupled method used in the class. It calls nearly 18 external classes. Class like PatientItem directly influences the response of the method. External methods used in the class like addError modifies the message sent back to the calling class. It indicates both content and control coupling. |
| Lowest | HomeController  (**package::** femr.ui.controllers  ) | CBO=2  RFC =3  WMC=3  The metric above indicate there are 2 objects that are coupled indicating weak coupling.  This might affect 3 methods in all defined in the class and 3 methods could be affected by the changes in the class. | This class is a non trivial class which is relative complexion implementation. It uses 2 external classes ISessionService and CurrentUser. In the index() method the response from the SessionService assigns the value to the CurrentUser instance which in turn influence the behavior of the value returned by the method. This represents control coupling. |
|  | ManageViewModelGet (**package::**femr.ui.models.admin.users) | CBO=2  RFC =2  WMC=2  The metric above indicate there are 2 objects that are coupled indicating weak coupling.  This might affect 2 methods in all defined in the class and 2 methods could be affected by the changes in the class. | This class is a non trivial class which is relative complexion implementation. It uses external classes UserItem and the list object. The methods have clear implementation and are weakly coupled. It indicates data coupling where only the required data is passed between methods. |

# PART-2

# Detecting code smells in fEMR

## Select 2 smelly classes/methods detected by InCode for each of 3 distinct type of smells (e.g., God Class) – total of 6 smelly classes/methods. Then, for each smelly code component:

## Briefly describe the smell - i.e., the class, methods, attributes, etc. involved in the smell.

## Explain why the flagged class/method is smelly (be specific).

## 3. Do you agree that the detected smell is an actual smell? Justify your answer.

| Bad Smell Type | Class/Method |
| --- | --- |
| **Bloaters** | Class Name : MedicationService  The primary responsibility of the class is provide information for anything related to both medications and prescriptions. It is not responsible for anything to do with the quantities being stored or dispensed (this is the role of the Inventory service).  It is a large class with average complexity, that is strongly coupled to other classes/ interfaces.  It contains 2 data clumps that is code that changes together.  Why the code is smelly?  Data Clumps: The method has a long parameter list, and its signature or a significant fragment thereof is duplicated by other methods. This indicates that for 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. The methods affected by the data clumps are:  Method Name: [createPrescriptionWithNewMedication](http://1331_gotopagelink_/)  Parameter list : String medicationName, Integer administrationId, int encounterId, int userId, int amount, String specialInstructions  Method Name: [createPrescription](http://1281_gotopagelink_/)  Parameter list : int medicationId, Integer administrationId, int encounterId, int userId, int amount, String specialInstructions  Do we agree with the result from inCode?  We do agree with the result from the inCode tool that it is likely to cause data duplication, complexity and unmanageable code, the current implementation when we go through the code is not very large or complex and the class is not disjoint in the functionality. We were not able to seem the issues of having such as duplication of code. |
| Class Name : ItemModelMapper  The primary responsibility of the class is to create item objects required for user interface and service layer. The tool categorizes this class as a God Class.  It is a large class with multiple complex,unrelated responsibilities.  It contains 3 data clumps that is code that changes together.  Why the code is smelly?   1. The class is large and complex. It uses attributes and methods of several external other classes.   ex: Method : createPatientItem which requires a long list of input parameter that directly influences the attribute of external classes called within the class.  2) Data Clumps: The method has a long parameter list, and its signature or a significant fragment thereof is duplicated by other methods. This indicates that for 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 Name: createTabFieldItem  Parameter list : String name, String type, String size, Integer order, String placeholder, String value, String chiefComplaint, boolean isCustom, String userNam  Method Name: createPatientItem  Parameter list : 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, Integer isBirthDateCorrect  Do we agree with the result from inCode?  Yes, we do agree with the result from the inCode tool because when we go through the code it is very clear how large, complex and disjoint the class is. We were able to seem the issues of having such a large and complex class implementation such as duplication of code and non cohesive  and tightly coupled code. |
| Couplers | Class Name : SearchService  The primary responsibility of the class is to obtain the details associated with the Patient that were requested using search feature.  It is a large class with multiple complex,unrelated responsibilities.  It contains 1 Message chain that is code that changes together.  Why the code is smelly?   1. The class is large and complex. It is strongly coupled and is not easy to modify. It uses attributes and methods of several external other classes.   ex: Method : retrievePatientsForSearch is large (around 90 lines of code) and complex.  2) Message chain: The method has a long parameter list, and its signature or a significant fragment thereof is duplicated by other methods. This indicates that for 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 Name:  [retrievePatientsForSearch](http://5320_gotopagelink_/) (Integer tripId)  Parameter list : Integer tripId  Message chain:  StringUtils.isNotNullOrWhiteSpace(***missionTrip.getMissionCity().getMissionCountry().getName()))*** {  allPatients = QueryHelper.findPatients( **patientRepository,missionTrip.getMissionCity().getMissionCountry().getName()**  Do we agree with the result from inCode?  Yes, we do agree with the result from the inCode tool because when we go through the code it is very clear how large, complex and appears to handle too many responsibilities. There are at least 5 filters used to retrieve the result and each is chained as an input for the next executing code. |
| Class Name : ItemModelMapper  The primary responsibility of the class is to create item objects required for user interface and service layer. The tool categorizes this class as a God Class.  It is a large class with multiple complex,unrelated responsibilities.  It contains 4 feature envy and 1 message chain.  Why the code is smelly?   1. Message chain: The method uses one object to access another object, then uses the obtained object to access another object, and so on, all objects having different types.   ex: Method : createMissionTripItem :  missionTripItem.setTripCountry(missionTrip.getMissionCity().getMissionCountry().getName())  2) Feature Envy: The method heavily uses data from one or more external classes, directly or via accessor operations. Furthermore, in accessing external data, the method is intensively using data from at least one external capsule. Also the createPatientItem does not define any attributes.  Method Name: createPatientItem  Parameter list : 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, Integer isBirthDateCorrect  External class used: PatientItem  Affects: The method is tightly coupled and has low cohesion as it does not use the internal attributes and external attribute dependency on PatientItem is high.  Method Name: createTabFieldItem  Parameter list : String name, String type, String size, Integer order, String placeholder, String value, String chiefComplaint, boolean isCustom  External class used: TabFieldItem  Affects: The method is long, complex, tightly coupled and has low cohesion as it does not use the internal attributes and external attribute dependency on TabFieldItem is high.  Method Name: createSettingItem  Parameter list : List systemSettings  External class used: SettingItem  Affects: The method has low cohesion and moderately complex.  Method Name: createTripItem  Parameter list : String teamName, String tripCity, String tripCountry, Date startDate, Date endDate) : External class used: TripItem  Affects: The method has low cohesion and moderately complex.  Do we agree with the result from inCode?  Yes, we do agree with the result from the inCode tool because when we go through the code it is very clear how large, complex and disjoint the class is. We also observer that the chances of the code failure for message chain is very high and since exception is not handled within code and were able to seem the issues of having such a large and complex class implementation such as duplication of code, non cohesive. and tightly coupled code. |
| Dispensibles | Class Name : LocaleUnitConverter (util/calculations/dateUtils.java)  The primary responsibility of the class is to provide methods to facilitate date related computations.  It is a large class with multiple complex,unrelated responsibilities.  It contains 1 external duplication that is code that changes together.  Why the code is smelly?   1. large and complex. It is strongly coupled and is not easy to modify. It uses attributes and methods of several external other classes.   2) Duplicated /dead code The method has a long parameter list, and its signature or a significant fragment thereof is duplicated by other methods. This indicates that for 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.  ex: Method : getMeters(Float Feet, Float Inches)  Do we agree with the result from inCode?  Yes, we do agree with the result from the inCode tool because when we go through the code it is very clear how large, complex and appears to handle too many responsibilities. There are at least 5 filters used to retrieve the result and each is chained as an input for the next executing code. |
| Class Name : ResearchService  It is a large class with multiple complex,unrelated responsibilities.  It is affected by 1 feature envy and 3 internal duplications.  Why the code is smelly?   1. The class is large and complex. It is strongly coupled and is not easy to modify. It uses attributes and methods of several external other classes.   ex: Method : buildVitalResultSet is large (around 200 lines of code) and very complex.  2) Duplicate code: This method has duplication with a method from an external class.  Internal Class Duplication:  buildVitalResultSet(List encounters, ResearchFilterItem filters) : ResearchResultSetItem  buildHeightResultSet(List encounters, ResearchFilterItem filters) : ResearchResultSetItem  Do we agree with the result from inCode?  Yes, we do agree with the result from the inCode tool because when we go through the code it is very clear how large, complex and appears to handle too many responsibilities. Because it is handling so many responsibilities the code is not easy to change or manage. It is tightly coupled and error prone. The methods are very large and not easy to understand which in terms results in code duplication. |
|
|
|
|

# PART-3

# Refactoring analysis

## Describe the refactorings you made and compare both manual and automated refactoring. For each one of the smells removed:

## 1. List and describe in detail the refactorings (i.e., the code changes) used to remove the smell.

## 2. Give the rationale of the chosen refactoring operations.

## 3. Explain what code changes you had to do manually, in addition of the changes performed with IntelliJ’ support.

|  |  |  |
| --- | --- | --- |
| Refactoring | MANUAL | AUTOMATED |
| Bad Smell :: Dispensibles | | |
| Class Name | LocaleUnitConverter (util/calculations/dateUtils.java) | Class Name : ResearchService |
| Refactoring | getMeters(Float Feet, Float Inches)   1. checked the code to find any references to the method 2. IntelliJ displayed prompt to notify the method was not used. 3. Removed the used method. | There were 2 methods that was using duplicate code within this class:  buildVitalResultSet(List encounters, ResearchFilterItem filters) : ResearchResultSetItem  buildHeightResultSet(List encounters, ResearchFilterItem filters) : ResearchResultSetItem.  Manual: Identifying the code fragment that can be separated without affecting the code integrity with minimal change.  Automated:   1. Once the code fragment that can be refactored was identified, we needed to extract a method could be commonly used by the 2 functions. 2. We use the IntelliJ refactoring feature to extract a new method. 3. The method name for the new method is getSecondaryData(ResearchFilterItem filters, ResearchResultSetItem resultSet, IPatient patient, ResearchResultItem resultItem, IResearchEncounter encounter) 4. We call this common method from both the classes. |
| Rationale | The method was a simple implementation overload to getMeters(Integer, Integer).  In order to remove the external duplication we would either need to comment this code or remove it.  Since the method was not being used it and since this was seen as a bad smell we decided to remove this code. | Both the methods were large and were using the duplicate code. The best approach to remove the bad smell is by extracting this to a new method. |
| Advantage | Since the change was local and not complex, there were no issues with dependencies. So the manual change was the best approach. | Since the change was local and but complex, we needed to detect dependencies in any part of the code. So the automated refactoring helped with extracting the necessary input and output parameters. |
| Disadvantage |  | Requires adequate testing to ensure that the change was accurately analyzed and implemented. |

|  |  |  |
| --- | --- | --- |
| Refactoring | MANUAL | AUTOMATED |
| Bad Smell :: Bloaters -> Data Clumps | | |
| Class Name | Class Name : MedicationService | Class Name : ItemModelMapper |
| Refactoring | 1) Method Name: [createPrescriptionWithNewMedication](http://1331_gotopagelink_/)  Parameter list : String medicationName, Integer administrationId, int encounterId, int userId, int amount, String specialInstructions  Manual:   * We create a new class with all the attributes that were the inputs to the method- [createPrescriptionWithNewMedication](http://1331_gotopagelink_/)- InputPrescriptionForNewMedication * The class name : InputPrescriptionForNewMedication   **public** ServiceResponse<PrescriptionItem> createPrescriptionWithNewMedication(InputPrescriptionForNewMedication inputPrescriptionForNewMedication)  2) Method Name: [createPrescription](http://1281_gotopagelink_/)  Parameter list : int medicationId, Integer administrationId, int encounterId, int userId, int amount, String specialInstructions  Manual:   * We create a new class with all the attributes that were the inputs to the method- [createPrescription](http://1281_gotopagelink_/) * The class name : InputPrescription inputPrescription   **public** ServiceResponse<PrescriptionItem> createPrescription(InputPrescription inputPrescription) | 1) Method Name: createTabFieldItem  Parameter list : String name, String type, String size, Integer order, String placeholder, String value, String chiefComplaint, boolean isCustom, String userNam  Automated:   * We used the IntelliJ refactoring feature to extract a new parameter object for all the input parameters. * The parameter object name : InputTabFieldItem   public TabFieldItem createTabFieldItem(InputTabFieldItem inputTabFieldItem)   * All the methods or classes that referred to this method were changed to reflect this change. The input parameters provided by each method was replaced by the Parameter object.   2) Method Name: createPatientItem  Parameter list : 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, Integer isBirthDateCorrect  Automated:   * We used the IntelliJ refactoring feature to extract a new parameter object for all the input parameters. * The parameter object name : inputPatientItem * The input parameters for this method was replaced with the following: **public** PatientItem createPatientItem(InputPatientItem inputPatientItem) * All the methods or classes that referred to this method were changed to reflect this change. The input parameters provided by each method was replaced by the Parameter object. |
| Rationale | Common refactoring permitted for resolving the issue with Data Clumps are :  Extract class,Move field /method,Introduce parameter object. | |
| Advantage | When going through the code we checked if any related classes could be used in place of creating a new class for the purpose of collecting input. This can ensure that the code is not getting duplicated. | It is a very quick and clean method to extract a parameter object as required when trying to extract a new class to reduce data clumps.  It also automatically detects any reference to the method and changes the reference to call the input parameter that was changed. |
| Disadvantage | It is a time consuming and unnecessary step to manually refactor the code for such a replacement as this requires manually changing each reference to this method that was changed, which is time consuming and error prone. | It is essential to manually understand and analyse whether a preexisting class can be used in place of adding a new class. Otherwise,this can lead to duplicate code and increased coupling which is not desirable. |