Assignment #2

SE 6356 Software Maintenance, Evolution and Re-Engineering
Due date: 11/17/2016
Team 2: Michelle Jones and Rangy Samuel

Analyzing fEMR's cohesion and coupling

1. We decided to examine EncounterService, InventoryService, StringUtils and QueryHelper for cohesion. The metrics we were analyzing against was Lack of Cohesion of Methods 5 (LCOM5). The table below lists the classes and their LCOM5 score.

Class	LCOM5
femr.business.services.system.EncounterService	1
femr.business.services.system.InventoryService	1
femr.business.helpers.QueryHelper	7
femr.util.stringhelpers.StringUtils	9

One of the reasons that EncounterService and InventoryService have such a low score is because their functionality is defined and limited to a specific object. These two classes aren't being initialized or used throughout the code. Instead they are isolated to a controller class. One of the noticeable attributes about the Service classes is that they are only manipulating one object type, although some are the interface. On the other hand, StringUtils is a unique case because it is designed to be called by many classes regardless of their functionality. We would argue that StringUtils does a great job of isolating the manipulation and formatting of strings to one class, instead of being spread out and repeated across multiple. But, because so many other classes are relying on it to format strings and it isn't be treated as a Static class, those other classes each have to initialize a StringUtils object before being able to use it. And, if someone were to change the return values for StringUtils to include a space or new line character, every class that used StringUtils would have to be updated to handle this change. QueryHelper has a high LCOM5 score with a value of 7. This is because the query is doing one job functionally (searching), but doing it across multiple data types. The class is responsible for gathering data from Patient, MissionCity and Repository objects in various methods.

2. We decided to examine ItemModelMapper, DataModelMapper, SearchController, and SessionService for coupling. Specifically, we evaluated classes based on their Number of Outgoing Invocations (NOI). For a method, this refers to the number of directly called methods. For class, this refers to the number of directly called methods of other classes, including method invocations from attribute initializations. The table below lists the classes and their NOI number. We decided to evaluate classes with a NOI higher than 10, because we wanted to see a difference between classes that had a high NOI vs a low NOI (and we felt that a number lower than 10 wasn't really going to show a comparable class).

Class	NOI
femr.common.ItemModelMapper	197
femr.data.DataModelMapper	101
femr.ui.controllers.SearchController	22
femr.business.services.system.SessionService	21

ItemModelMapper had the highest NOI of all of FEMR. This class is responsible for mapping data to multiple objects, so it makes sense it has a high NOI. But, this class is also highly dependent on the objects it initializes. For example, ItemModelMapper required a change in FEMR-137 when we updated the patient to capture whether the birthdate was an estimation or approximation. The method createPatientItem in ItemModelMapper is taking multiple inputs and creating a new object from them. It also has logic in place to check for invalid input, using none other than the StringUtils class. Personally, we would argue this is lowering the cohesion of the class by having it do data validation on top of data mapping, but we understand why the class was coded to have that check in place (and of course we are more concerned with its coupling with regards to this question). The class is greatly dependent on the classes it initializes, especially if the class changed requires an additional input for object instantiation. This could be mitigated by having multiple object creation methods, but that also leaves open the possibility for a default value to be stored in an object when it should be initialized. The DataModelMapper class was the second highest, with almost half the NOI score than ItemModelMapper. DataModelMapper is slightly different than ItemModelMapper because it is grabbing and object that already exists and then setting a value within that object. This class is necessary for updating an object after initialization and also suffers from the high coupling mentioned with ItemModelMapper. If you have a new field in an object that DataModelMapper manipulates, DataModelMapper will most likely have to be updated to also manipulate that field with a getter/setter mentality. On the opposite side of the spectrum is SearchController and SessionService. SearchController has a lower coupling than ItemModelMapper and DataModelMapper, but it also has a different purpose. Where the ModelMapper classes were focused on initializing an object with data, SearchController is concerned with determining whether data already exists. It is dependent on the objects in the database to perform this search, so it would have to be updated if those fields being searched upon were changed. But, hypothetically if the patient class was updated to have a title, this wouldn't necessarily cause the method doesPatientExist to be updated. It could operate as is just fine. SessionService has a small NOI of 21, which we attribute to it being a very specialized class. The only responsibility this class has is creating, retrieving or invalidating a session. It differs greatly from SearchController because the class is focused on determining if the current user is allowed in the system.

Detecting code smells in fEMR

1. God class MedicalController with a severity of 10. The class is currently at 536 lines of code and worse has whole methods commented out rather than removed. It is highly dependent on the objects it uses, often using getters and setters to manipulate the data. There are calls for or sets external attributes over 70 times in this one class! Because of its complexitiy and high coupling/low cohesion, we agree the class is a bloated god class.
2. God class ItemModelMapper with a severity of 10. With 610 lines of the code, the class has grown to a massive size. Unfortunately, the very function the class is supposed to provide has caused it to become bloated. The class is designed to create and initialze objects, but it also now is calculating values. For example, createMedicationItem should only create a medication item with the values supplied, but it is also responsible for determining if a medication is deleted as well as the generic strenght of the medication. Because of this, we agree that the class is a bloated god class.
3. Schizophrenic class dateUtils with a severity of 3. Of the 11 public methods this class has, 5 are never used in the system. The remaining 6 methods are used in 5 other classes, with

ItemModelMapper using 4. Considering the class is supposed to be a Utility class, it seems hindered with such specific date formats. The methods are necessary for formatting the date, but we don't understand why you can't have one common format (2 at most)? Wouldn't it be easier to always use the international date format as the standard? What does the rest of the world do? On top of that, having the class return Integers, Strings, DateTime and Floats for a date seems overly complicated. Yes the class is schizophrenic. It keeps talking to nobody...

4. Schizophrenic class LocalUnitConverter with a severity of 1. So the class is converting from Imperial to Metric, but it is doing so for length measurements as well as temperatures. Which, aren't really related to eachother.... other than the whole converting. On top of converting, the class is overwritting object values in the method toMetric(), rather than just returning a result and letting the other class handle the values. For a class that is supposed to be converting, altering a Patient object seems out of place. This class does have issues, but we don't think schizophrenic is the correct smell. Instead, we feel that feature envy best describes the class. Converting is fine, but converting and altering the patient object as well as calculating Celcius from Fahrenheit is just to much.
5. Data class PatientItem with a severity of 4. The class doesn't have complex methods which is good. It only contains getters and setters (yay for encapsulation!). The problem arises when you examine the constructor for the class, which requires nothing and then realize you now have to set 19 separate variables in order to finish creating a patient. We feel that the class could be abstracted further for certain items, such as address. Instead of having 2 strings, one for address and another for city, an Address class could be created that contains not only the address and city, but the zip code (or lat/long). The same could be done with age and height, with the Age object allowing either a year/month combo or a birthdate and then Age would automatically calculate the fields, rather than making another call to another class. And having the Height object covert from Imperial to Metric would be best because the value of feet/inches wouldn't have to be saved as an int, thus helping the LocalUnitConverter class because it had problems with precision when the value was an int instead of a float. Yes, the class requires setting way to much data before an object is fully initialized. And allowing the default constructor is fine, but at least have a generic "John Doe" to fill in the values.
6. Data class PrescriptionItem with a severity of 3. The class is very similar to PatientItem, because it keeps its fields private and uses encapsulation to prevent other classes from directly changing the values. And 14 of the 15 private variables are not complex objects, but instead integers, floats, booleans and strings (String is a complicated class in its own right, but the way this class is using the String isn't). Again, many of these private variables could be abstracted out into objects, like Patient, Medication and Prescriber rather than the 15 variables currently used. Yes the class is a sever data class.

Refactoring analysis

1. Manual Refactoring of PatientItem
    a. PatientItem was showing up as a data class in inCode. In order to remove the smell, I decided to abstract the name, address and measurements for the patient. Multiple strings for the first/last name where replaced with a Name object which houses first and last name. Right now it doesn't seem to efficient, but this name class could be updated to contain title and be used generically for all people. That way, you don't have to have a patientFirstName, patientLastName, doctorFirstName, doctorLastName string...

    instead you just need a patientName and doctorName! I also created an Address object that contained the strings for city and address. Again, this class can be updated to include more detail, such as lat/long or country. It was just a starting point. The getters and setters were updated for the code. Finally, a new Measurements class was created to house the height and weight of a person in metric and imperial measurements. In all, 10 variables were replaced with 3. A nice way to streamline if I do say so myself!

    b. The rational behind this change was to try and simplify the PatientItem object. It has about 20 separate fields that each had their own getters and setters. Abstracting the name, address, and measurements out seemed like an obvious choice.
    c. I had to create a Name, Address, and Measurements class with getters and setters, update the PatientItem class to set those values with it's own getters and setters. Then I tried compiling the code so it would show me the errors. Once the errors popped up, I was able to update PatientService.java, ItemModelMapper.java, PDFController.java, TriageController.java, SearchService.java, and updated PatientItemTest.java as well as the test case.
2. Manual Refactoring of LocalUnitConverter
    a. So this class had the most refactoring for one very simple reason, it was to specific for a utility class that converts data. Whoever submitted the code changes for femr136 decided that the class converts from imperial to metric by having a PatientItem passed to it.... So I removed the methods in their entirety. The good news is the actual conversion methods were already in the code, but it was a weird give me the Patient first so I can calculate then return the patient, rather than give me the patient height and here is the metric height. This wasn't the only change I made, and to be honest it feels like I completely redid the work submitted for femr-136.
    b. The rational behind this change was that a utility class should not be dependent on a complex object like PatientItem, especially if the function is merely converting from imperial to metric and vice versa.
    c. Manually removed all traces of passing a PatientItem/returning a PatientItem from LocalUnitConverter, updated ItemModelMapper.java to calculate the values for metric based on the inputted imperial values as well as updating the test case.
3. Automated Refactoring of PrescriptionItem
    a. Based on the refactorings already done for PatientItem, the prescriber first and last name was replaced with a Name object (manually) and I updated the getters and setters for it. From there I had IntelliJ remove unused imports.
    b. Rational was again to try and simplify the data needed for the object. It seems logical to have a name object rather than individual fields.
    c. Updated the PrescriptionItem.java and test class.
4. Automated Refactoring of dateUtils
    a. DateUtils class had a lot of repeated code within the methods that was used to initialze an object. For example, there were instances of needing to compare the difference between two dates. Originally this took 3 lines to check if null, 3 lines to initialize the objects, 1+ line(s) to initialize an object by performing the test, and then return the result. Through refactoring, I was able to take the 7+ lines and redo it down to one line using in-line refactoring. This made the code much cleaner to read. Also, one method

was returning a new instance of another method, and that was it. I decided to remove the method entirely and refactor any code that originally called the method to instead create the instance itself. It adds extra code in all, but the method was worthless because it was just another loop to jump through to get the end result.

b. The rational behind the refactoring was that the class was to complicated to read for being a date utility class. Methods were repeating the same 7 lines before doing a check. This was excessive and bulky. The class went from being 201 lines of code down to 144 lines. So almost 25% of the code in the class was repeated information. Now the code looks much cleaner and easier to read.
c. I was able to automatically removed unused imports after removing the unnecessary method getCurrentDateTime(). That was the only automatic automation. I manually removed the method and updated LogicDoer.java, EncounterService.java, MedicationService.java, TabService.java, UserService.java, DataModelMapper.java, DatabaseSeeder.java, and (of course) dateUtils.java.