# Coupling and Cohesion

## Cohesion metrics

SourceMeter defines its cohesion metric as "number of functionalities of the class," that is, "how many coherent classes into which the class could be split." Since a class should only have one main functionality, ideally the number of coherent classes into which you can be split should be 0. So for this metric, lower numbers translate to higher cohesion and higher numbers translate to lower cohesion. As such, this metric is also measuring functional cohesion, so these classes will all be explored in their relation to functional cohesion.

The class with the lowest cohesion by far was ItemModelMapper.java. With a metric score of 16, InCode thinks this class could be ideally split into 16 distinct coherent classes. This class is called by all over the source code. This class is referenced by ConceptService.java, EncounterService.java, InventoryService.java, MedicationService.java, MissionTripService.java, PatientService.java, PhotoService.java, SearchService.java, TabService.java, UserService.java, and VitalService.java. Its main purpose seems to be taking in the various attributes of objects from the view, and populating those objects while performing some calculations and adjustments to the data along the way, returning the objects. This class has such low cohesion because it performs a series of completely unrelated functions – while there is maybe some logical cohesion to having a class hold a series of methods that converts attributes into objects, none of the objects are related and are all in different domains – in fact, it seems to touch just about every domain in the FEMR system. In essence, ItemModelMapper is a middle man class with a lot of feature envy.

StringUtils.java is the next least cohesive non-trivial class, with a metric store of 9. This skipped Repository.java, with a metric of 10, because we believe Repository to be a trivial class. StringUtils is such a non-cohesive class because it's a Utility library, and therefore suffers from coincidental cohesion – methods are grouped there just because they couldn't find a better location for them. These libraries, while useful, especially in std libraries like Date, String, and Math libraries, nevertheless have very low cohesion due to their very nature. In the case of StringUtils, it seems to just have various operations that tenuously relate to strings, and has operations ranging from null conversions to date/time operations to various string formatting and parsing.

PhotoController.java is one of the non-trivial classes that has a metric score of 0, or perfect cohesion. The reason for this seems to be that the PhotoController class only calls local methods and acts upon a single local member object (photoService) in its three functions. This is the highest form of cohesion (functional) because it only deals with a single task – getting and showing photos. It's also informationally cohesive because all functions use the photoService object.

DataModelMapper is one of the non-trivial classes that has a metric score of 1. There were a large number of classes with a metric score of 0, but we considered these to be trivial or seemed to not be Java classes. At first DataModelMapper was a surprising choice for a high cohesion score, because of how many different member variables it owned and operated on; however, it seems to be very cohesive in that it only serves one purpose: creating the model objects given some data. In that respect, it is a cohesive class. The only problem and the reason for its score of 1 is the function updatePatientEncounterWithDiabetesScreening; this method definitely seems out of place in this class, as even the comment at the top of the class says "*Responsible for creating model objects (data/models)*", which this method doesn't do. My guess would be this method didn't find a home anywhere else, and was just stuck on here after the fact because the author couldn't or didn't want to try to find a better fit for it or create a new cohesive class for just this function.

The difference between ItemModelMapper and StringUtils vs PhotoController and DataModelMapper is that PhotoController and DataModelMapper (mostly) only perform action on their own attributes (e.g. local member variables and objects), while ItemModelMapper and StringUtils perform actions on other objects passed in to their methods – in fact, this seems to be the sole purpose for ItemModelMapper, which explains its extremely low cohesion score. Interestingly enough, ItemModelMapper actually calls StringUtils quite a bit, which means that not only are ItemModelMapper and StringUtils uncohesive, but they're intercoupled as well.

## Coupling Metrics

We went with a combination of different coupling metrics, including Coupling Between Object classes (CBO), CBO Inverse (CBOI), Number of Incoming Invocations (NII), and Number of Outgoing Invocations (NOI). CBO represents the number of directly used other classes, either through inheritance, function call, type reference, or attribute reference. CBOI is the opposite – it is the number of other classes which directly use this class. NII is on the method level, and represents the count of other methods and attribute initializations that directly call this method. NII corresponds to CBOI. NOI is the number of directly called methods, and corresponds to CBO.

One of the highest coupled classes in the system comes as no surprise – ItemModelMapper, with the highest NOI value of 197, and the third highest CBO value in the entire system. ItemModelMapper has such high values in these scores because they count the number of directly used other methods and classes, and ItemModelMapper does just that – touches a lot of other methods and classes in its many disjoint methods. It makes sense too – classes with high coupling tend to have low cohesion as well and vice versa. This class definitely suffers from doing too many things and as such is going to be very change prone, as many of the classes it uses changes so it too will have to change. We even modified ItemModelMapper ourselves while working on one of the FEMR defects in the first assignment (and probably contributing to the coupling...). One of the types of coupling in ItemModelMapper is Control coupling, where the internal paths of the methods are often dependent on external functions such as StringUtils.isNullOrWhiteSpace. The largest source of coupling though is Data coupling, because most of the work involves passing parameters to the objects it creates.

Another of the highest coupled classes in the system happens to be DataModelMapper – one of the most cohesive classes in the system as well. This we found was interesting, because you typically think high cohesion correlates with low coupling and vice versa, but apparently not always, and not in this case. DataModelMapper is cohesive in the sense that it only does a single thing – creating objects, populating their attributes and returning them. Because of this though, it is highly coupled to the objects it creates via data coupling. As it creates 31 of these objects, it is therefore highly coupled.

One of the least coupled classes is FEMRAuthenticated. This class also happens to have a cohesion score of 1, so it's an example of both a low coupled and highly cohesive class. It is small (only 2 methods) which helps for both metrics. The coupling is low because it doesn't reference any other class except Date, which it uses a few times. This coupling is a type of Data coupling.

Another low coupled class is CSVWriterGson, which has a CBOI of 1 and a NII of 1 and a 0 for the other coupling metrics. This class has a cohesion score of 1 as well. The coupling is so low because this class doesn't use anything else, and itself is only used once, by ResearchService, which creates it and calls writeAsCsv. This is Data coupling because it just passes in some parameters. CSVWriterGson is highly cohesive partially because it doesn't use any other objects, so its cohesion and low coupling go hand in hand.

The difference between the high coupled classes (such as ItemModelMapper and DataModelMapper) and the low coupled classes (such as FEMR Authenticated and CSVWriterGson) are that the high coupled classes almost act as containers for many different object types, serving to create and populate the objects similar to the factory pattern. This helps a class like DataModelMapper achieve high cohesion, but it comes at the price of low coupling. On the other hand, despite its high cohesion and low coupling, CSVWriterGson doesn't interact with any other classes and is very niche, so its importance in the system is limited.

# Detecting code smells in fEMR

## Message Chains

InCode found that ItemModelMapper was the worst offender for message chains in FEMR– a chain 4 objects deep. Message chain code smells indicate unnecessary coupling and dependence on object hierarchy. If any part of the chain changes, the ItemModelMapper class will have to change as well. Typically we start to look at refactoring any message chain that gets 3 objects deep, and 4 is definitely too deep, so we agree with InCode on this one. This instance comes from the method createMissionTripItem, where it calls the following: missionTrip.getMissionCity().getMissionCountry().getName()

This couples the classes MissionCountry and MissionCity to the ItemModelMapper class where no coupling necessarily needs to exist. All the calling code needs is a string name of the country, it doesn't necessarily need to access MissionCountry and MissionCity to retrieve that.

The second worst offender for message chains found by InCode was actually the same chain, used in a different location – inside MissionTripService::createNewTrip, which calls missionTrip.getMissionCity().getMissionCountry().getName(). InCode gives this chain a severity of 3, which we don't understand given it's the exact same chain used in ItemModelMapper (why isn’t it 4?). Regardless, this smell is a good candidate for refactoring because it is showing up in multiple places and fixing it once will improve several different areas of the code.

## Feature Envy

InCode found ItemModelMapper to be one of the worst offenders for feature envy, with a severity of 9. It has a method, createPatientItem, which does not act upon any local data at all, but only on methods of the PatientItem class. Typically, data and the operations on that data are encapsulated in the same class; in this case, all the operations on the PatientItem data are performed outside of the PatientItem class in the ItemModelMapper class. We agree with InCode's finding in this case, createPatientItem seems like a method that belongs in PatientItem itself, not in ItemModelMapper. There doesn't seem to be a real reason for ItemModelMapper to map each field – this is easily fulfilled by either a constructor or a method residing in PatientItem which ItemModelMapper can call. This will have the benefit of keeping the data and the function that operates on that data next to each other in the same class, so that for instance, if a piece of data is renamed or moved around, ItemModelMapper is not affected by the change whereas it would be as currently written.

The other worst offender for feature envy has the same severity of 9: ResearchController::createResearchFilterItem. This method suffers from the same issue ItemModelMapper::createPatientItem suffers from – it is a function that solely exists to populate another object with data using that objects own mutators. For all the same reasons, this causes unnecessary coupling between ResearchController and the class being populated, ResearchFilterItem, and thus we agree with InCode on this one. In order to reduce coupling, all the data should be pulled into ResearchFilterItem itself, either through a constructor or a mutator method.

## God class

God classes are classes that do too many things. InCode detects God classes as a combination of the cyclomatic complexity and nesting level of the methods, lack of encapsulation, high coupling and low cohesion. MedicalController has a severity of 10 for the God class smell. The MedicalController class manages, or controls, the information for a number of other smaller classes that are primarily data-driven. For example, the editPost method of this class controls data for the IEncounterService, IPhotoService and IMedicationService. Similarly, the editGet method takes ISearchService, IEncounterService, IVitalService, ITabService and IPhotoService to put together a patient’s profile. We agree with InCode; this class accesses data and methods of too many other classes. For instance, method editGet calls 31 methods of different classes, which seems a little excessive. MedicalController seems to be a large God class because it manages a lot of detail that it doesn't necessarily need to manage; for instance, it has a method getPatientEncounterVitals where it seems to populate a map with the data from an UpdateVitalsModel object. In this case, it can reduce its coupling and God class score by allowing UpdateVitalsModel to populate the map with its own data and return it. One major step in reducing this class’s God class status will be undoing a lot of the feature envy happening here.

The second God class found by InCode also has a severity level of 10, and is ItemModelMapper.

ItemModelMapper suffers from many different code smells, which all combine to help it have such a high God class severity. In addition to the methods caught by previous smells (such as createPatientItem), there are other methods with similar issues, such as createMedicationItem, createMissionTripItem, and so on. Most of these functions simply are the same as createPatientItem – creating an object and populating it by using its mutators. Unfortunately this causes the number of external methods to be very high, which is counted in determining a God class smell. It would be less coupled to move those mass populate methods into their respective classes, for the same reasons as listed in MedicalController.

# Refactoring analysis

Note: for the refactoring analysis, Alex and Daniel had already done 5 of the 6 refactorings when Jimmy dropped, so we just left them in. The one missing refactoring is Automated: God Class for MedicalController, which we did not get to.

## Automated Refactoring: Feature Envy

Class: ItemModelMapper

Method: createPatientItem

Smell: FeatureEnvy

Steps

1. An ItemModelMapper test was created to ensure createPatientItem still succeeded with correct data all populated.

2. Ensured test passed.

3. First, highlight the block of text in ItemModelMapper::createPatientItem that should be extracted.

4. Right click in Intellij -> Refactor -> Extract -> Method.

5. Choose the name "populate".

6. A new method will have been created in the same class, copy it, and paste it into PatientItem.

7. Change the return type to void.

8. Copy and paste the method signature into the new method in PatientItem.

9. Remove "ageClassification" from the method signature, as it is not being used and there is no member variable to be saved in PatientItem.

10. In PatientItem::populate, updates all calls to its mutators to no longer be using the local patientItem object created in ItemModelMapper.

11. In ItemModelMapper, update the populate function call to originate from the patientItem object, and pass in all parameters as appropriate.

12. Run the tests again to ensure all data is correctly being populated.

Rationale

We chose to extract method because this is the operation being performed essentially: moving the createPatientItem method into the PatientItem class, and leaving ItemModelMapper::createPatientItem as a middle man (a common side effect of reducing feature envy, but in our case judged as acceptable). However, IntelliJ doesn't let us extract the method into another class, so we have to copy and paste it into the PatientItem class. From there it's simply a matter of giving it the correct method signature. Since all the functions being called are PatientItem mutators, we can just remove the "patientItem." Using the find -> replace all passing in an empty string.

## Manual Refactoring: Feature Envy

Class: ResearchController

Method: createResearchFilterItem

Smell: FeatureEnvy

Steps

1. The given method is in the controller and as of today we don’t know how to test the controller due to lack of knowledge. We decided to test this manually from the UI.

2. We navigate to research menu option and set the filters and exported the csv. Test passed as we get csv with data

3. Manually added a parameterized constructor in ResearchFilterItem with parameter FilterViewModel

5. We moved all the code till the return statement from method createResearchFilterItem to the class ResearchFilterItem

6. Next we set all the variables of ResearchFilterItem in the newly created constructor.

7. Run the test again from step 2. Passed

Rationale

The createResearchFilterItem of ResearchController is accessing the getAccessors of FilterViewModelobject and using the setters of ResearchFilterItem to populate the researchFilterItem. It made sense to move this responsibility to ResearchFilterItem by creating a new constructor that would accept a filterViewModel object.

## Automated Refactoring: Message Chain

Class: ItemModelMapper

Method: createMissionTripItem

Smell: Message chain

Steps

1. Added test testRetrieveAllTripInformation in MissionTripServiceTest. This test calls the retrieveAllTripInformation method of MissionTripService which calls the createMissionTripItem method of ItemModelMapper

2. Ensured test passed

3. Used IDE option to generate a default constructor in MissionTripItem.

4. Went back to newly created constructor and used the refactoring option "Change Signature" to add the parameter IMissionTrip missionTrip.

5. The next step was a bit of manual refactoring. We moved all the code till the return statement to the class MissionTripItem using copy/paste.

6. Next we set all the variables of MissionTripItem in this newly created constructor.

7. Run the test again. Passed.

Rationale

The createMissionTripItem of ItemModelMapper is accessing the getAccessors of MissionTrip object and using the setters of MissionTripItem to populate the missionTripItem. It made sense to move this responsibility to MissionTripItem by creating a new constructor that would accept a missionTrip object. The message chaining is the result of the middle man responsibility of ItemModelMapper having to set the properties of MissionTripItem from MissionTrip. Hence creating a new constructor and passing the missionTrip to it will move this responsibility, thereby also removing the chaining.

Manual Refactoring Done

Steps 4 & 5 were manual as we could not find a refactoring option within the IDE that would suit those specific actions.

## Manual Refactoring: Message Chain

Class: MissionTripService

Method: createNewTrip

Smell: Message Chain

Steps

1. Created MissionTripServiceTest and added test to create the trip item by calling createNewTrip of MissionTripService.

2. Ensured test passed.

3. Went back to Line 397 in method createNewTrip, where the message chaining existed.

4. Instead of passing properties of missionTrip to the method createTripItem, pass the missionTrip object itself.

5. Updated the ItemModelMapper class to take in the passed object using its supported interface and set the tripItem properties in ItemModelMapper

6. Run the test again. Passed

Rationale

The createTripItem of ItemModelMapper is accessing all the getAccessors of MissionTrip object, resulting in too many parameters being passed. The message chaining is the result of the parameters needed to be passed. Hence passing the object itself will eliminate the need of too many parameters, thereby also removing the chaining.

## Manual Refactoring: God Class

Class: ItemModelMapper

Smell: God class

Steps

1. InCode pointed out that a large part of the score for ItemModelMapper's God class smell was because of usage of external data. In this class, it takes the form of feature envy. So, to fix it, we're going to fix feature envy for the worst methods, and then run InCode to see how the score improves (if it improves!) after each one.
2. We've already refactored createPatientItem to no longer have feature envy for a single method. Next we do createMedicationItem.
3. Create a UnitTest for createMedicationItem. Ensure it passes.
4. Follow the steps outlined in AutomatedRefactoring: FeatureEnvy for createMedicationItem.
5. Extract createMedicationItem to a new method, and move it to the MedicationItem class.
6. Call the new method populate, give it the correct method signature, and call the new populate function in ItemModelMapper.
7. Ensure createMedicationItem passes the unit test again.
8. After fixing the feature envy for ItemModelMapper::createMedicationItem, we ran InCode again. Still a severity of 10! On to the next method.
9. ItemModelMapper::createPatientEncounterItem seems to be a large offender. Choose it next for refactoring.
10. Create a unit test for createPatientEncounterItem. Ensure it passes.
11. Extract the method to populate, move to PatientEncounterItem.java, and set the method signature correctly.
12. Update ItemModelMapper::createPatientEncounterItem accordingly, and ensure it compiles/runs.
13. Ensure it passes the unit test again.
14. Run InCode after fixing feature envy for ItemModelMapper::createPatientEncounterItem. It's still 10! Find another transgressor method. We chose createPrescriptionItem.
15. Create a unit test for createPrescriptionItem. Ensure it passes.
16. Extract the method to populate, move to PrescriptionItem.java, and set the method signature accordingly.
17. Update ItemModelMapper::createPrescriptionItem accordingly, and ensure it compiles/runs.
18. Ensure it passes the unit test again.
19. Run InCode again, it's still 10. InCode lists 2 methods remaining as having feature envy; createTabFieldItem and createSettingItem. We're going to refactor both of those before running InCode again.
20. Follow the above steps for createTabFieldItem, ensure it passes unit test after refactoring.
21. Follow the above steps for createSettingItem, ensure it passes unit test after refactoring.
22. Run InCode again. Mission complete! ItemModelMapper no longer even shows up under the GodClass smell, and it has no more feature envy entries.

Rationale

For some reason ItemModelMapper is essentially performing the task of creating these objects, instead of the objects creating themselves. In order to cut down on the external data access, we simply moved these operations into the classes of the various objects being created for the worst offenders, until ItemModelMapper no longer registered as a God class. Plus, if any future classes need to populate one of these objects with data, they no longer have to call every mutator method, but can simply call the "populate" method, which reduces code duplication in the system as a whole. We chose to use extract method because that's all we were doing – extracting the method from ItemModelMapper and then moving it into the respective classes. We had to align the method signatures manually but this took almost no time. This refactoring was so simple that even doing manual operations was not difficult.

## Refactoring Comparison

For our chosen smells, the automated refactoring support wasn’t critically useful, as most of what we were doing was essentially copy/paste/rename and moving stuff around. I think it’s helpful to keep in mind that we were keeping our changes to fairly localized regions of the code, so we didn’t necessarily need extensive automated refactoring report. Also, the code smells we chose to refactor weren’t overly involved so we didn’t have any trouble doing it manually. The most useful auto refactoring tool was probably rename symbol, as well as find and replace. Honestly, if you consider copy/paste to be an automated refactoring tool, then that was the one we used by far the most. There might have been better refactoring tools available for us to use, but we aren’t familiar with IntelliJ and so we might have missed out on those.