EMSE 6356 – Fall 2016

Team Name: Team 4

Assignment 2

Members: Gerald Tucker | Chikadibia Ihejimba

Code URL: <https://github.com/gerrytucker78/femr/tree/femr_smells_and_refactoring>

Contents

[Analyzing fEMR’s cohesion and coupling 2](#_Toc467093414)

[Cohesion 2](#_Toc467093415)

[Coupling 4](#_Toc467093416)

[Detecting code smells in fEMR 6](#_Toc467093417)

[Summary 6](#_Toc467093418)

[God Class 8](#_Toc467093419)

[femr.ui.controllers.MedicalController 8](#_Toc467093420)

[femr.common.ItemModelMapper 9](#_Toc467093421)

[Feature Envy 10](#_Toc467093422)

[femr.ui.controllers.ResearchController.createResearchFilterItem 10](#_Toc467093423)

[femr.common.ItemModelMapper.createPatientItem 11](#_Toc467093424)

[Message Chains 12](#_Toc467093425)

[femr.common.ItemModelMapper.createMissionTripItem 12](#_Toc467093426)

[femr.business.services.system.MissionTripService.createNewTrip 12](#_Toc467093427)

[Refactoring 14](#_Toc467093428)

[Automated 14](#_Toc467093429)

[femr.common.ItemModelMapper.createPatientItem 14](#_Toc467093430)

[femr.common.ItemModelMapper.createMissionTripItem 15](#_Toc467093431)

[Manual 15](#_Toc467093432)

[femr.business.services.system.MissionTripService.createNewTrip 15](#_Toc467093433)

[femr.ui.controllers.ResearchController.createResearchFilterItem 16](#_Toc467093434)

[Comparison of Automated to Manual 17](#_Toc467093435)

# Analyzing fEMR’s cohesion and coupling

## Cohesion

**Criteria for non-trivial:** TLLOC >= 250

**Type of Cohesion: LCOM5**

Per the SourceMeter Manual, it uses **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. It is calculated by taking a non-directed graph, where the nodes are the implemented local methods of the class and there is an edge between the two nodes if and only if a common (local or inherited) attribute or abstract method is used or a method invokes another. The value of the metric is the number of connected components in the graph not counting those, which contain only constructors, destructors, getters, or setters.

Class: number of functionalities of the class.

**Most Cohesive**

|  |
| --- |
| /home/gtucker/repos/femr/app/femr/util/startup/MedicationDatabaseSeeder.java |
| /home/gtucker/repos/femr/app/femr/util/startup/DatabaseSeeder.java |

**Least Cohesive**

|  |
| --- |
| /home/gtucker/repos/femr/app/femr/business/services/system/SearchService.java |
| /home/gtucker/repos/femr/app/femr/common/ItemModelMapper.java |

**Rationale:** Why the selected classes have high/low cohesion/coupling?

We sorted the LCOM5 count in ascending order for the “Most Cohesive” and descending order for the “Least Cohesive” and selected the top two classes:

|  |  |
| --- | --- |
| Class | LCOM5 Count |
| MedicationDatabaseSeeder.Java | 1 |
| DatabaseSeeder.Java | 1 |
| ItemModelMapper.Java | 16 |
| SearchService.Java | 2 |

From the code perspective:

* **MedicationDatabaseSeeder**: Does a well-defined job of handling of Medication Database seeding. It’s methods (MedicationDatabaseSeeder(), seed(), seedConceptPrescriptionAdministrations(), etc.) all have interrelated function and don’t handle anything order than Medication Seeding for the database.
* **DatabaseSeeder**: This also has a well-defined job. Its methods (DatabaseSeeder(), seed(), seedDiagnosis(), seedMissionTripInformation(), seedPatientAgeClassification(), seedSystemSettings(), seedDefaultTabFields(), seedDefaultTabNames(), etc.) are strongly related and they all support a central purpose of seeding to the database.
* **SearchService**: Least cohesive because it does multiple unrelated functions. For example, the following methods handle different functionalities:

1. ServiceResponse<File> - **Retrieves** CSV **Export** File
2. ResearchExportItem() - **Creates** Research Export Item
3. ServiceResponse<ResearchResultSetItem> - Retrieves **GraphData**
4. getWeeksPregnant() - **get** number of weeks **pregnant**

* **ItemModelMapper**: This class is the least cohesive amongst all the classes and does multiple unrelated functions, and its methods are not related to each other.

Here are the methods and its functionalities:

1. CityItem() - Creates **City** Item
2. MedicationItem() - Creates **Medication** Item
3. MissionItem() - Creates **Mission** Item
4. MissionTripItem() - Creates **Mission Trip** Item
5. PatientItem() - Creates **Patient** Item
6. PatientEncounterItem() - Creates **Patient Encounter** Item
7. PhotoItem() - Creates **Photo** Item
8. PrescriptionItem() - Creates **Prescription** Item
9. ProblemItem() - Creates **Problem** Item
10. SettingItem() - Creates **Setting** Item
11. TabItem() - Creates **Tab** Item
12. TabFieldItem() - Creates **Tab Field** Item
13. TeamItem() - Creates **Team** Item
14. TripItem() - Creates **Trip** Item
15. UserItem() - Creates **User** Item
16. VitalItem() - Creates **Vital** Item
17. MedicationAdministrationItem() - Creates **Medication Administration** Item

**Difference:** The differences between high- and low-cohesive/coupled classes?

For the highly cohesive classes (MedicationDatabaseSeeder and DatabaseSeeder), they do a defined and well set out job of seeding the database. Whereas the least cohesive classes (SearchService and ItemModelMapper) does multiple set of jobs. For example, ItemModelMapper houses multiple functions that have to do with “create” notwithstanding if they are related or not.

## Coupling

**Type of Coupling: CBO**

It uses the Coupling Between Object classes (CBO). It is a count of the number of classes that are coupled to a class i.e. where the methods of one class call the methods or access the variables of the other. These calls need to be counted in both directions so the CBO of class A is the size of the set of classes that class A references and those classes that reference class A. Since this is a set - each class is counted only once even if the reference operates in both directions i.e. if A references B and B references A, B is only counted once.

Class: 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.

**Least Coupled**

|  |
| --- |
| /home/gtucker/repos/femr/app/femr/ui/controllers/PDFController.java |
| /home/gtucker/repos/femr/app/femr/util/startup/MedicationDatabaseSeeder.java |

**Most Coupled**

|  |
| --- |
| /home/gtucker/repos/femr/app/femr/business/services/system/SearchService.java |
| /home/gtucker/repos/femr/app/femr/common/ItemModelMapper.java |

**Rationale:** Why the selected classes have high/low cohesion/coupling?

We sorted the CBO count in ascending order for the “Least coupled” and descending order for the “Most coupled” and selected the top two classes:

|  |  |
| --- | --- |
| Class | CBO Count |
| MedicationDatabaseSeeder.java | 13 |
| PDFController.java | 15 |
| SearchService.java | 35 |
| ItemModelMapper.java | 38 |

From the code perspective:

* **MedicationDatabaseSeeder**: Has low coupling compared to the rest of the classes. Though it is a bit tightly coupled with some classes. Creating a “concept” interface will reduce the coupling and also the CBO count. Example of the classes:
  1. ConceptMedication - conceptMedicationRepository
  2. ConceptMedicationGeneric - conceptMedicationGenericRepository
  3. ConceptMedicationGenericStrength - conceptMedicationGenericStrengthRepository
  4. ConceptMedicationUnit - conceptMedicationUnitRepository
  5. ConceptMedicationForm - conceptMedicationFormRepository
  6. ConceptPrescriptionAdministration - conceptPrescriptionAdministrationRepository
* **PDFController**: This is among the classes with low coupling in femr. Though its reliability on other classes/interfaces is a bit high, it can be easily mitigated by creating an interface to act as an intermediary. Example of such classes/interfaces include:

1. ISearchService - searchService
2. IEncounterService - encounterService
3. ITabService - tabService
4. IVitalService – vitalService
5. Controller

* **SearchService**: This class is tightly coupled with lots of classes that also couples with multiple classes. For example, it implements the ISessionService interface which is coupled with createSession class. At the same time, the ServiceResponse class is also coupled with createSession. It also has lots of interfaces which is tightly coupled with multiple classes and hence increasing the coupling. For example:
  1. IUserService - userService
  2. IMissionTripService - missionTripService
  3. IPasswordEncryptor - passwordEncryptor
  4. ISessionHelper - sessionHelper
  5. IRepository<ILoginAttempt> - loginAttemptRepository (Implements another interface)
  6. IDataModelMapper - dataModelMapper
  7. IRepository<ISystemSetting> - systemSettingRepository (Implements another interface)
  8. IRepository<IRole> - roleRepository (Implements another interface)
* **ItemModelMapper**: This can be referred to as the God class. It does almost everything and as a result has high coupling. It also showed up in our least cohesive list which is expected. It has a high dependency on other classes, example of such classes include:
  1. createCityItem
  2. createMedicationItem
  3. createMissionItem
  4. createMissionTripItem
  5. createPatientItem
  6. createProblemItem
  7. createSettingItem
  8. createMedicationAdministrationItem

**Difference:** the differences between high- and low-cohesive/coupled classes?

Our least coupled classes (PDFController and MedicationDatabaseSeeder) have multiple interfaces maintained that reduced the level of implementation coupling, whereas the highly coupled classes (SearchService and ItemModelMapper), made lots of direct calls to various classes and hence making them tightly coupled. Furthermore, the highly coupled classes are also the least cohesive which in this case is related. This is due to the classes both serving as “utility” classes or “helper” classes.

# Detecting code smells in fEMR

## Summary

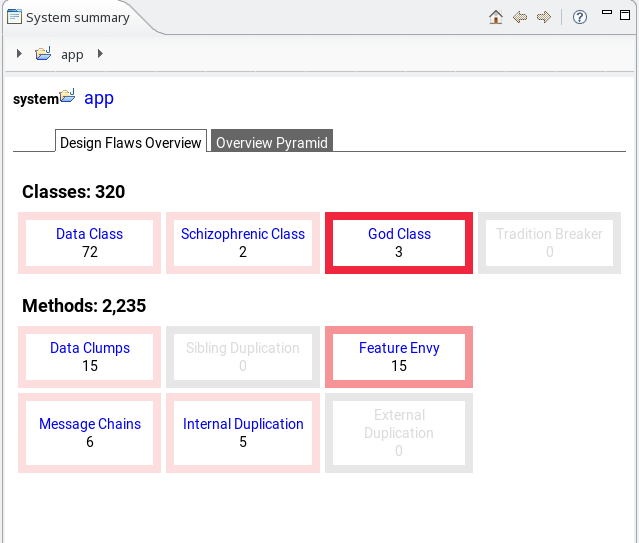
Below is the system summary after executing inCode against the fEMR source code prior to smell refactoring. The top 2 most impactful design flaws shown below are God Class and Feature Envy. When deciding what the 3rd most impactful flaw was, the team narrowed the analysis down between Data Class, Data Clumps, and Message Chains. After reviewing the definitions of the flaws, the decision was that Message Chains would be more impactful based on the level of indirect coupling that occurred.

Figure Summary Smell Analysis Prior to Refactoring

All three of these flaws aligned with the type of coding flaws we noticed during Assignment 1 while attempting to learn the code and make changes.

Within each flaw category, the top 2 examples of these flaws were selected and analyzed. Part the analysis involved extracting the following metrics from inCode and SourceMeter for the class or method.

| Metric | Applicability | Description |
| --- | --- | --- |
| Incoming Coupling Dispersion for an Operation (ICDO) [[1]](#footnote-2) | Class  Method | This metric is defined as the number of distinct (external) classes or modules from which operations (methods or functions) call the measured operation. For an operation, the higher the value of ICDO is, the higher the impact of a change in this operation on all the client capsules (modules or classes). |
| Outgoing Coupling Dispersion for an Operation (OCDO) 1 | Class  Method | This metric is defined as the number of distinct (external) classes or modules from which operations (methods or functions) are called by the measured operation. Operation calls are a form of coupling between capsules. Since coupling in general is not desirable, this metric is used to measure the amount of coupling between capsules. |
| Access to Local Data (ALD) 1 | Class  Method | This metric is defined as the number of distinct data (attributes and/or global variables) accessed by an operation (method or function), whereby this data belongs to the same capsule (class or module) where the operation is defined. |
| Tight Class Cohesion (TCC) 1 | Class | This metric is defined as the relative number of directly connected operations. An operation O1 is directly connected to another operation O2, if both O1 and O2 or their corresponding call trees access at least one common variable.  Capsules with TCC<0.5 are considered non-cohesive. A capsule with TCC=0.8 is considered "quite cohesive," while a capsule with TCC=1 is a maximally cohesive. |
| Access to Foreign Data (ATFD) | Method |  |
| Number of Incoming Invocations (NII)[[2]](#footnote-3) | Method | Number of other methods and attribute initializations which directly call the method. If the method is invoked several times from the same method or attribute initialization, it is counted only once. |
| Number of Outgoing Invocations (NOI)2 | Method | Number of directly called methods. If a method is invoked several times, it is counted only once. |

## God Class

A God Class is defined as a class that heavily uses the attributes of other classes either directly or indirectly. The top 2 God Class findings are shown below.

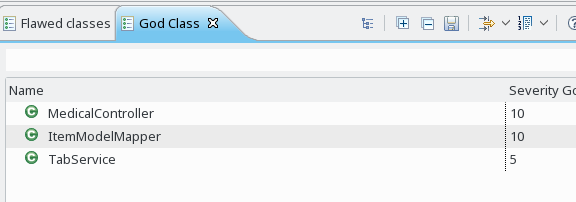


Figure God Class Top Candidates by Severity

### femr.ui.controllers.MedicalController

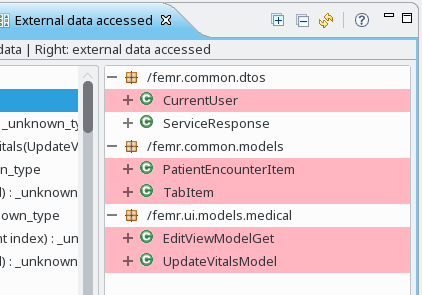
**Summary Information:**

Figure MedicalController Ext. Data Accessed

|  |  |  |
| --- | --- | --- |
| Source | Metric | Value |
| inCode | ICDO | 0 |
| inCode | OCDO | 10 |
| inCode | ALD | 30 |
| inCode | TCC | .25 |
| SourceMeter | LCOM5 | 1 |
| SourceMeter | CBO | 30 |

**Explanation**

Given the lack of cohesion based on the TCC, the high coupling indicated by ALD and CBO, and size and complexity of the class, inCode has indicated this is a God Class.

The class has low data cohesion amongst the public methods based on the TCC < 0.5. However, the LCOM5 metric indicates that the cohesion for the class is high. We believe that both metrics are correct in their own way. Given that this class is a MVC pattern Controller class, it serves the role of retrieving and acting upon the Model data for the Views that interact with it. Given that, the class is Cohesive for the role that it serves (LCOM5 and ALD). However, that requires that the TCC will be low as it simply serves as a broker of information. The TCC measure analysis is reinforced by the number of outgoing dependencies of 10 classes based on the OCDO coupling measure.

**Agreement**

Yes, the team agrees with the assessment of this being a God Class. This class is the 10th largest (TLLOC = 337) of the significant classes (based on TLLOC > 250). Furthermore, this class is the 4th most complex of the significant classes (CBO = 30). Based on those metrics, we concur.

### femr.common.ItemModelMapper

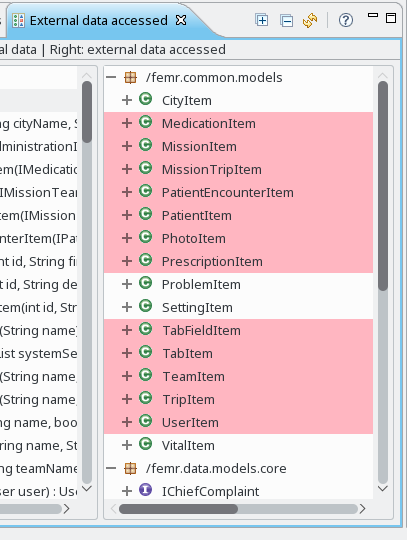
**Summary Information**

Figure ItemModelMapper Ext. Data Accessed

|  |  |  |
| --- | --- | --- |
| Source | Metric | Value |
| inCode | ICDO | 0 |
| inCode | OCDO | 14 |
| inCode | ALD | 0 |
| inCode | TCC | 0 |
| SourceMeter | LCOM5 | 16 |
| SourceMeter | CBO | 38 |

**Explanation**

After reviewing the source code, this entire class serves as a data utility class for creating different data items based on provided primitive data.

Given the lack of cohesion based on the TCC, the high coupling indicated by OCDO, and size and complexity of the class, inCode has indicated this is a God Class.

The LCOM5 concurs reinforces the God Class indication based on indicating the class could be divided into 16 distinct classes.

**Agreement**

Yes, the team agrees with the assessment of this being a God Class. This class is the 6th largest (TLLOC = 396) of the significant classes (based on TLLOC > 250). Furthermore, this class is the most complex of the significant classes (CBO = 38). These metrics coupled with the source code analysis by the team provided above is the basis for this agreement.

Anecdotally, this class was encountered the many times while working on project 1 while learning the code and tracing the path of data which seems to be a valid indicator of this being a God Class as well.

## Feature Envy

Feature envy is defined as a class/method caring more about the data of other objects than object that contains it. Below are the top candidates of this smell based on severity.

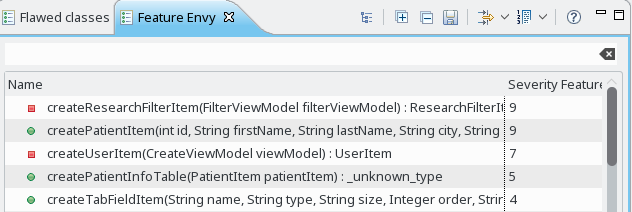


Figure Feature Envy - Top Candidates by Severity

### femr.ui.controllers.ResearchController.createResearchFilterItem

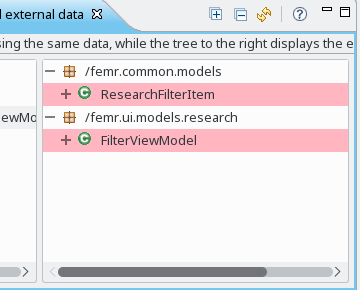
**Summary Information**

Figure createResearchFilterItem - External Data Usage

|  |  |  |
| --- | --- | --- |
| Source | Metric | Value |
| inCode | ICDO | 0 |
| inCode | OCDO | 2 |
| inCode | ALD | 0 |
| inCode | ATFD | 19 |
| SourceMeter | NII | 2 |
| SourceMeter | NOI | 22 |

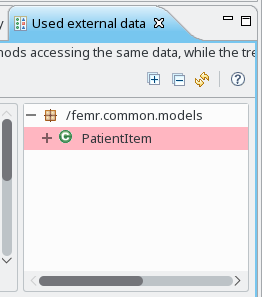
**Explanation**

After reviewing the source code, this method essentially serves as a Factory Method for generating ResearchFilterItem objects using a FilterViewModel object. The private method uses no part of the ResearchController class and is instead used as a helper method. Given the definition of what a Factory method is (method to creating objects given input data), this validates the feature envy smell.

**Agreement**

Yes, the team agrees based on the NOI and ALD metrics along with the explanation listed above. This method seems more fitting as a factory method in a more appropriately related class.

### femr.common.ItemModelMapper.createPatientItem

**Summary Information**

|  |  |  |
| --- | --- | --- |
| Source | Metric | Value |
| inCode | ICDO | 0 |
| inCode | OCDO | 3 |
| inCode | ALD | 0 |
| inCode | ATFD | 18 |
| SourceMeter | NII | 0 |
| SourceMeter | NOI | 25 |

**Explanation**

After reviewing the source code, this entire class serves as a data utility class for creating different data items based on provided primitive data. Specifically, this method essentially serves as a Factory Method for generating PatientItem objects using primitive data provided. Given the definition of what a Factory method is (method to creating objects given input data), this validates the feature envy smell.

Figure createPatientItem - External Data Usage

**Agreement**

Yes, the team agrees based on the NOI and ALD metrics along with the explanation listed above. The implementation of this factory pattern method is also breaking the encapsulation concept for the PatientItem class by dealing with “business logic” checks for setting data elements. Details such as what defines a valid PatientItem and which values can be null are all exposed and coded in the method.

## Message Chains

This smell is defined as one method/class using data access methods of other classes in a nested fashion. Below are the top candidates of this smell based on severity.

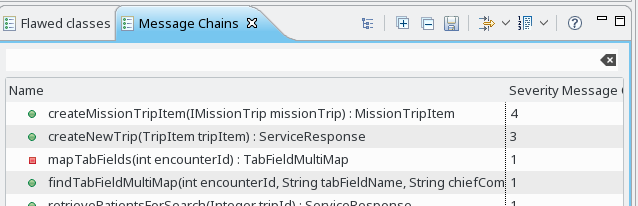


Figure Message Chains - Top Candidates by Severity

### femr.common.ItemModelMapper.createMissionTripItem

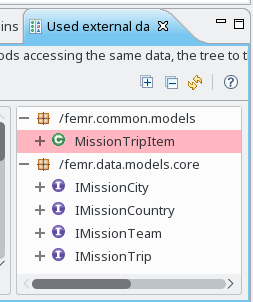
**Summary**

Figure createMissionTripItem - External Data Usage

|  |  |  |
| --- | --- | --- |
| Source | Metric | Value |
| inCode | ICDO | 0 |
| inCode | OCDO | 2 |
| inCode | ALD | 0 |
| inCode | ATFD | 9 |
| SourceMeter | NII | 0 |
| SourceMeter | NOI | 24 |

**Explanation**

Reviewing the code, it is clear visually that Message Chains is occurring by finding the long line of nested getters. This is primarily driven by the usage of the IMissionTrip class that has IMissionCity which provides access to IMissionCountry. This level of indirection simplifies the method signature, but then artificially forces the message chaining.

**Agreement**

Yes. Chaining in this situation easily cause issues if additional levels of hierarchy were inserted into the location information. For example, a city might have a state/province added in the future that then has a country. This would force changes to this class for no reason.

### femr.business.services.system.MissionTripService.createNewTrip

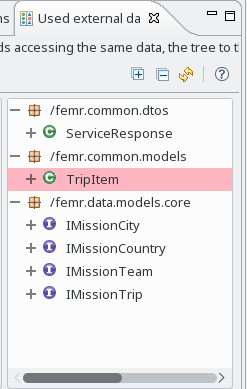
**Summary**

Figure createNewTrip - External Data Usage

|  |  |  |
| --- | --- | --- |
| Source | Metric | Value |
| inCode | ICDO | 0 |
| inCode | OCDO | 5 |
| inCode | ALD | 6 |
| inCode | ATFD | 6 |
| SourceMeter | NII | 0 |
| SourceMeter | NOI | 25 |

**Explanation**

Similar to createMissionTripItem, IMissionCity and IMissionCountry are the primary drivers for the Message Chains. The TripItem takes the place of IMissionTrip from createMissionTripItem. Reviewing the code, it is clear visually that Message Chains is occurring by finding the long line of nested getters. This level of indirection simplifies the method signature, but then artificially forces the message chaining.

**Agreement**

Yes. Chaining in this situation easily cause issues if additional levels of hierarchy were inserted into the location information. For example, a city might have a state/province added in the future that then has a country. This would force changes to this class for no reason.

# Refactoring

## Automated

### femr.common.ItemModelMapper.createPatientItem

**Refactoring Steps**

1. Created and verified a JUnit test for this method prior to any refactoring, added to git, and committed.
2. Decided that refactoring should entail moving the creationPatientItem method from ItemModelMapper to the PatientItem class as a static factory method.
   1. Assumption was that IntelliJ features would allow for this type of refactoring in an automated way.
   2. Rationale was that given this was a Factory Method, it made sense to include it in the class that it was serving as a factory of.
3. Attempted to use IntelliJ refactoring features to extract or move the create. We determined that there was not an automated feature to execute this refactoring.
4. Manually moved the method to PatientItem and changed to a public static method.
   1. Note: Prior to moving, we forgot to utilize the find usage capability.
5. A new Junit test for PatientItem to verify move of the function.
   1. Moved and updated test method from ItemModelMapperTest.
6. Attempted to build and execute JUnit test and realized that we missed updating usage of the method prior to moving it.
7. Utilized build failure results to update usage clients.
8. Execute JUnit test and verified test results were still successful.
9. Re-ran inCode and verified the createionPatientItem was no longer listed under Feature Envy.
   1. Note: Test methods are now included in the results. However, this is not a concern as the tests require this level of usage to verify a class.
10. Committed code changes to git.

|  |  |
| --- | --- |
| Before | After |
|  |  |

### femr.common.ItemModelMapper.createMissionTripItem

**Refactoring Steps**

1. Created and verified a JUnit test for this method prior to any refactoring, added to git, and committed.
2. Decided that refactoring should entail adding IMissionCity and IMissionCountry directly to the method signature.
   1. Assumption was that IntelliJ features would allow for this type of refactoring in an automated way.
   2. Rationale was that since the method required the data, it should be explicitly provided rather than assumed to be present in the IMission
3. Attempted to use IntelliJ refactoring features to add the new parameters to the method with default null value.
   1. Automated refactoring worked except for “breaking” advanced Java Lambda expression usage.
4. Manually replaced all default null values with proper inputs using Find Usage option.
5. Analyzed and converted the Java Lambda expression to a simplified for each loop.
6. Manually updated the method to use the new parameters.
7. While updating, determined that business logic code related to knowing what a “friendly date” should look like and calling the setter on MissionTripItem should be refactored as well. This logic was manually moved into the appropriate setDate methods in MissionTripItem.
8. Utilized the automated refactoring ‘Safe Delete’ feature to locate the setFriendlyDate equivalent method usage. Removed calls, re-ran ‘Safe Delete’ and methods were removed.
9. Minor updates were made to the JUnit test and it was re-run to verify all tests continued to pass.
10. Re-ran inCode and verified the createionMissionTripItem was no longer listed under Message Chains.
11. Committed code changes to git.

|  |  |
| --- | --- |
| Before | After |
|  |  |

## Manual

### femr.business.services.system.MissionTripService.createNewTrip

**Refactoring Steps**

1. Attempted to create a JUnit test prior to attempting to remove the code smells and determined that changes were required to this class and a few supporting classes to support making them testable.
2. Below are the following changes made to support testability:
   1. IMissionTripService / MissionTripService
      1. Changed generic response class type from TripItem to MissionTripItem. This was required as the MissionTripItem contained the database row id for the newly created row. This was required by the JUnit test so that the database could be properly cleaned up after test execution.
      2. Added getter for missionTripRepository. This was required to access to missionTrip database via the repository for database cleanup. The repository was injected into the MissionTripService as part of test setup. This appeared to be the quickest means for accessing class to perform the proper cleanup.
   2. IItemModelMapper / ItemModelMapper
      1. Changed createTripItem to return MissionTripItem and accept an id to set in MissionTripItem.
   3. TripController
      1. Updated usage of createNewTrip to support MissionTripItem generic class type instead of TripItem. No updates were required beyond method call.
3. Verified a JUnit test for this method prior to any refactoring, added to git, and commited.
4. Decided to remove TripItem class from method signature and replace instead with all required primitives. This would remove the need for chains within TripItem.
5. Manually updated the method signature.
6. Utilized build errors to update client usage.
7. Minor updates were made to the JUnit test and it was re-run to verify all tests continued to pass.
8. Re-ran inCode and verified the createNewTrip was no longer listed under Message Chains.
9. Committed code changes to git.

|  |  |
| --- | --- |
| Before | After |
|  |  |

### femr.ui.controllers.ResearchController.createResearchFilterItem

**Refactoring Steps**

1. Given that this is a private method, made the method public and static to support creating an initial JUnit test.
2. Created and verified a JUnit test for this method prior to any refactoring, added to git, and committed.
3. Decided to move the method to ResearchFilterItem as it served as a Factory Method for this class.
4. Manually move the method to ResearchFilterItem class and updated usage within ResearchController.
5. Move test from existing JUnit test to a new ResearchFilterItemTest.
6. Minor updates were made to the JUnit test and it was re-run to verify all tests continued to pass.
7. Re-ran inCode and verified the createResearchFilterItem was no longer listed under Feature Envy.
8. Committed code changes to git.

|  |  |
| --- | --- |
| Before | After |
|  |  |

## Comparison of Automated to Manual

Overall, the automated refactoring was only useful for a small subset of the refactoring work executed above. Many attempts were made at trying to use some of the automated features to only determine that it was not viable for that situation via a warning message. When usable, the advantage was the IDE automatically determining the validity of the proposed code change being made (Method Signature Update, Safe Delete). However, those cases, manual updates were required post refactoring (finding updated usage statements to replace default values with correct values) or prior to refactoring (manually delete usage statements). This is not a judgment of the feature. In fact, this is essentially required as there is no heuristic that can determine exactly how to refactor every code instance.

Manual refactoring only performed similar steps in finding usage and making updates compared to the automated above. However, it simply required the developer to understand when to perform a usage find prior to refactoring. Even when this was inadvertently missed, the automated build failures served as a safety net for finding usage issues.

Overall, the advantage of the automated refactoring is to protect the developer from inadvertently forgetting to execute certain steps of the refactoring analysis process. The disadvantage is that it cannot automate resolving the analysis steps and therefore is limited in utility.

1. Descriptions pulled from inCode Software Metrics Reference in the Online Help [↑](#footnote-ref-2)
2. Description pulled from SourceMeter 8.1 User Guide [↑](#footnote-ref-3)