**Analyzing Cohesion and Couple:**

1. Top Two non-trivial classes with high cohesion

*Cohesion Metrics: Lack of cohesion in Method 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*

1. **ItemModelMapper and Repository**

LCOM5 value 16 and 10 for those classes.

Why:

These classes have a lot of injections between the modules, which makes them have high cohesion. Also, all of the functionality is represented within the same class. To achieve high cohesion, these modules are focused on one topic and goal that they need to achieve.

Type – functional: one single goal is achieved by the module.

1. Top two non-trivial classes with low cohesion

**PhotoController, PatientItem**

LCOM5 value is zero (0) for both the classes

Why:

These classes are low cohesion because all the actions are included within the same module. Though the module is grouped in a way that is thought to be related, the parts of the classes are not related by the nature of them.

Type- logical: module thought to be related but in reality there is no association.

*Coupling between Object classes (CBO)*

*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.*

1. Top two non-trivial with low coupling

**MissionTripItem, PatientEncounterItem**

CBO value is zero for both the classes

Why:

These modules are preferred because they are loosely coupled, the classes are not so linked to each other. Only the data that is required is the one that is transferred. The data type is also in the simplest form.

Type – Data cupling: simplest form, one - to- one.

1. Top two non-trivial with high coupling

**DataLayerModule, ItemModelMapper**

CBO value is 7,138 for both the classes

Why:

These classes are highly coupled because they rely on the internal workings of other modules. They cannot complete a task on their own. There is high inter-module dependency.

Type: content coupling: high interdependency between modules which cause ripple effects.

**Detecting code smells in fEMR**

God Class

a.

1. ResearchService :

This class has a long complexity code. The cohesion among the classes is low. There is duplication of methods, i-e ISerachService. BuildAgeResultSet method has internal duplication. This class is highly coupled.

2. MedicalController:

This class has a long complexity code. The cohesion among the classes is low. There is duplication of methods, i-e ISerachService. There are 17 errors in the class. Index, routes, edit, prescriptionRow, newVitals, listsVitals are not defined/declared even though they are returning statement for the class. This class is highly coupled. This class also provides data information to the GUI and causes bugs in the information.

b. Yes, I agree with the detected smell because some of the attributes are undefined and there is duplication in the method. Also, a large number of instances are created, which can create memory consumption.

Data Class

1. IndexViewModelPost:

Attributes are defined but not used, can cause memory consumption.

Some of methods are not used i-e setAddress(),setAge,setBloodPressure() etc.

1. PatientEncounteredItem:

Attributes are declared but not used, which may cause memory consumption. Lists are also used in this class, but it should be object information that is used.

c. Yes, I agree with the class PatientEncounteredItem smell detect by Incode, as there is a list and some of the attributes are unused, which may cause duplication in the list and can also overwrite the attribute values. For the IndexViewModelPost class, I do not agree with the smell detect. If the method of attributes are not used, it does not have any impact with other classes. This also cannot lead to the bugs in the application.

Data Clumps

1. [createTabFieldItem](http://7142_gotopagelink_/)():

This method is used in class ItemModelMapper. This method has a long parameter list, which causes duplication by other methods. This can also give an impact on runtime performance.

1. ResearchResultSetItem()

This method is used in the implementation of class ResearchSerivce. Extra instances are created in the method, which can cause duplication with other methods. This can reduce the performance and can create runtime errors too on the compilation. This method can consume the memory too.

d. Yes, I agree with smell detected by Incode, The methods are duplicates which cause performance issues. If we create an instance of an object, we can initialize the attributes and can use them in the whole class. A good programming practice is, initialize attributes on the time of creation.

**Refactoring analysis**

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

ResearchService

We used a separate class named ResearchServiceRefactor. This class will return the service response to ResearchService. This will decrease the complexity of the class and make it low coupled.

MedicalController: Include the missing classes. Remove duplication in CurrentUser attribute.

Make a class MedicalControllerImplRefactor which extends medicalController. This class will return the services to controller. This will reduce the complexity and the code will be highly cohesive.

PatientEncounterItem:

Removed the list from the class, since it’s an object, it should hold objection information.

IndexViewModelPost:

Removed unused attributes.

1. Give the rationale of the chosen refactoring operations.

We used “Use interface where possible” because it passes on the implementation of the specific methods we want into an inner class using the same interface.

We used Push Members Down to organize the hierarchy and moving class members to subclasses.

We also used Find and Replace Code Duplicates to replace codes that showed similarities to the methods chosen, and replaced them with original constants.

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

We changed the PatientEncounterItem and IndexViewModelPost codes by ourselves because we knew that these classes used lists. Since this was pojo, we needed to remove the lists since pojo does not use it. We also removed the unused attributes from IndexViewModelPost to uncomplicated the code and make it smaller. The rest of the codes were changed automatically by IntelliJ’.

1. Compare the manual and automated refactoring processes that you performed. Describe the difficulties, advantages, and disadvantages of using one or the other.

Doing the changes manually was difficult because we were unaware of the flow in the code, and it was difficult to understand the impact analysis. Using IntelliJ’ was a help because it changed the code automatically without us having to interfere. Doing the codes manually would take longer, but would be more efficient in finding issues.