CSCD01 Team 21

Deliverable #3

Team 21: *The BugWriters*

Chengrong Zhang, Zhongyang Xia, Tan Jiaxin, Zongye Yang, Xingyuan Zhu

March 2020

Contents

0.1 Features Analysis

In this section, we will give a full detailed analysis of two features we choose from the issue list.

0.1.1 Feature Analysis #1

STAND-64

Change DBMS

Description

The mysql-mxj project is discontinued, the current BDMS will cause some troubles when using standalone instances. Performance-wise, MySQL is not as great as other SQL RDBMS, e.g., PostgreSQL or MariaDB here.

Code Analysis

An extensive dive in was conducted to the current OpenMRS code. However, the result is not very optimistic.

To begin with, MariaDB is a community-driven fork of MySQL, therefore they are very similar in some aspects. For a beginner, it may be possible to simply replace a MySQL database with MariaDB while hiding all the internal details to the MySQL connector. However this could be potentially dangerous, since the epoch of the fork, there are significant changes committed to the MariaDB project, especially internal implementations. The SQL dialect used by MariaDB also differs from MySQL slightly. Therefore, we would need to specify Hibernate to use MariaDB dialect. While it seems simple, extensive changes have to be made. The OpenMRS project did not follow design principles completely. Database connection is not handled by one abstract class. Instead, *Initialization*, *Connection*, *Database Update* and *web* all use their own connectors, which makes it hard to make a one-for-all change. Moreover,

MySQL is the first-class citizen of this project. A great portion of the code is tailored for MySQL, some are even hard-coded literals (e.g., string concatenation with the “jdbc://mysql:” URI and hard-coded “com.mysql.jdbc.Driver” connector), which makes it even harder to impose change and adopt native MariaDB connectors.

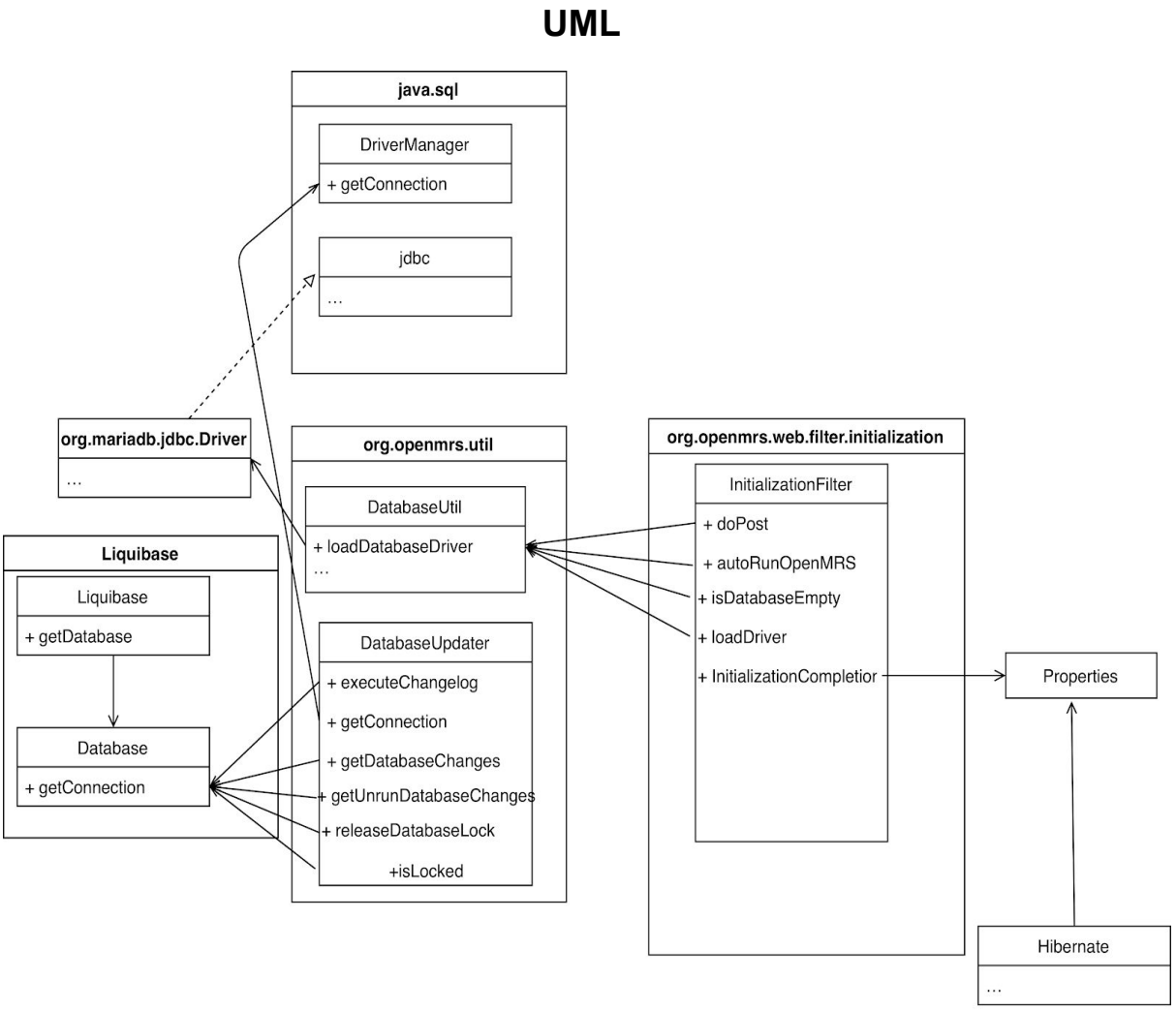


Figure 1.1: UML of the DBMS

Code Trace

After Implement, the DBMS will be change to MariaDB:

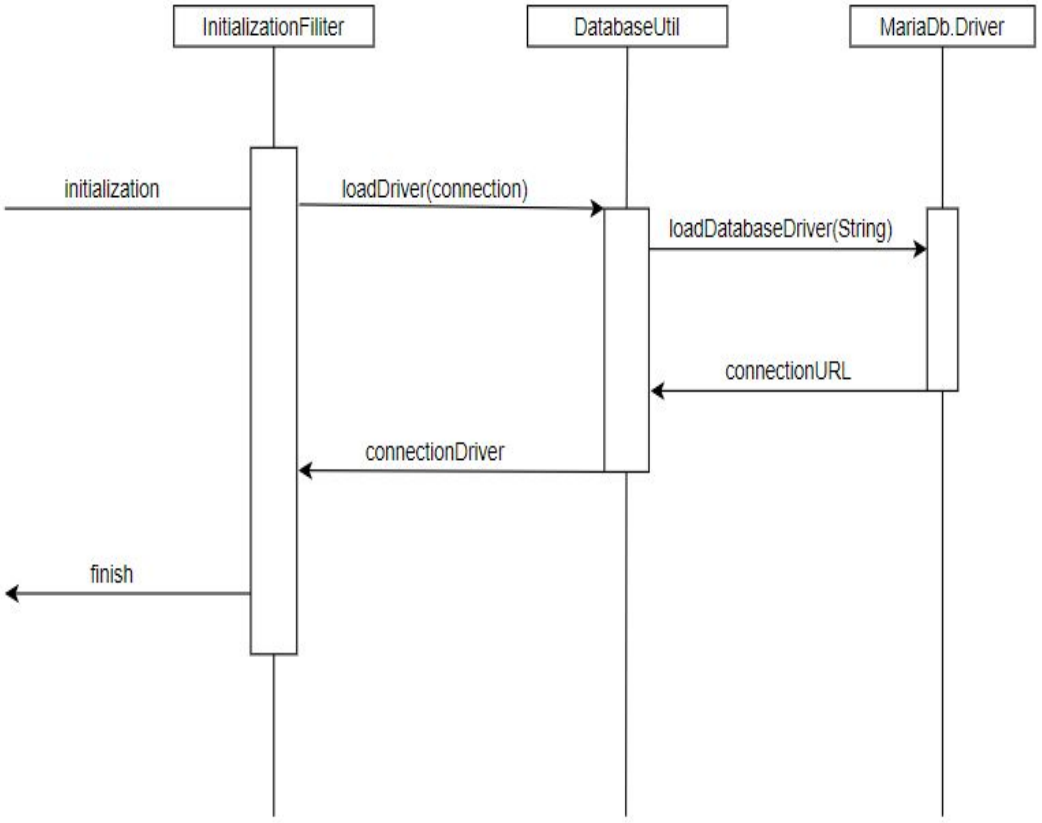


Figure 0.2: when start loading a driver

0.1.2 Feature Analysis #2

TRUNK-4022

Merge duplicate patient identifier types

Description

It would be nice to have an automatic feature to merge patient identifier types if duplicates exist. In the database that contains the patient identifier types, we should have a feature that checks for duplicate entries and merge them

Code Analysis

From doing extensive analysis on the code base, one of the files that would most likely need to be modified is HibernatePatientDAO.java, a data access object for the patient table in the database. We will implement a function called check_Table() here that calls getAllPatientIdentifierTypes() function from HibernatePatientDAO.java (Figure 1.1) and use check equality of every pair of patient identifier types by implementing a function called equalIdentifierType() inside PatientIdentifierType.java, and finally merge them if they are duplicate. We should also set up behavior in the code so that every time the table is updated, we call the function to merge duplicates. Other key files that we may need to modify to implement this behavior is PatientDAO.java.

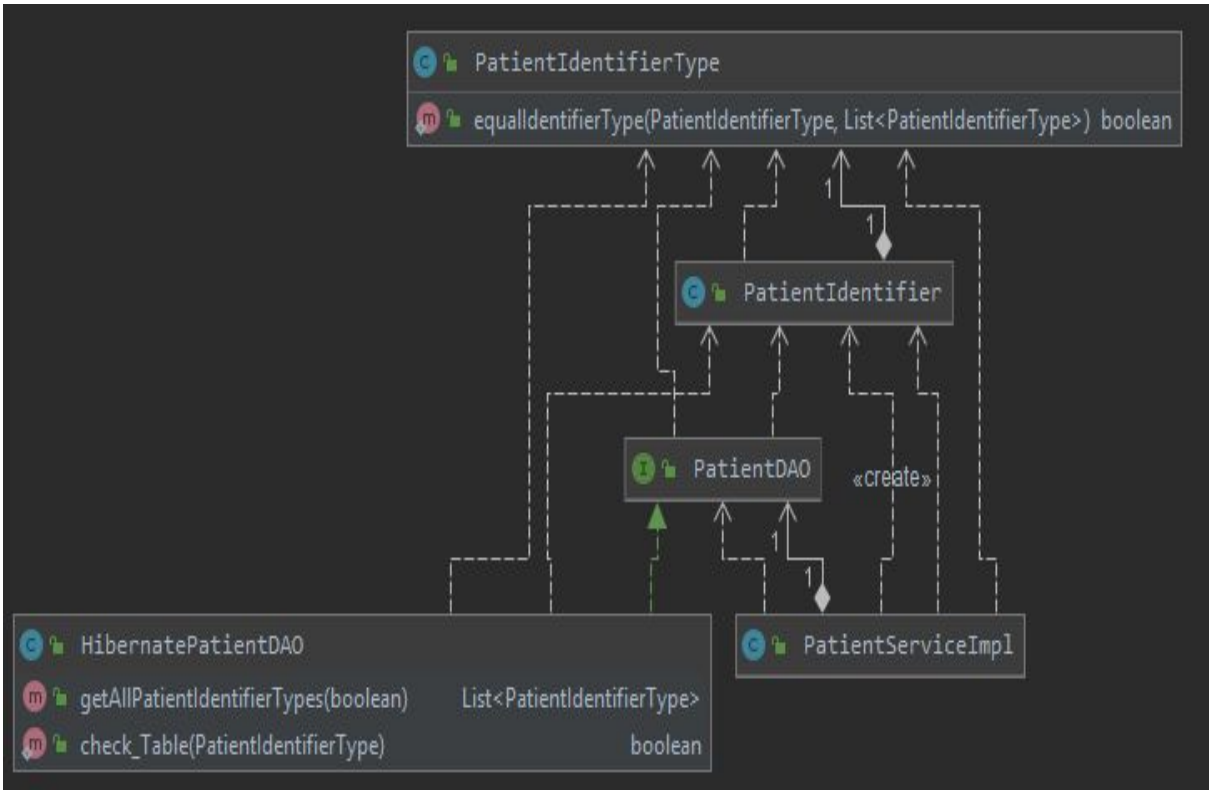
```
338         @Override
339     public List<PatientIdentifierType> getAllPatientIdentifierTypes(boolean includeRetired) throws DAOException {
340         Criteria criteria = sessionFactory.getCurrentSession().createCriteria(PatientIdentifierType.class);
341 
342         if (!includeRetired) {
343             criteria.add(Restrictions.eq("retired", false));
344         } else {
345             //retired last
346             criteria.addOrder(Order.asc("retired"));
347         }
348 
349         //required first
350         criteria.addOrder(Order.desc("required"));
351         criteria.addOrder(Order.asc("name"));
352         criteria.addOrder(Order.asc("patientIdentifierTypeId"));
353 
354         return criteria.list();
355     }
```

Figure 1.1: getAllPatientIdentifierTypes(), HibernatePatientDAO.java (return all patients)

Source Code Files:

1. api/src/main/java/org/openmrs/PatientIdentifierType.java
2. api/src/main/java/org/openmrs/PatientIdentifier.java
3. api/src/main/java/org/openmrs/api/db/hibernate/HibernatePatientDAO.java

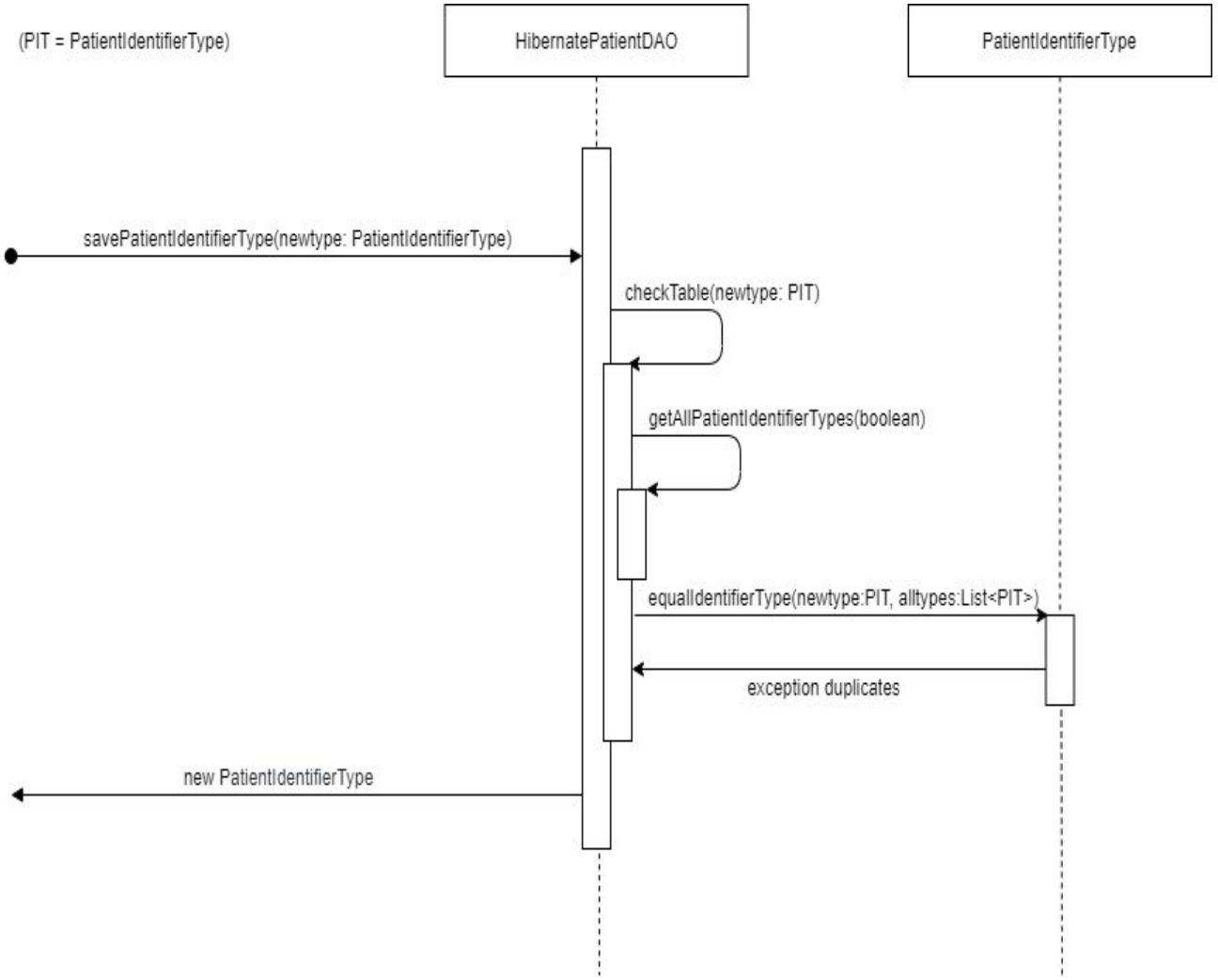
UML



check_Table and equalIdentifierType will be implemented inside HibernatePatientDAO.java and PatientIdentifierType for the new feature. getAllPatientIdentifierTypes is already present in the class.

Code Trace

After Implementation:



0.2 Features Implementation Plan and Test

In this section, we will map out a fully detailed implementation plan for one of the features we chose from the above section. We will also talk about the reason why we chose it and produce a suite of test cases for it.

0.2.1 Reasons we choose

Our team has decided to implement the feature TRUNK-4022 (see 0.1.2) in the upcoming deliverable 4. The following are the main reasons behind this decision by our team:

1. TRUNK-4022 is about implementing the feature that auto merges the duplicate patient identifier types if they exist in the system. By merging the duplicate types there will be a huge amount of space saved for the system, and the system therefore will respond more quickly. In other words there will be a significant improvement of OpenMRS after the feature is implemented.
2. The OpenMRS project is a huge open-source software with over hundreds of different modules, and in the previous deliverables our team spent most time dealing with the api module called openmrs-core. Since the feature will be implemented by adding code inside the openmrs-core and we are quite familiar with the structure of the module, we can get started to work very quickly.
3. After analyzing the two issues in above section, we found out that the issue STAND-64 (see 0.1.1) is way too complicated, also it is hard to make a one-for-all change and will require modification over 10 different database files, it is hard for us to implement in a short given time. Therefore the TRUNK-4022 is a better choice for us to implement.

0.2.2 Implementation Plan

We consider the following steps to implement the feature:

1. Modify the structure of PatientIdentifierType class.
   Add a compare function to the PatientIdentifierType class thus PatientIdentifierType subject can be compared with another.
2. Implement equalIdentifierType() function in PatientIdentifierType.java.
   equalIdentifierType() function will return True if two PatientIdentifierType have the same id or same format, and return False otherwise.
3. Modify the structure of PatientIdentifier class.
4. Implement check_Table() function inside HibernatePatientDAO.java.
   check_Table() function will return True if there exist duplicate PatientIdentifierType elements inside the database and merge them, otherwise the method will return False and do nothing.
5. Modify savePatientIdentifierType() function in HibernatePatientDAO.java.
   Call check_Table() function at savePatientIdentifierType() thus whenever there is a new PatientIdentifierType added into the database, the system will merge duplicates.
6. Modify deletePatientIdentifierType() function in HibernatePatientDAO.java.
   Call check_Table() function at deletePatientIdentifierType() thus whenever there is a deletion of PatientIdentifierType, the system will merge duplicates.

0.2.3 Acceptance Tests

Here our team designed a series of acceptance tests for the new changes. First of all, we surely need to test the new function equalIdentifierType() to check if it could detect the duplicate patient identifier types in anyway:

PatientIdentifierTypeTest.java:

Before the test, assume there is a PatientIdentifierType named type1, with id = 1 and format = “format1”.

1. When a user creates a PatientIdentifierType type2 with id = 1 and he calls equalIdentifierType(type1, type2), then the system should return True.
2. When a user creates a PatientIdentifierType type2 with id = 2 and he calls equalIdentifierType(type1, type2), then the system should return False.
3. When a user creates a PatientIdentifierType type2 with format = “format1” and he calls equalIdentifierType(type1, type2), then the system should return True.
4. When a user creates a PatientIdentifierType type2 with format = “format2” and he calls equalIdentifierType(type1, type2), then the system should return False.

```java
public class PatientIdentifierTypeTest extends BaseContextSensitiveTest {

    private PatientIdentifierType pit;

    @Before
    public void runBeforeEachTest() {
        Context.openSession();
        authenticate();
        pit = new PatientIdentifierType(1);
        pit.setFormatDescription("Format1");
        pit.setLocationBehavior(LocationBehavior);
        pit.setUniquenessBehavior(UniquenessBehavior);
        pit.setFormat("Format1");
        pit.setValidator("Validator1");
    }

    @Test
    public void testIdIsEqual() {
        test1 = new PatientIdentifierType(1);
        pit.equalIdentifierType(test1);
        assertTrue(pit.equalIdentifierType(test1));
    }

    @Test
    public void testIdNotEqual() {
        test1 = new PatientIdentifierType(2);
        pit.equalIdentifierType(test1);
        assertFalse(pit.equalIdentifierType(test1));
    }

    @Test
    public void testFormatIsSame() {
        test1 = new PatientIdentifierType(2);
        test1.setFormat("Format1");
        pit.equalIdentifierType(test1);
        assertTrue(pit.equalIdentifierType(test1));
    }

    @Test
    public void testFormatNotSame() {
        test1 = new PatientIdentifierType(2);
        test1.setFormat("Format2");
        pit.equalIdentifierType(test1);
        assertFalse(pit.equalIdentifierType(test1));
    }
```

Figure 2.1: PatientIdentifierTypeTest.java

Second, we need to write some other test cases to test our new check_Table() function:

HibernatePatientDAOTest.java:

1. When a user creates two duplicate PatientIdentifierType type1 and type2, and inserts them into the database, after the user calls check_Table() function, then there should be only one merged PatientIdentifierType left inside the database, and check_Table returns True.
2. When a user creates two non-duplicate PatientIdentifierType type1 and type2, and inserts them into the database, after the user calls check_Table() function, then nothing will change with the database, and check_Table returns False.
3. When a user calls check_Table() function while the database is empty, then nothing happens and check_Table returns False.

```
26  public class CheckTableTest extends BaseContextSensitiveTest{
27      private HibernatePatientDAO hibernatePatientDAO = new HibernatePatientDAO();
28
29      @Before
30      public void runBeforeTheTest(){
31          Context.openSession();
32          authenticate();
33          hibernatePatientDAO.setSessionFactory((SessionFactory) applicationContext.getBean("sessionFactory"));
34      }
35
36      public User generateNewUser() {
37          PatientIdentifierType p1 = new PatientIdentifierType(1);
38          p1.setFormatDescription("Format1");
39          p1.setLocationBehavior(LocationBehavior.NOT_USED);
40          p1.setUniquenessBehavior(UniquenessBehavior.UNIQUE);
41          p1.setFormat("Format1");
42          p1.setValidator("validator1");
43      }
44
45      public User generateSameUser() {
46          PatientIdentifierType p1 = new PatientIdentifierType(1);
47          p1.setFormatDescription("Format1");
48          p1.setLocationBehavior(LocationBehavior.NOT_USED);
49          p1.setUniquenessBehavior(UniquenessBehavior.UNIQUE);
50          p1.setFormat("Format1");
51          p1.setValidator("validator1");
52      }
53
54      public User generateDifferentUser() {
55          PatientIdentifierType p1 = new PatientIdentifierType(1);
56          p1.setFormatDescription("Format2");
57          p1.setLocationBehavior(LocationBehavior.NOT_USED);
58          p1.setUniquenessBehavior(UniquenessBehavior.UNIQUE);
59          p1.setFormat("Format2");
60          p1.setValidator("validator2");
61      }
```

Figure 2.2: HibernatePatientDAOTest.java (part 1)

```
63      /*
64      * put same users into the database. If check_Table return true, then the merge success.
65      */
66      @Test
67      public void testSameUser(){
68          hibernatePatientDAO.deletePatient();
69          hibernatePatientDAO.savePatientIdentifierType(generateNewUser());
70          hibernatePatientDAO.savePatientIdentifierType(generateNewUser());
71          assertTrue(check_Table());
72      }
73
74      /*
75      * put different users into the database. If check_Table return false, then the merge failed.
76      */
77      @Test
78      public void testDifferentUser(){
79          hibernatePatientDAO.deletePatient();
80          hibernatePatientDAO.savePatientIdentifierType(generateNewUser());
81          hibernatePatientDAO.savePatientIdentifierType(generateDifferentUser());
82          assertFalse(check_Table());
83      }
84
85      /*
86      * If the database is empty, then the check_Table should return false.
87      */
88      @Test
89      public void testNoUser(){
90          hibernatePatientDAO.deletePatient();
91          assertFalse(check_Table());
92      }
```

Figure 2.3: HibernatePatientDAOTest.java (part 2)

Lastly, we need to make sure that this merging feature happens automatically whenever an update happens in the database.

1. When the user inserts a PatientIdentifierType that is identical to one of the Types already present in the database, then check_Table() should run automatically during the transaction and merge the duplicate types, keeping the table duplicate-free.
2. When the user modifies an existing PatientIdentifierType to be identical to some other entry in the table, then the feature should run automatically and merge the two types. Resulting in a table with no duplicates.
3. When the user inserts a new PatientIdentifierType, the merging operation will still run, but nothing will be done since there are no duplicates. So the insertion goes through.
4. When the user modifies a PatientIdentifierType and it doesn't introduce a duplicate, then nothing happens and the modification goes through.
5. When the user deletes a PatientIdentifierType from the database, it won't introduce a duplicate but check_table will still run since the database got updated.

0.3 Architecture Document

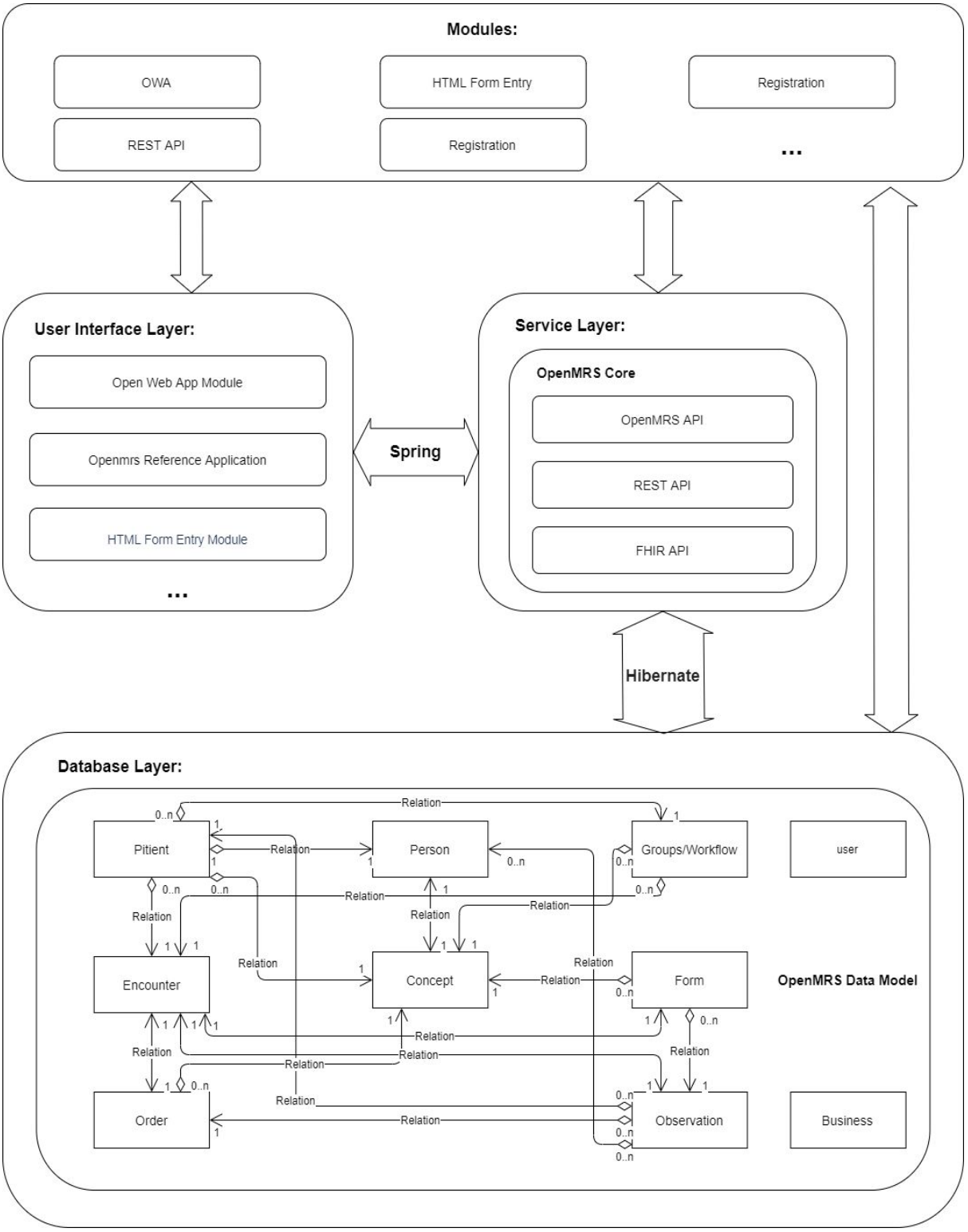


Figure 3.1: OpenMRS Architecture

Three main layers make the source code of OpenMRS:

1. The user interface layer
2. The service layer
3. The database layer

The User Interface Layer:

The user interface layer is responsible for handling different web pages of the system as well as the welcome page, user login page, forum page, etc. The User Interface layer of OpenMRS is built upon Spring MVC, Direct Web Remoting (DWR), JSP and JavaScript.

The Service Layer:

The Service layer is responsible for managing the business logic of the application. It is built around the Spring framework. The OpenMRS API has methods for all of the primary functions, such as adding/updating a patient, encounter, observation, etc. The OpenMRS service layer classes make extensive use of the Spring framework for several tasks, including the following:

1. Spring Aspect-Oriented Programming (AOP) is used to provide separate cross-cutting functions (e.g. authentication, logging).
2. Spring Dependency Injection (DI) is used to provide dependencies between components.
3. Spring is used to manage transactions between service layer classes.

The Database Layer:

The Data Access layer is responsible for storing the actual data model and its changes.

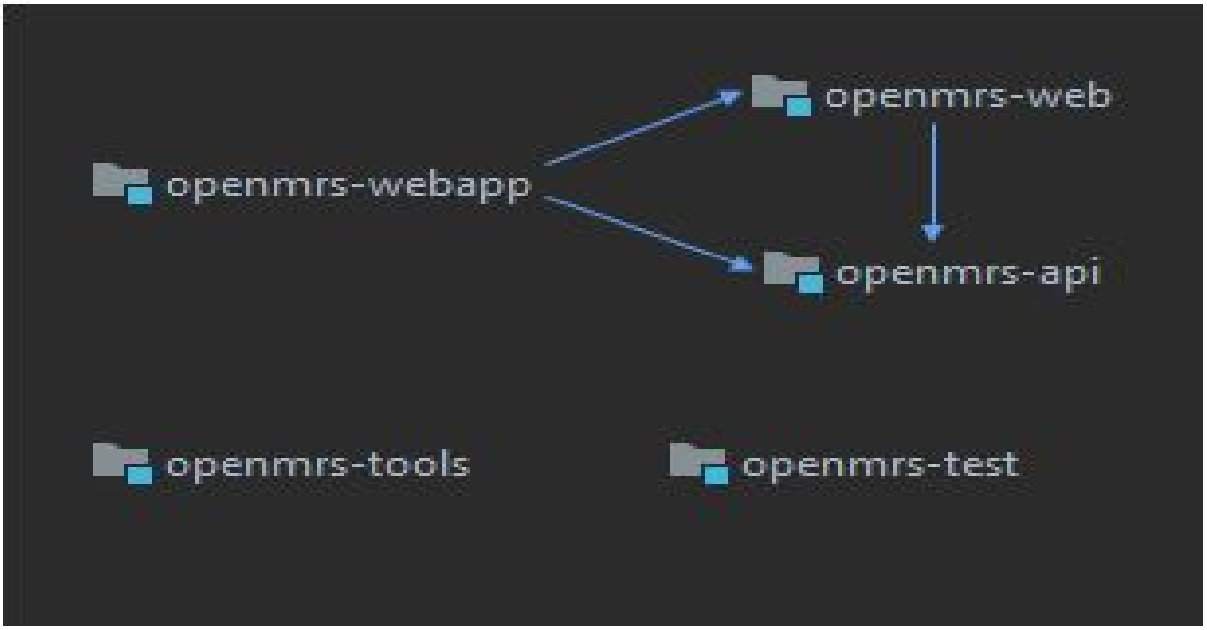


Figure 3.2: openmrs-core structure

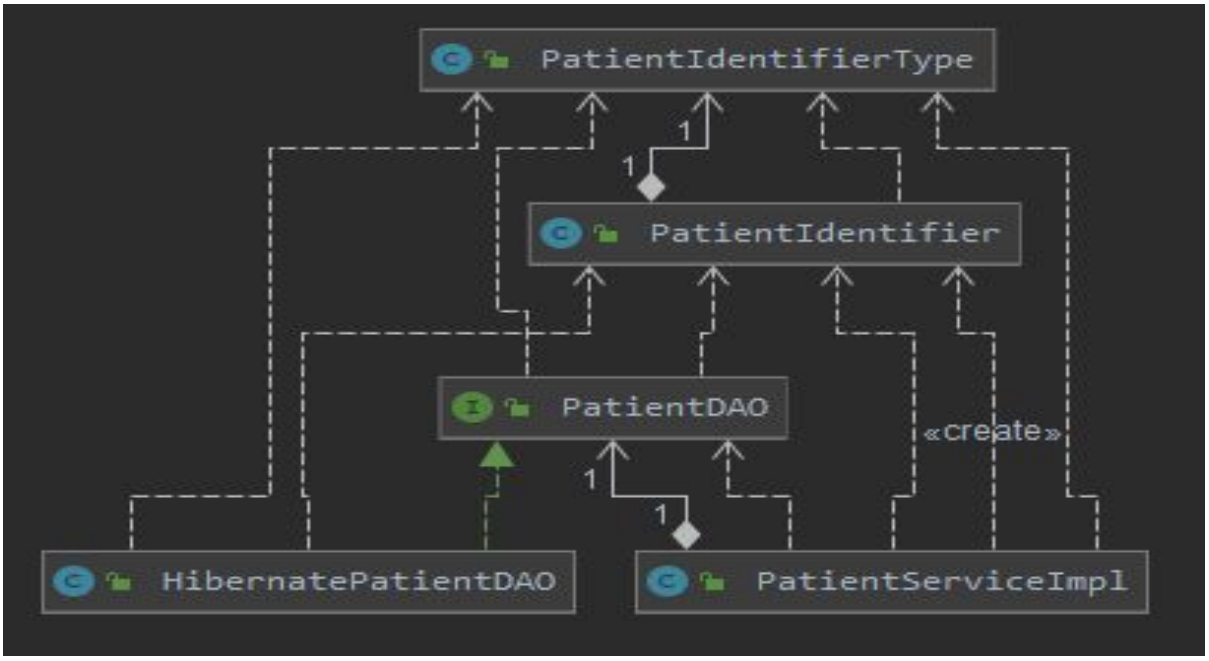


Figure 3.3: openmrs-core patient api structure

Above is the UML of openmrs-core api, since our team spent most of time dealing with issues inside the api, we have discovered that the openmrs-core api using abstract factory pattern to encapsulate a group of individual classes that have a common theme without specifying their concrete classes. In that way the whole application is more flexible and the structure can be changed easily. Besides, this also makes the developer easy to implement new modules/features into the application.

0.4 Project Management

In this section, we will record on how Team 21 manages the project in deliverable 3.

0.4.1 Kanban

Team 21 is continuing to use Kanban as our development process. In this deliverable, we are going to analyze two features, and choose one of them to map out the implementation plan and write test cases.

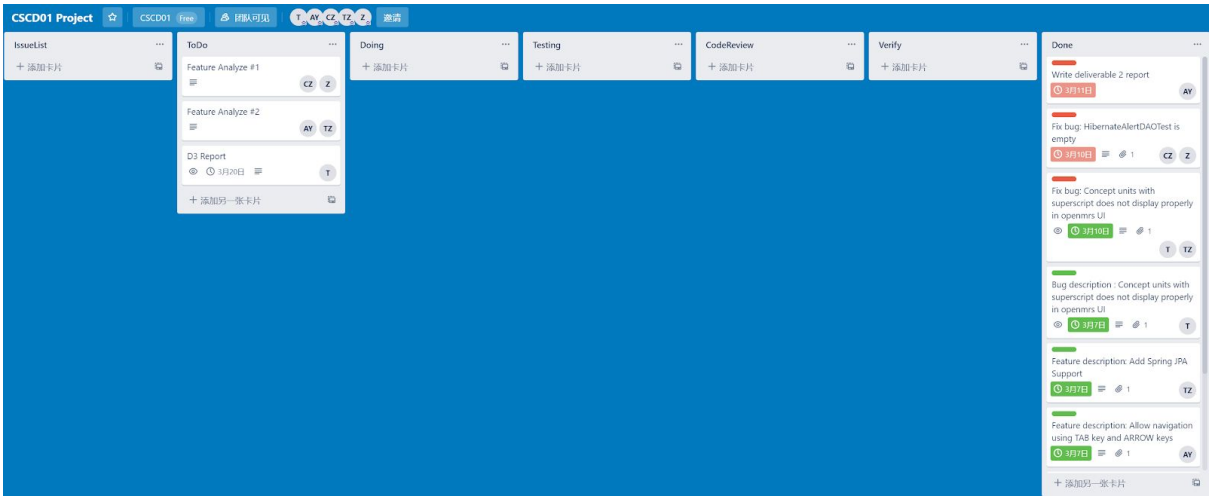


Figure 4.1: Kanban board (at the start of deliverable 3)

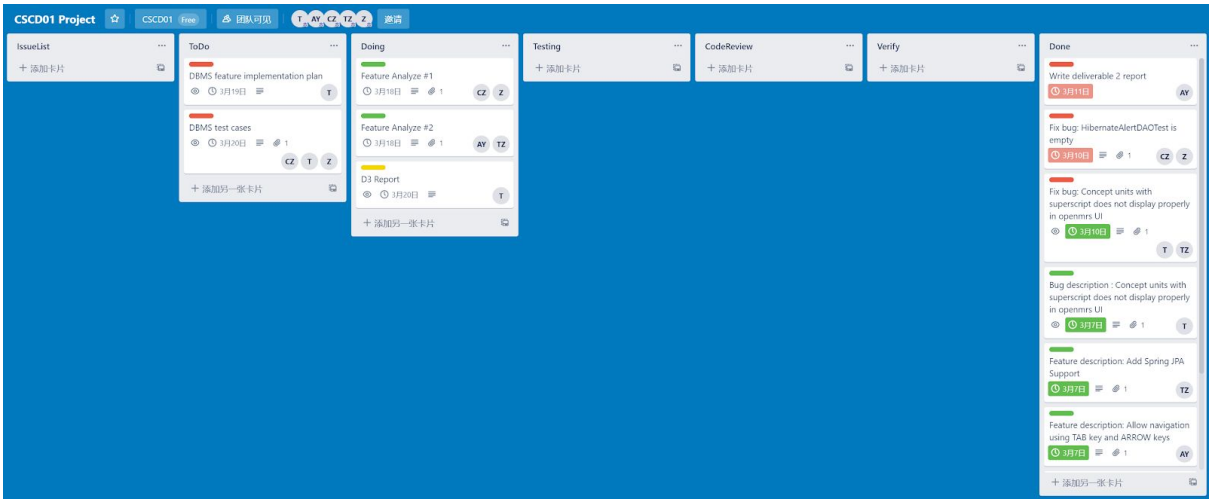


Figure 4.2: Kanban board (after the second meeting)

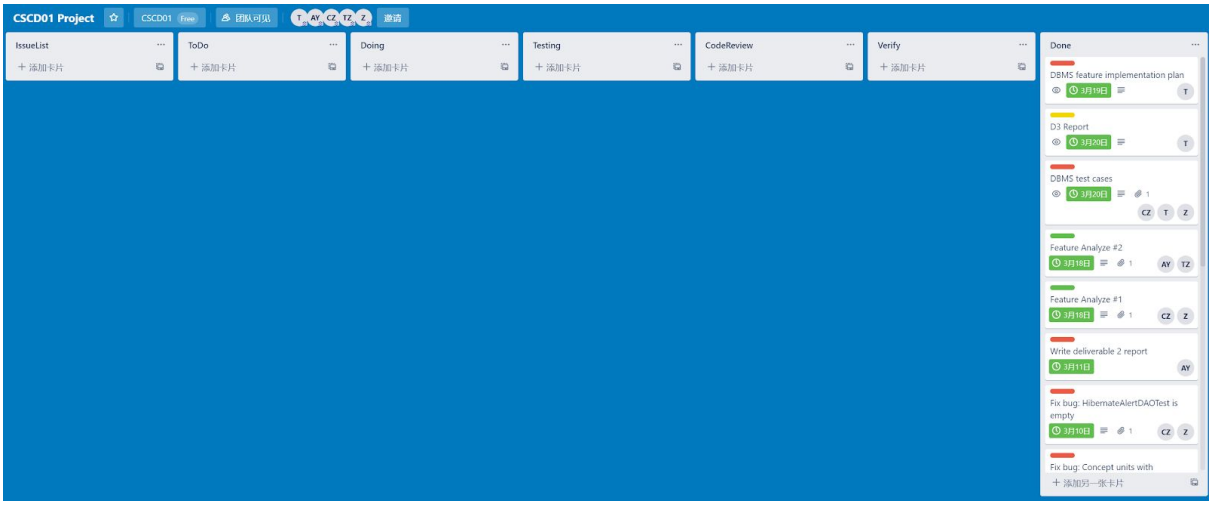


Figure 4.3: Kanban board (at the end of the deliverable 3)

0.4.2 Burn-up Chart

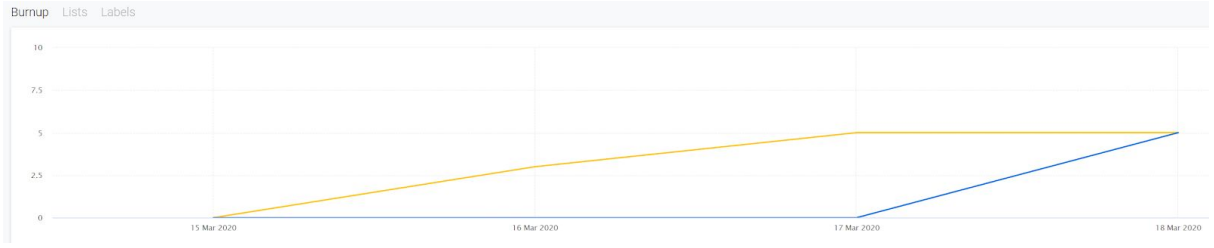


Figure 4.4: Burn-up Chart (generate by Corrello)

0.4.3 Standups

As part of the Kanban process, we are required to do daily standups. However, since we are students and our schedules are not very flexible, we settled on conducting 3 meetings each week.

Mar. 16th: We got everyone up to speed and came up with a plan for the deliverable. The plan is to have our group together to decide which two features we're going to analyze. And then have two teams of two work on two features while the leftover one writes the report.

Mar. 17th: We had a meeting to catch up on how everyone was doing, and we made the decision about the feature we're going to implement in deliverable 4.

Mar. 18th: We talked about the implementation plan for the chosen feature and produced the test cases together. After that, we finished the report together.