CSCD01 Team 21

Deliverable #4

Team 21: *The BugWriters*

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

March - April 2020

Contents

0.1 Feature Design

0.1.1 Code Design

0.1.2 UML

0.1.3 Changes with Implementation Plan

0.2 User Guide

0.3 Acceptance Test

0.4 Unit Test

0.5 Project Management

0.5.1 Kanban

0.5.2 Standups

0.5.3 Evidence Of Kanban

0.1 Feature Design

In this section, we will show you how our team designed the feature and how we followed the implementation plan we have created back in deliverable 3.

0.1.1 Code Design

All following modified java files are under the implemented_feature folder.

Implementation in PatientIdentifierType.java:

The PatientIdentifierType class is implemented inside PatientIdentifierType.java file, thus for adding the new auto merge feature, the first thing our team needs to do is make PatientIdentifierType object comparable. Hence, we need to implement a function called equalIdentifierType() which take another PatientIdentifierType object as a parameter, to compare the self object with the given one, if the two PatientIdentifierType objects consider as the duplicates, then the function will return true, otherwise, return false. The next thing we need to decide is to declare in what situation the two PatientIdentifierType objects are considered equal. As we can see in PatientIdentifierType.java, the PatientIdentifierType object has two main fields, which are id (integer) and format (string), thus if any one of those two fields of two PatientIdentifierType objects are the same, then the two objects are considered as the duplicates. But the class also contains other fields for example formatDescription (string) and validator (string), as our team did a further discovery, we finally decided three different cases of comparison of two PatientIdentifierType objects:

1. If two PatientIdentifierType objects have the same id or same format, then they are considered duplicates.
2. If two PatientIdentifierType objects have the same format description, then they are considered duplicates.
3. Otherwise, the two PatientIdentifierType objects are not equal.

Here is the code of equalIdentifierType() function:

```
216     /**
217      * the function checks the similarity of the current IdentifierType and the given IdentifierType
218      * if two consider the same then the function return true, otherwise return false
219      * @param patientIdentifier
220      * @return the IdentifierType is equal to the given one
221      */
222     public boolean equalIdentifierType(PatientIdentifierType patientIdentifier){
223         // if the id or the format is same, then two PatientIdentifierTypes are same
224         if((this.patientIdentifierTypeId == patientIdentifier.getPatientIdentifierTypeId())
225             || (this.format == patientIdentifier.getFormat())){
226             return true;
227         }
228         // if the formatDescription is same, then two PatientIdentifierTypes are same
229         else if (this.formatDescription != null) {
230             if (this.formatDescription == patientIdentifier.getFormatDescription()) {
231                 return true;
232             }
233         }
234         // else the two given PatientIdentifierTypes are not same then return false
235         else{
236             return false;
237         }
238     }
```

Figure 1.1: equalIdentifierType() function

Implementation in HibernatePatientDAO.java:

As we discussed in deliverable 3, the HibernatePatientDAO.java is the file that controls the update/delete of patient-related objects in the database. Therefore, to implement the auto-merge function we need to implement the function called check_Table() inside HibernatePatientDAO.java. The check_Table() function will take a PatientIdentifierType object as a parameter, the function is designed to communicate with the database and get all the existing PatientIdentifierType objects first, then the function will compare all PatientIdentifierType objects with the given parameter PatientIdentifierType, by using the above implemented comparison function equalIdentifierType(), if the function returns true, that means the two PatientIdentifierType are equal, and the check_Table() will merge them by deleting one of the objects from the database.

Here is the code of check_Table() function:

```
1000    /**
1001     * Check if there exists a duplicate patientIdentifierType
1002     */
1003    public Boolean check_Table(PatientIdentifierType patientIdentifierType) {
1004        // Get all patientIdentifierType
1005        Criteria criteria = sessionFactory.getCurrentSession().createCriteria(PatientIdentifierType.class);
1006
1007        criteria.addOrder(Order.desc("required"));
1008        criteria.addOrder(Order.asc("name"));
1009        criteria.addOrder(Order.asc("patientIdentifierTypeId"));
1010
1011        List<PatientIdentifierType> allList = criteria.list();
1012
1013        Boolean returnValue = false;
1014
1015        for (int i = 0; i < allList.size(); i ++) {
1016            if (allList.get(i).equalIdentifierType(patientIdentifierType) == true) {
1017                deletePatientIdentifierType(allList.get(i));
1018                returnValue = true;
1019            }
1020        }
1021        return returnValue;
1022    }
```

Figure 1.2: check_Table() function

Code from line 1005 to line 1011 is meant to send the required data to the database and get all PatientIdentifierType objects from the database and store them as a list. Code from line 1015 to line 1020 are meant to check every PatientIdentifierType objects with the given one by calling the equalIdentifierType() function we just implemented in PatientIdentifierType.java, if the two are equal, then the function will call deletePatientIdentifierType() function (this is a built-in function) to delete one of them from the database. Finally, at line 1021 check_Table() will return a boolean value, if there exists a duplicate and the function already merges them, then check_Table() will return true, otherwise, return false.

The above implementation only adds check and merge functionality, by implementing the auto-merge feature we also need to call check_Table() function whenever the database changes. Therefore, we need to rewrite the savePatientIdentifierType() function.

```java
    /**
     * @see org.openmrs.api.db.PatientDAO#savePatientIdentifierType(org.openmrs.PatientIdentifierType)
     */
    @Override
    public PatientIdentifierType savePatientIdentifierType(PatientIdentifierType patientIdentifierType) throws DAOException {
        sessionFactory.getCurrentSession().saveOrUpdate(patientIdentifierType);
        Boolean ifDuplicate = check_Table(patientIdentifierType);
        return patientIdentifierType;
    }
```

Figure 1.3: New savePatientIdentifierType() function

Now, whenever a new PatientIdentifierType object is inserted to the database, the savePatientIdentifierType() function will automatically check if there exists a duplicate and merge them if so.

0.1.2 UML

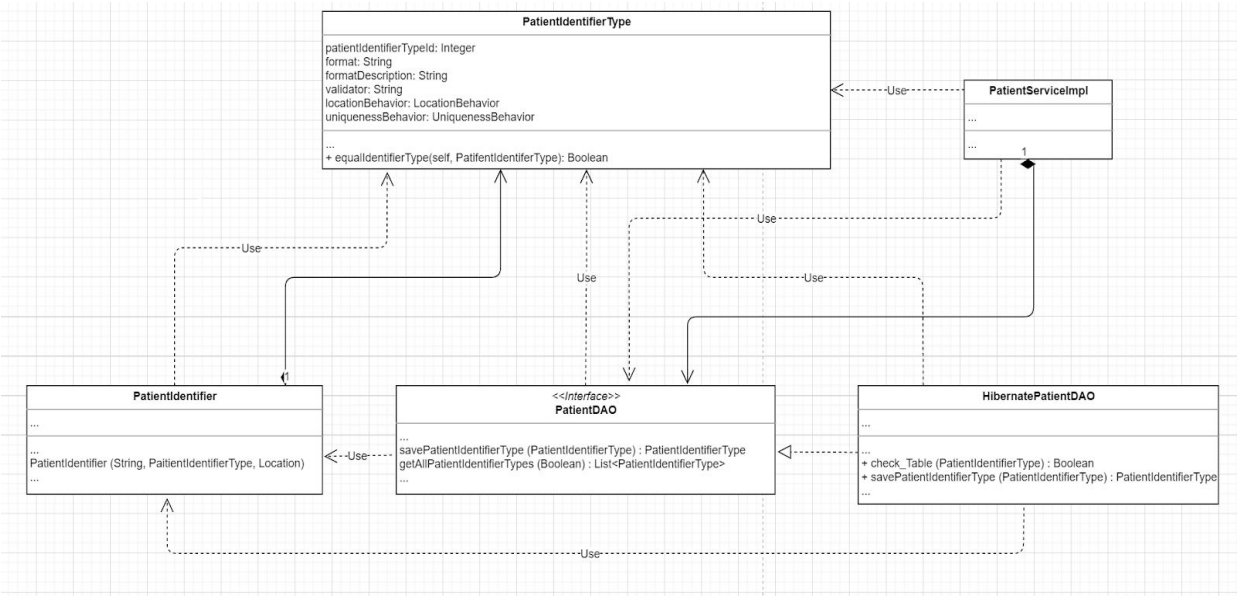

Figure 1.4: UML

0.1.3 Changes with Implementation Plan

The only change we made on the implementation plan in deliverable 3 is we had not modify anything in deletePatientIdentifierType() function in HibernatePatientDAO.java, because there is no need to track the database to find duplicates when delete a PatientIdentifierType object. Anything else is according to the original plan.

0.2 User Guide

The feature implemented by our team will give the OpenMRS software the functionality to auto-merge the duplicate patient identifier type whenever a new patient identifier type is added to the database. In particular, the user needs to call the savePatientIdentifierType() function from HibernatePatientDAO.java to save a new PatientIdentifierType object into the database.

```
303     /**
304      * @see org.openmrs.api.db.PatientDAO#savePatientIdentifierType(org.openmrs.PatientIdentifierType)
305      */
306         @Override
307     public PatientIdentifierType savePatientIdentifierType(PatientIdentifierType patientIdentifierType) throws DAOException {
308         sessionFactory.getCurrentSession().saveOrUpdate(patientIdentifierType);
309         Boolean ifDuplicate = check_Table(patientIdentifierType);
310         return patientIdentifierType;
311     }
```

Figure 2.1: Modified savePatientIdentifierType() function

And whenever the above function is called, the new feature will auto compare the given PatientIdentifierType object with every PatientIdentifierType object from the database, to check if there exists a duplicate. And for the definition of duplicate, our team believes if any two of PatientIdentifierType objects share the same id or have the same format, then they will be considered as duplicate. Then, if the duplicate exists, the new feature will auto-merge them by deleting one of them from the database.

Finally, if the user wishes to check the new feature to see if it really works, the user can use the following code after saving a new PatientIdentifierType object. The following code will print the id and the format of all PatientIdentifierType objects from the database. Then the user will be able to check if there exists a duplicate.

```
Criteria criteria = sessionFactory.getCurrentSession().createCriteria(PatientIdentifierType.class);

criteria.addOrder(Order.desc("required"));
criteria.addOrder(Order.asc("name"));
criteria.addOrder(Order.asc("patientIdentifierTypeId"));

for (PatientIdentifierType p : criteria.list()) {
   System.out.println("Id =" + t.getPatientIdentifierTypeId());
   System.out.println("Format =" + t.getFormat());

}
```

Figure 2.2: Code that prints id and format of all PatientIdentifierType objects

0.3 Acceptance Test

In deliverable 3 our team designed a series of acceptance tests, they are all able to run and are valid to test for different cases, but once we actually started our implementation work we found that there are some more cases we can test. Therefore, we redesigned the acceptance tests by implementing a number of test files, each of them representing a series of user's actions. The following are the final version of our acceptance test files, with a description of each file's functionality. All following acceptance test files are under the acceptance_tests folder:

- test_User1.java

  When a user saves two PatientIdentifierType objects into the database, one with the id = 1 and another one with the id = 2, then the user wants to get a list of all PatientIdentifierType objects in the database. As a result, the list should now contain two PatientIdentifierType objects.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import org.openmrs.PatientIdentifierType;
import org.openmrs.api.db.hibernate.HibernatePatientDAO;

public class test_User1 {

    private HibernatePatientDAO hibernatePatientDao;

    public static void main(string args[]) {
        updateSearchIndex();
        hibernatePatientDao = (HibernatePatientDAO) applicationContext.getBean("patientDAO");

        PatientIdentifierType type1 = new PatientIdentifierType(1);
        PatientIdentifierType type2 = new PatientIdentifierType(2);
        hibernatePatientDao.savePatientIdentifierType(type1);
        hibernatePatientDao.savePatientIdentifierType(type2);

        // The output should be 2
        System.out.print(hibernatePatientDao.getAllPatientIdentifierTypes(true).size());
    }

}
```

Figure 3.1: test_User1.java

- test_User2.java

  When a user saves two PatientIdentifierType objects into the database, both with the id = 1, then the user wants to get a list of all PatientIdentifierType objects in the database. As a result, the list should now contain only one PatientIdentifierType object.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import org.openmrs.PatientIdentifierType;
import org.openmrs.api.db.hibernate.HibernatePatientDAO;

public class test_User2 {

    private HibernatePatientDAO hibernatePatientDao;

    public static void main(string args[]) {
        updateSearchIndex();
        hibernatePatientDao = (HibernatePatientDAO) applicationContext.getBean("patientDAO");

        PatientIdentifierType type1 = new PatientIdentifierType(1);
        PatientIdentifierType type2 = new PatientIdentifierType(1);
        hibernatePatientDao.savePatientIdentifierType(type1);
        hibernatePatientDao.savePatientIdentifierType(type2);

        // The output should be 1
        System.out.print(hibernatePatientDao.getAllPatientIdentifierTypes(true).size());
    }

}
```

Figure 3.2: test_User2.java

- test_User3.java

  When a user saves two PatientIdentifierType objects into the database, one with the id = 1 and format = “format 1”, the other one with the id = 2 and format = “format 2”, then the user wants to get a list of all PatientIdentifierType objects in the database. As a result, the list should now contain two PatientIdentifierType objects.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import org.openmrs.PatientIdentifierType;
import org.openmrs.api.db.hibernate.HibernatePatientDAO;

public class test_User3 {

    private HibernatePatientDAO hibernatePatientDao;

    public static void main(string args[]) {
        updateSearchIndex();
        hibernatePatientDao = (HibernatePatientDAO) applicationContext.getBean("patientDAO");

        PatientIdentifierType type1 = new PatientIdentifierType(1);
        PatientIdentifierType type2 = new PatientIdentifierType(2);
        type1.setFormat("format 1");
        type2.setFormat("format 2");
        hibernatePatientDao.savePatientIdentifierType(type1);
        hibernatePatientDao.savePatientIdentifierType(type2);

        // The output should be 2
        System.out.print(hibernatePatientDao.getAllPatientIdentifierTypes(true).size());
    }

}
```

Figure 3.3: test_User3.java

- test_User4.java

  When a user saves two PatientIdentifierType objects into the database, one with the id = 1 and format = “format 1”, the other one with the id = 2 and format = “format 1”, then the user wants to get a list of all PatientIdentifierType objects in the database. As a result, the list should now contain only one PatientIdentifierType object.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import org.openmrs.PatientIdentifierType;
import org.openmrs.api.db.hibernate.HibernatePatientDAO;

public class test_User4 {

    private HibernatePatientDAO hibernatePatientDao;

    public static void main(string args[]) {
        updateSearchIndex();
        hibernatePatientDao = (HibernatePatientDAO) applicationContext.getBean("patientDAO");

        PatientIdentifierType type1 = new PatientIdentifierType(1);
        PatientIdentifierType type2 = new PatientIdentifierType(2);
        type1.setFormat("format 1");
        type2.setFormat("format 1");
        hibernatePatientDao.savePatientIdentifierType(type1);
        hibernatePatientDao.savePatientIdentifierType(type2);

        // The output should be 1
        System.out.print(hibernatePatientDao.getAllPatientIdentifierTypes(true).size());
    }

}
```

Figure 3.4: test_User4.java

- test_database1.java

  The database now contains two PatientIdentifierType objects, one with id = 1 and another with id = 2. A user wants to save a new PatientIdentifierType object with id = 1 into the database, then the user wants to get a list of all PatientIdentifierType objects in the database. As a result, the list should now contain only two PatientIdentifierType objects.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import org.openmrs.PatientIdentifierType;
import org.openmrs.api.db.hibernate.HibernatePatientDAO;

public class test_database1 {

    private HibernatePatientDAO hibernatePatientDao;

    public static void main(string args[]) {
        updateSearchIndex();
        hibernatePatientDao = (HibernatePatientDAO) applicationContext.getBean("patientDAO");

        PatientIdentifierType type1 = new PatientIdentifierType(1);
        PatientIdentifierType type2 = new PatientIdentifierType(2);
        hibernatePatientDao.savePatientIdentifierType(type1);
        hibernatePatientDao.savePatientIdentifierType(type2);

        PatientIdentifierType user_type = new PatientIdentifierType(1);
        hibernatePatientDao.savePatientIdentifierType(user_type);

        // The output should be 2
        System.out.print(hibernatePatientDao.getAllPatientIdentifierTypes(true).size());
    }

}
```

Figure 3.5: test_database1.java

- test_database2.java

  The database now contains two PatientIdentifierType objects, one with id = 1 and format = “format 1” and another with id = 2 and format = “format 2”. A user wants to save a new PatientIdentifierType object with id = 3 and format = “format 2” into the database, then the user wants to get a list of all PatientIdentifierType objects in the database. As a result, the list should now contain only two PatientIdentifierType objects.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import org.openmrs.PatientIdentifierType;
import org.openmrs.api.db.hibernate.HibernatePatientDAO;

public class test_database2 {

    private HibernatePatientDAO hibernatePatientDao;

    public static void main(string args[]) {
        updateSearchIndex();
        hibernatePatientDao = (HibernatePatientDAO) applicationContext.getBean("patientDAO");

        PatientIdentifierType type1 = new PatientIdentifierType(1);
        PatientIdentifierType type2 = new PatientIdentifierType(2);
        type1.setFormat("format 1");
        type2.setFormat("format 2");
        hibernatePatientDao.savePatientIdentifierType(type1);
        hibernatePatientDao.savePatientIdentifierType(type2);

        PatientIdentifierType user_type = new PatientIdentifierType(3);
        user_type.setFormat("format 2");

        // The output should be 2
        System.out.print(hibernatePatientDao.getAllPatientIdentifierTypes(true).size());
    }

}
```

Figure 3.6: test_database2.java

0.4 Unit Test

As we implement the feature by adding and modifying functions inside two java files, we also need two test files to do the unit tests. All following unit test files are under the unit_tests folder:

- PaitentIdentifierTypeTest.java:

  This file is used to test methods in PaitentIdentifierType.java

```java
public class PatientIdentifierTypeTest {

    /**
     * Test the equalIdentifierType method in PatientIdentifierType object
     * two PatientIdentifierType objects have same id, should return true
     */
    @Test
    public void same_id_objects() {
        PatientIdentifierType type1 = new PatientIdentifierType(1);
        PatientIdentifierType type2 = new PatientIdentifierType(1);

        assertEquals(true, type1.equalIdentifierType(type2));
    }

    /**
     * Test the equalIdentifierType method in PatientIdentifierType object
     * two PatientIdentifierType objects have different id, should return false
     */
    @Test
    public void diff_id_objects() {
        PatientIdentifierType type1 = new PatientIdentifierType(1);
        PatientIdentifierType type2 = new PatientIdentifierType(2);

        assertEquals(false, type1.equalIdentifierType(type2));
    }

    /**
     * Test the equalIdentifierType method in PatientIdentifierType object
     * two PatientIdentifierType objects have different id but same format, should return true
     */
    @Test
    public void diff_id_same_format_objects() {
        PatientIdentifierType type1 = new PatientIdentifierType(1);
        PatientIdentifierType type2 = new PatientIdentifierType(2);

        type1.setFormat("format 1");
        type2.setFormat("format 1");
        assertEquals(true, type1.equalIdentifierType(type2));
    }
```

Figure 4.1: PaitentIdentifierTypeTest.java (part 1)

```java
/**
 * Test the equalIdentifierType method in PatientIdentifierType object
 * two PatientIdentifierType objects have different id and different format, should return false
 */
@Test
public void diff_id_diff_format_objects() {
    PatientIdentifierType type1 = new PatientIdentifierType(1);
    PatientIdentifierType type2 = new PatientIdentifierType(2);

    type1.setFormat("format 1");
    type2.setFormat("format 2");
    assertEquals(false, type1.equalIdentifierType(type2));
}

/**
 * Test the equalIdentifierType method in PatientIdentifierType object
 * two PatientIdentifierType objects have different id and different format but same formatDescription, should return true
 */
@Test
public void diff_id_diff_format_same_describes_objects() {
    PatientIdentifierType type1 = new PatientIdentifierType(1);
    PatientIdentifierType type2 = new PatientIdentifierType(2);

    type1.setFormat("format 1");
    type2.setFormat("format 2");

    type1.setFormatDescription("this is format 1");
    type2.setFormatDescription("this is format 1");
    assertEquals(true, type1.equalIdentifierType(type2));
}

/**
 * Test the equalIdentifierType method in PatientIdentifierType object
 * two PatientIdentifierType objects have different id and different format and different formatDescription, should return true
 */
@Test
public void diff_id_diff_format_diff_describes_objects() {
    PatientIdentifierType type1 = new PatientIdentifierType(1);
    PatientIdentifierType type2 = new PatientIdentifierType(2);

    type1.setFormat("format 1");
    type2.setFormat("format 2");

    type1.setFormatDescription("this is format 1");
    type2.setFormatDescription("this is format 2");
    assertEquals(false, type1.equalIdentifierType(type2));
}
```

Figure 4.2: PaitentIdentifierTypeTest.java (part 2)

- HibernatePatientDAOTest.java:

  This file is used to test methods in HibernatePatientDAO.java.

```java
public class HibernatePatientDAOTest extends BaseContextSensitiveTest {

    private HibernatePatientDAO hibernatePatientDao;

    @Before
    public void beforeEach() {
        updateSearchIndex();
        hibernatePatientDao = (HibernatePatientDAO) applicationContext.getBean("patientDAO");
    }

    @Test
    public void getPatientIdentifiers_shouldGetByIdentifierType() {
        List<PatientIdentifierType> identifierTypes = singletonList(new PatientIdentifierType(2));
        List<PatientIdentifier> identifiers = hibernatePatientDao
                .getPatientIdentifiers(null, identifierTypes, emptyList(), emptyList(), null);
        List<Integer> identifierIds = identifiers.stream().map(PatientIdentifier::getId)
                .collect(Collectors.toList());

        Assert.assertEquals(2, identifiers.size());
        Assert.assertThat(identifierIds, hasItems(1, 3));
    }

    @Test
    public void getPatientIdentifiers_shouldGetByPatients() {
        List<Patient> patients = Arrays.asList(
                hibernatePatientDao.getPatient(6),
                hibernatePatientDao.getPatient(7)
        );
        List<PatientIdentifier> identifiers = hibernatePatientDao
                .getPatientIdentifiers(null, emptyList(), emptyList(), patients, null);
        List<Integer> identifierIds = identifiers.stream().map(PatientIdentifier::getId)
                .collect(Collectors.toList());

        Assert.assertEquals(2, identifiers.size());
        Assert.assertThat(identifierIds, hasItems(3, 4));
    }
```

Figure 4.3: HibernatePatientDAOTest.java (part 1)

```java
/**
 * Test the check_Table method in HibernatePatientDAO object
 * two PatientIdentifierType objects have same id, should return true
 */
@Test
public void testCheckTableSameId(){
    deleteAllIdentifierTypes();
    PatientIdentifierType identifierType = new PatientIdentifierType(1);
    hibernatePatientDao.savePatientIdentifierType(identifierType);
    assertTrue(hibernatePatientDao.check_Table(new PatientIdentifierType(1)));
}

/**
 * Test the check_Table method in HibernatePatientDAO object
 * two PatientIdentifierType objects have same format, should return true
 */
@Test
public void testCheckTableSameFormat(){
    deleteAllIdentifierTypes();
    PatientIdentifierType identifierType = new PatientIdentifierType(1);
    identifierType.setFormat("Format1");
    PatientIdentifierType identifierTypeSame = new PatientIdentifierType(2);
    identifierTypeSame.setFormat("Format1");
    hibernatePatientDao.savePatientIdentifierType(identifierType);
    assertTrue(hibernatePatientDao.check_Table(identifierTypeSame));
}

/**
 * Test the check_Table method in HibernatePatientDAO object
 * two PatientIdentifierType objects have different id, should return false
 */
@Test
public void testCheckTableDifferent(){
    deleteAllIdentifierTypes();
    PatientIdentifierType identifierType = new PatientIdentifierType(1);
    hibernatePatientDao.savePatientIdentifierType(identifierType);
    assertFalse(hibernatePatientDao.check_Table(new PatientIdentifierType(2)));
}
```

Figure 4.4: HibernatePatientDAOTest.java (part 2)

```java
/**
 * Test the check_Table method in HibernatePatientDAO object
 * if database is empty, then check_Table should return false
 */
@Test
public void testCheckTableVoid(){
    deleteAllIdentifierTypes();
    assertFalse(hibernatePatientDao.check_Table(new PatientIdentifierType(2)));
}

/**
 * function that delete all IdentifierTypes objects from database
 */
public void deleteAllIdentifierTypes(){
    List<PatientIdentifierType> identifierTypes = hibernatePatientDao.getAllPatientIdentifierTypes(true);
    for(PatientIdentifierType identifierType: identifierTypes){
        hibernatePatientDao.deletePatientIdentifierType(identifierType);
    }
}
```

Figure 4.5: HibernatePatientDAOTest.java (part 3)

The final version unit tests have some changes with the unit test we designed back in deliverable 3, we add more cases and write better comments in the test files. We meant to make each case easy to understand for any users.

0.5 Project Management

In this section, we will record how Team 21 manages the project in deliverable 4.

0.5.1 Kanban

Team 21 is continuing to use Kanban as our development process. In this final deliverable, we are going to implement the feature we choose from deliverable 3. As we follow the implementation plan, two of our members are going to implement and change the structure of the PaientIdenifierType.java (also writing a complete unit test suite), the other two will implement the changes inside HibernatePaientDAO.java (also write a complete unit test suite), the last member will create the user guide and an acceptance test suite. Finally, the whole team will finish the deliverable 4 report together.

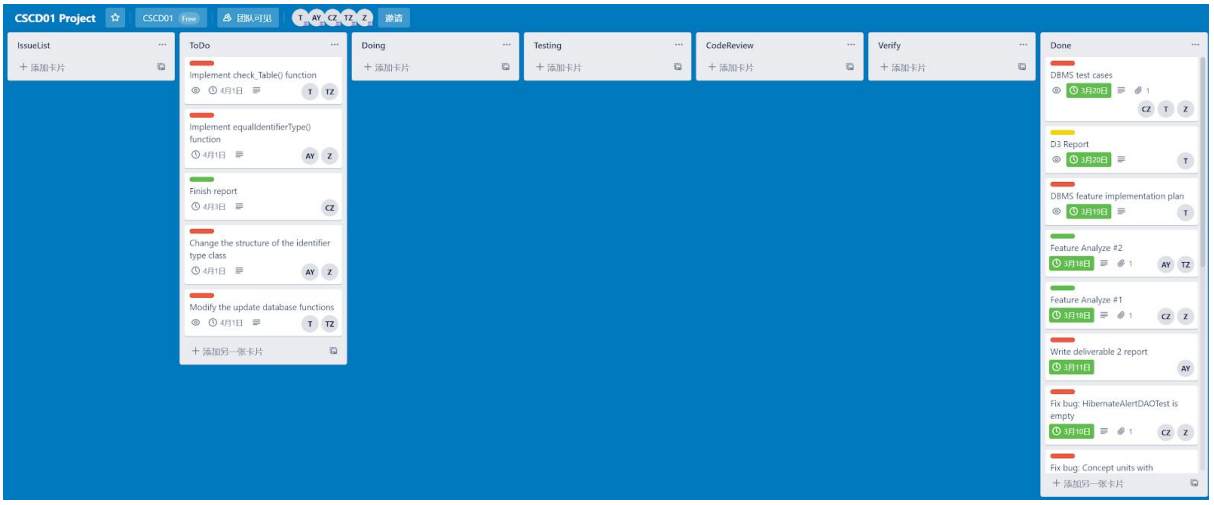

Figure 5.1: Kanban board (at the start of deliverable 4)

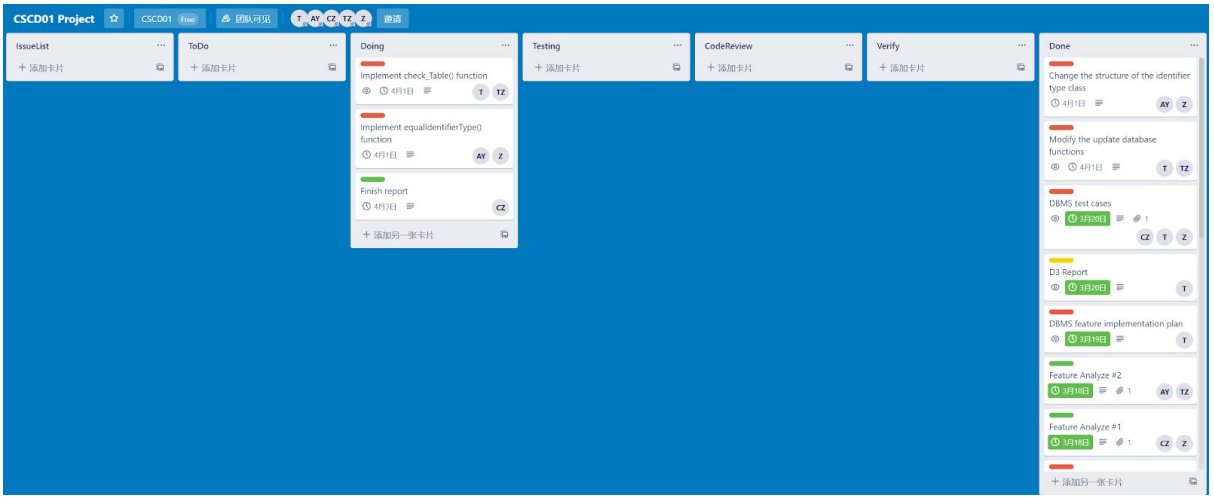

Figure 5.2: Kanban board (after the second meeting)

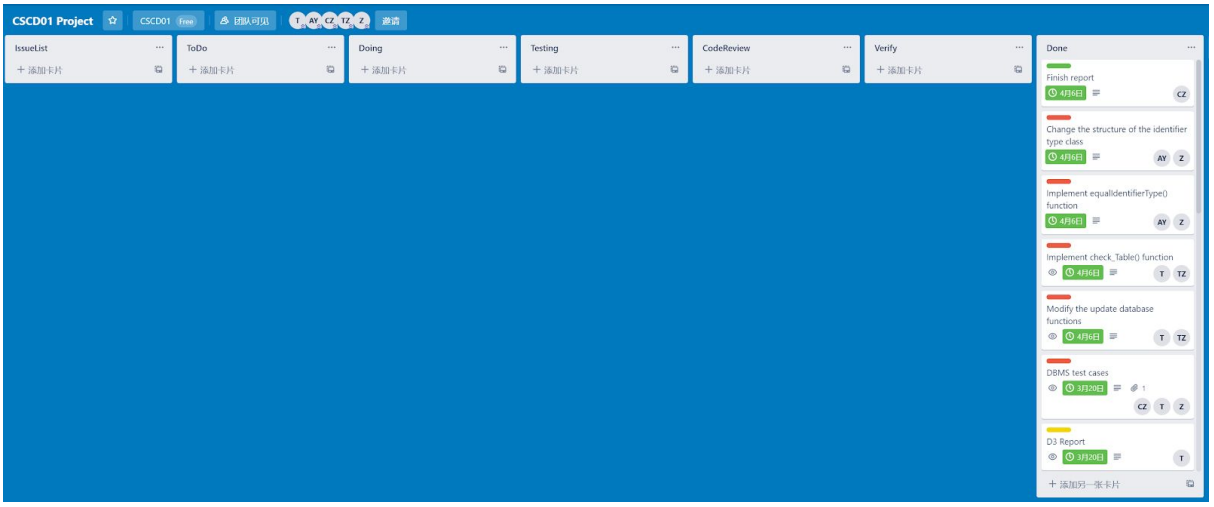

Figure 5.3: Kanban board (at the end of the deliverable 4)

0.5.2 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 4 meetings during the work period of deliverable 4.

Mar. 28th: We got everyone up to speed and reviewed the implementation plan from the deliverable 3. Then we decided to split our team into two groups of two working on two main parts of implementation while the leftover one writes the user guide and an acceptance test suite.

Mar. 30th: We had a meeting to catch up on how everyone was doing, and we talked about changes/differences from the original implementation plan.

April. 1st: We test the feature and make sure everything is fine. After that, we talked about adding more test cases.

(Add one, after Dr.Anya extended the due date)
April. 5th: We reviewed all of our work to make sure everything works fine. Then we check and fix the typo and grammar mistakes for the deliverable report before we make the final submission.

0.5.3 Evidence Of Kanban

As we described in earlier deliverables, Kanban is meant to maximize the speed and quality of output, which is suitable for large size program development. And from the two months of development experience with Kanban, our team very much benefited from it.
One thing that makes Kanban so good at large scale program development is that compared to other software development processes, the Kanban process is more flexible since it does not require software engineers to have exactly the same skill level, it allows the developer team to add tasks anytime during the work period. Since every member of our team has no experience in development in OpenMRS, we all practically learn new stuff (ex. OpenMRS structure, API structure) during development, thus it is very common that some tasks need to be changed or added during the process. Kanban's flexibility is helping us with such problems, every time our team decides to change something we just change it without a thought, we don't need to follow an unchangeable task till the end of a sprint like other development processes do, the flexibility saves us a bunch of time.
Another feature that made Kanban unique is limit work in progress (WIP), by limiting the number of tasks it will improve our focus on a few tasks and to identify bottlenecks. Our team has set the number of WIP to be five during the whole development period. It ensures that each member of the team has only one task to

do, such that the quality of our work will remain high and the workload of each person is very likely equal.
Overall, Kanban helps our team a lot during the work term, and we are glad to choose Kanban as our development process so that we can finish the development together.