Android Unit Testing

SEG3103[Z] Summer 2022 Project Group 19

Group members: Gong (Victor) Feng 300176400

Introduction and Abstract:

This project is to explore the testing processes of Android applications by integrating the knowledges and tools from the lectures and the assignments. The system under testing is a simple stackOverFlow Browser android application, which is constructed using MVC pattern. Since there are many external and internal dependencies while developing android application, the project focused on TestDouble and Mocking to simulate the external dependencies, and the coverage testing and mutation testing were also applied in this project to make sure there are sufficient test units to cover all functions and there is not coverage gap in the application. At the end, test driven development will be briefly introduced as the future learning path.

Android applications consist of several layers having different functionalities. UI layer contains UI logic, this logic is responsible for drawing whatever it is that the screen shows and capturing user input. Next layer is application layer. Logic that resides in this layer is responsible for controlling the application flowing. Then comes domain layer. Domain layer is responsible for encapsulating complex business logic, or simple logic that is reused by multiple ViewModels. The last layer is infrastructure layer. This layer contains the logic that implements generic functionality like networking, storage, event bus and so on.

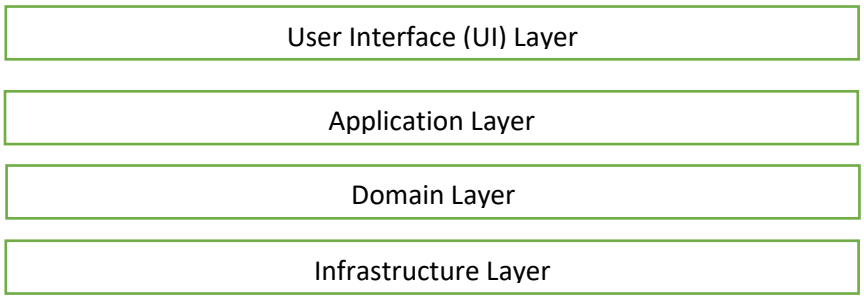


Figure 1: The Layers of Android Application

UI layer consists mainly of Android views, which are not unit testable. The infrastructure layer is super difficult to test. Therefore, this project focused the unit test for the logic in Application layer and Domain layer.

Brief overview of tools and techniques used in the project

First, all production code and test units were developed using Android studio, which is a platform to develop android applications, and it can be treated as a super set of Java— basic Java plus Android specific APIs. The tools and techniques were used in this project is TestDouble and Mockito, Coverage test and mutation test.

TestDouble and Mockito:

All objects need to interact in the object-oriented system. To test a system with dependencies, the system under test can not be instantiated without satisfying these dependencies. In order to test the

system, components with alternative implementations that the developer designs specifically to be used in the test can substitute these dependencies in the system under test. For example,

```
public FetchLastActiveQuestionsUseCase(FetchLastActiveQuestionsEndpoint fetchLastActiveQuestionsEndpoint) {
    mFetchLastActiveQuestionsEndpoint = fetchLastActiveQuestionsEndpoint;
}
```

To instantiate the class of FetchLastActiveQuestionUseCase, an object of FetchLastActiveQuestionsEndPoint should be passed in. A class is created to substitute FetchLastActiveQuestionsEndPoint:

```
private static class EndpointTd extends FetchLastActiveQuestionsEndpoint {
```

This class extends FetchLastActiveQuestionsEndPoint and override the functions which specifically would be used in the test. There alternative implementations are called TestDouble. The Mockito was used in this project to simplify the process of Mocking.

Using Mockito:

Use this statement to create a mocking:

```
@Mock FetchLastActiveQuestionsUseCase.Listener mListener1;
@Mock FetchLastActiveQuestionsUseCase.Listener mListener2;
```

Use when and thenReturn function to define alternative implements:

```
when(mLoginHttpEndpointSyncMock.loginSync(any(String.class), any(String.class)))
        .thenReturn(new LoginHttpEndpointSync.EndpointResult(LoginHttpEndpointSync.EndpointResultStatus
```

Use capture to capture the argument

```
@Captor ArgumentCaptor<List<Question>> mQuestionsCaptor;
```

Use verify to do the assertion

```
verify(mListener1).onLastActiveQuestionsFetched(mQuestionsCaptor.capture());
verify(mListener2).onLastActiveQuestionsFetched(mQuestionsCaptor.capture());
```

Coverage Test:

In android studio, Coverage test is very easy to use, just right click, then choose “run test by coverage”, the percentage of method coverage and line coverage will pop out, and the covered lines would have green color in the beginning of the line, the uncovered lines would have red color. Coverage Test was used in this project to make sure sufficient test units were developed to cover all the methods and lines.

Mutation Test:

Mutation test was applied in this project to detect the coverage gap. Since the test units were written after the production codes were developed, it is quite hard to make sure if all requirements were covered. The idea is that if the unit tests covered the entire functionalities of the system under test,

then any change in its internal implementation should cause at least one test to fail. In this project, all statements are manually mutated to make sure the test units covered all requirements.

Overview of the system/Software under test

The system under test is a simple stackOverFlow Browser android application, the view displays a list of questions, users can click one of the questions, after clicking, the users will be navigated to another view, which displays the detail information about the selected question.

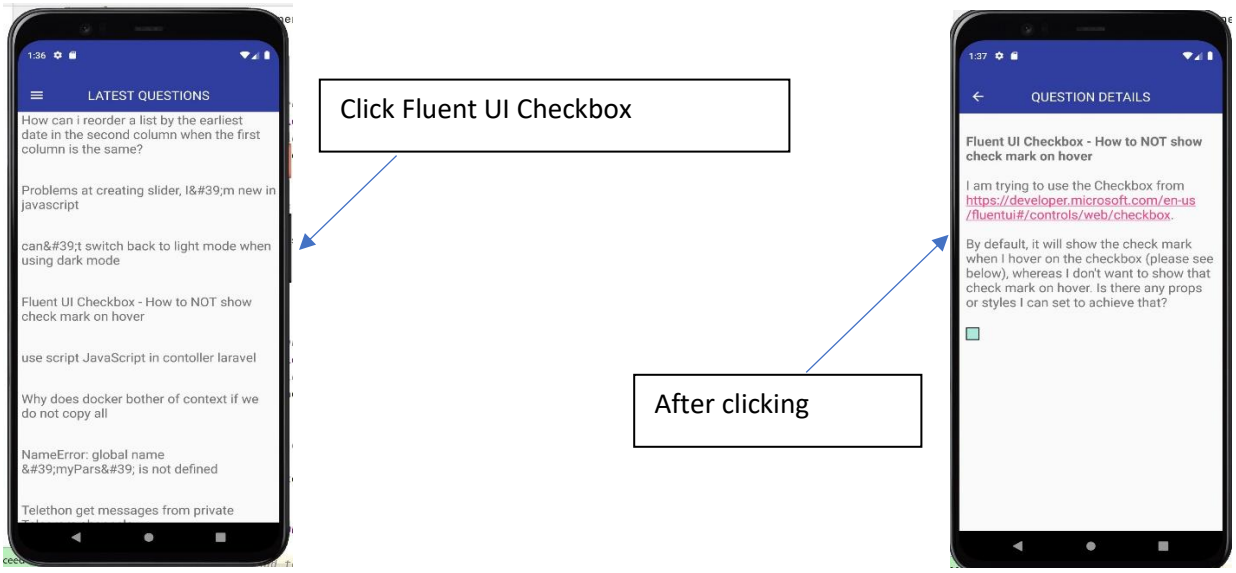


Figure 2: Views of system under test

This application is structured by MVC pattern, and there are several classes in this application (see Figure 3): FetchLastQuestionEndPoint can be treated as a server, when it receives the request from FetchLastActiveQuestionsUseCase, it sends a list of questions to FetchLastActiveQuestionsUseCase. The class FetchLastActiveQuestionsUseCase is the Model or the cache, which stores the questions retrieving from FetchLastQuestionEndPoint and passes the questions to the QuestionController. For QuestionController, it receives the questions from the FetchLastActiveQuestionsUseCase and passes to the QuestionListViewMVC, which is the user interface to display the list of questions. This application also uses the observer pattern, FetchLastActiveQuestionsUseCase and QuestionListViewMVC extend the abstract observer classes, and QuestionController implements the interfaces of the Listener in FetchLastActiveQuestionsUseCase and QuestionListViewMVC. Therefore, once QuestionController registers as a listener of FetchLastActiveQuestionsUseCase and QuestionListViewMVC, it will get notified for the changes in both Classes.

When the function onQuestionClicked() is invoked, the QuestionListController will get the signal and navigate the user to another view for the question details. The structure of the QuestionDetails is very similar to the QuestionList; therefore, this report will not introduce the structure and test units of QuestionDetails, but the codes and test units can be found in the Appendix.

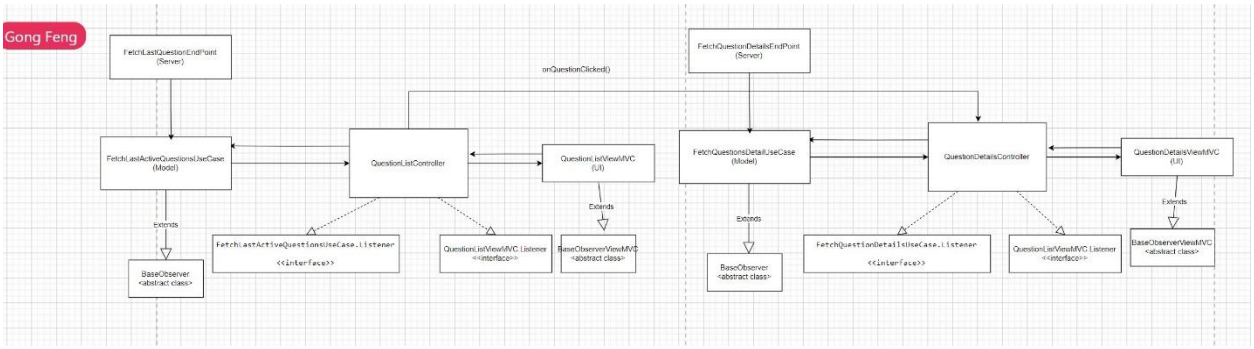


Figure 3: The structure of the System under test

Overview of the testing approaching

This project focuses on the testing of application layer and domain layer, the class of FetchLastActiveQuestionsUseCase resides in domain layer and QuestionListController is in application layer. Therefore, the first component that was tested is FetchLastActiveQuestionsUseCase, which fetches the list of questions from end point.

The codes of this class are relatively simple. An instance of FetchLastActiveQuestionsEndPoint must be passed to construct a FetchLastActiveQuestionsUseCase, and it has an interface of Listener, and it extends from the BaseObservable class. Therefore, the logic behind that is FetchLastActiveQuestionsUseCase invokes the method fetchLastActiveQuestionsAndNotify() to fetch the data from FetchLastActiveQuestionsEndPoint and notify its listeners, so it has two external dependencies. To implement tests with external dependencies, test double and Mockito were used.

```
public class FetchLastActiveQuestionsUseCase extends BaseObservable<FetchLastActiveQuestionsUseCase.Listener> {

    public interface Listener {...}

    private final FetchLastActiveQuestionsEndpoint mFetchLastActiveQuestionsEndpoint;

    public FetchLastActiveQuestionsUseCase(FetchLastActiveQuestionsEndpoint fetchLastActiveQuestionsEndpoint) {..

    public void fetchLastActiveQuestionsAndNotify() {...}

    private void notifyFailure() {...}

    private void notifySuccess(List<QuestionSchema> questionSchemas) {...}
}
```

Before the testing, an Endpoint test double and two mock listeners were declared to simulate the class of FetchLastActiveQuestionsEndPoint and the interface of Listener.

```
private EndpointTd mEndpointTd;
//I use Mokito to create two mock objects of listener to get the notification from FetchLastActiveQuestionU
@Mock FetchLastActiveQuestionsUseCase.Listener mListener1;
@Mock FetchLastActiveQuestionsUseCase.Listener mListener2;

//use this one to capture the argument
@Captor ArgumentCaptor<List<Question>> mQuestionsCaptor;
// endregion helper fields ------------------------------------------------------------------------

FetchLastActiveQuestionsUseCase SUT;
```

To define the behaviors of the Endpoint test double, it extends FetchLastActiveQuestionsEndPoint and overrides the method fetchLastActiveQuestion(Listener listener) in FetchLastActiveQuestionsEndPoint.

```
private static class EndpointTd extends FetchLastActiveQuestionsEndpoint {

    public boolean mFailure;

    public EndpointTd() { super( stackoverflowApi: null); }

    @Override
    public void fetchLastActiveQuestions(Listener listener) {
        if (mFailure) {
            listener.onQuestionsFetchFailed();
        } else {
            List<QuestionSchema> questionSchemas = new LinkedList<>();
            questionSchemas.add(new QuestionSchema( title: "title1",  id: "id1",  body: "body1"));
            questionSchemas.add(new QuestionSchema( title: "title2",  id: "id2",  body: "body2"));
            listener.onQuestionsFetched(questionSchemas);
        }
    }
}
// endregion helper classes
```

The logic is if the Endpoint receives the signal successfully, it will send a list of questions to FetchLastActiveQuestionsUseCase, if the Endpoint fails to receive the signal, the FetchLastActiveQuestionsUseCase will get notified of failure.

First, set up the environment before testing:

```
@Before
public void setup() throws Exception {
    mEndpointTd = new EndpointTd();
    SUT = new FetchLastActiveQuestionsUseCase(mEndpointTd);
}
```

Then, the first test unit is to test if the Endpoint receive the signal successfully, FetchLastActiveQuestionsUseCase will be notified with correct data:

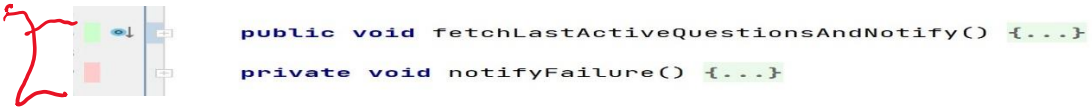
```
@Test
public void fetchLastActiveQuestionsAndNotify_success_listenersNotifiedWithCorrectData() throws Exception {
    // Arrange
    success();
    SUT.registerListener(mListener1);
    SUT.registerListener(mListener2);
    SUT.fetchLastActiveQuestionsAndNotify();
    // Assert
    verify(mListener1).onLastActiveQuestionsFetched(mQuestionsCaptor.capture());
    verify(mListener2).onLastActiveQuestionsFetched(mQuestionsCaptor.capture());
    List<List<Question>> questionLists = mQuestionsCaptor.getAllValues();
    assertThat(questionLists.get(0), is(QUESTIONS));
    assertThat(questionLists.get(1), is(QUESTIONS));
}
```

Then run the test with coverage:

42% classes, 28% lines covered in package 'com.techyourchance.unittesting.questions'

Element	Class, %	Method, %	Line, %
FetchLastActiveQuestionsUseCase	100% (2/2)	75% (6/8)	73% (17/23)

Only 75% of methods and 73% of lines were covered, and the covered methods and lines have green color in the beginning in the production codes and the uncovered methods and lines have red color in the beginning:

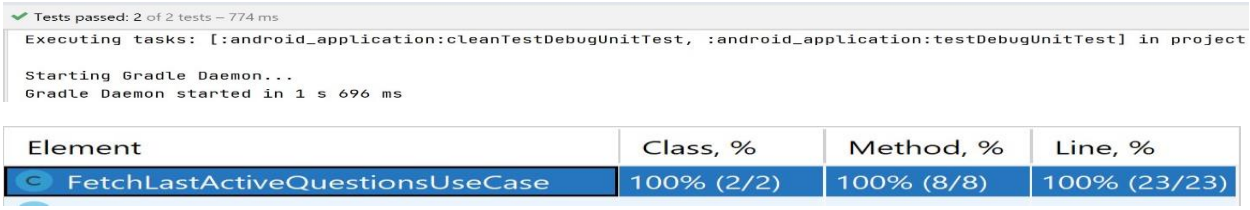


the coverage less than 100% means the test units are not sufficient to cover all the functionalities of the system under test, so more test units should be created.

And the next test unit is to test if the Endpoint fails to receive the signal, FetchLastActiveQuestionsUseCase will get notified of failure.

```
@Test
public void fetchLastActiveQuestionsAndNotify_failure_listenersNotifiedOfFailure() throws Exception {
    // Arrange
    //if questions are not fetched
    failure();
    SUT.registerListener(mListener1);
    SUT.registerListener(mListener2);
    // Act, called method fetchLastActiveQuestionAndNotify()
    SUT.fetchLastActiveQuestionsAndNotify();
    // Assert
    verify(mListener1).onLastActiveQuestionsFetchFailed();
    verify(mListener2).onLastActiveQuestionsFetchFailed();
}
```

After that, running test with coverage to make sure all functions and all lines of the system under test were covered by test units.

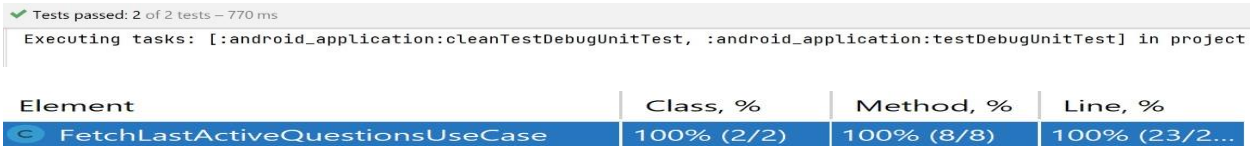


Although the method coverage and line coverage are 100%, it is not sufficient to make sure the test units exercise the entire functionalities of the system under test. For instance, if two statements in one of the test units are commented out. Basically, this test unit does not test anything, because the assertion part was commented out.

```
    @Test
    public void fetchLastActiveQuestionsAndNotify_success_listenersNotifiedWithCorrectData() throws Exception
        // Arrange
        success();
        SUT.registerListener(mListener1);
        SUT.registerListener(mListener2);
        SUT.fetchLastActiveQuestionsAndNotify();
        // Assert
        verify(mListener1).onLastActiveQuestionsFetched(mQuestionsCaptor.capture());
        verify(mListener2).onLastActiveQuestionsFetched(mQuestionsCaptor.capture());
        List<List<Question>> questionLists = mQuestionsCaptor.getAllValues();
//        assertThat(questionLists.get(0), is(QUESTIONS));
//        assertThat(questionLists.get(1), is(QUESTIONS));
    }
```

comment out

Then, run the test with coverage again.



The method coverage and line coverage were still 100 percent.

That is the reason the mutation test was used in this project to detect the coverage gap and to make sure the test units cover all edge cases. Since the codes in the class of FetchLastActiveQuestionsUseCase

are relatively simple, the mutation test tools were not used in this project. The statements were manually changed to make sure some tests fail. The idea is that if the unit tests covered the entire functionalities of the system under test, then any change in its internal implementation should cause at least one test to fail. For instance, one of the statements was commented out as following:

```
    private void notifyFailure() {
        for (Listener listener : getListeners()) {
//          listener.onLastActiveQuestionsFetchFailed();
        }
    }
```

comment out

Then, rerun the test:

```
Tests failed: 1, passed: 1 of 2 tests – 748 ms

Wanted but not invoked:
mListener1.onLastActiveQuestionsFetchFailed();
-> at com.techyourchance.unittesting.questions.FetchLastActiveQuestionsUseCaseTest.fetchLastActiveQuestionsAndNotify_failure_listenersNotifie
Actually, there were zero interactions with this mock.
```

One of the tests failed, which means this specific statement is covered by the test units.

Take another complex example:

```
    private void notifySuccess(List<QuestionSchema> questionSchemas) {
        List<Question> questions = new ArrayList<>(questionSchemas.size());
        for (QuestionSchema questionSchema : questionSchemas) {
//          questions.add(new Question(questionSchema.getId(), questionSchema.getTitle()));
            questions.add(new Question( id: "", questionSchema.getTitle()));
        }
        for (Listener listener : getListeners()) {
            listener.onLastActiveQuestionsFetched(questions);
        }
    }
```

Instead of passing in an actual ID of the question, empty string was passed in, then rerun the test:

```
Tests passed: 2 of 2 tests – 324 ms
Executing tasks: [:android_application:cleanTestDebugUnitTest, :android_application:testDebugUnitTest] in project C:\Users\gn927\Desktop\QA\p
```

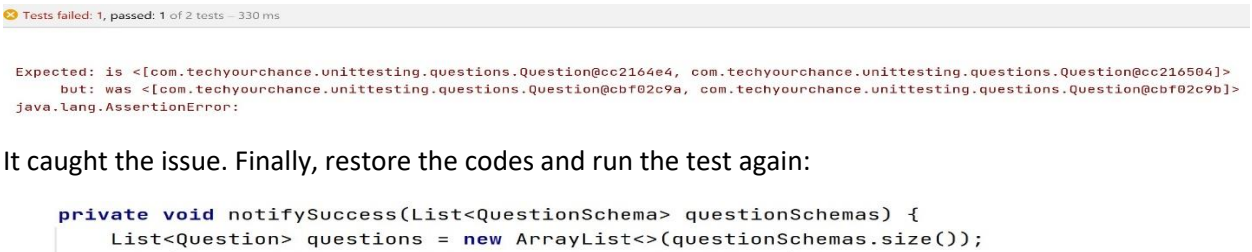
Test units passed, which means the test units did not catch the issue; therefore, the requirement that the ID of the question in FetchLastActiveQuestionsUseCase is the same ID that the Endpoint send to FetchLastActiveQuestionsUseCase, is not covered by the test units, so there is a gap in the test units.

After that, uncommented the assertion in the test units to ensure the data passed to FetchLastActiveQuestionsUseCase must be correct:

```
@Test
public void fetchLastActiveQuestionsAndNotify_success_listenersNotifiedWithCorrectData() throws Exception {
    // Arrange
    success();
    SUT.registerListener(mListener1);
    SUT.registerListener(mListener2);
    SUT.fetchLastActiveQuestionsAndNotify();
    // Assert
    verify(mListener1).onLastActiveQuestionsFetched(mQuestionsCaptor.capture());
    verify(mListener2).onLastActiveQuestionsFetched(mQuestionsCaptor.capture());
    List<List<Question>> questionLists = mQuestionsCaptor.getAllValues();
    assertThat(questionLists.get(0), is(QUESTIONS));
    assertThat(questionLists.get(1), is(QUESTIONS));
}
```

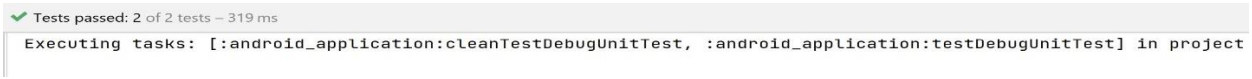
uncomment

Run the test again:



It caught the issue. Finally, restore the codes and run the test again:

```
    private void notifySuccess(List<QuestionSchema> questionSchemas) {
        List<Question> questions = new ArrayList<>(questionSchemas.size());
        for (QuestionSchema questionSchema : questionSchemas) {
            questions.add(new Question(questionSchema.getId(), questionSchema.getTitle()));
//              questions.add(new Question("", questionSchema.getTitle()));
        }
        for (Listener listener : getListeners()) {
            listener.onLastActiveQuestionsFetched(questions);
        }
    }
```



All testes passed.

Basically, to make sure the test units cover the internal functionalities of the system under test, open the production codes and start changing individual statement and make sure that on any change, at least one of the tests fails; otherwise, this specific requirement is not covered by the test units.

Test units for QuestionController can be found in appendixes.

Critical Evaluation of the System under test

The tested application is very simple, just a StackOverFlow Browser application; however, the test units which we developed for this application are quite complex. Since the application is structured by MVC pattern and Observer Patter. We need to use techniques of TestDouble and Mockito a lot to substitute the external and internal dependencies, and we not only need to test the class in domain layer like FetchLastActiveQuestionsUseCase, FetchQuestionDetailsUseCase, also we need to test the class in application layer like QuestionListController and QuestionController, for example, we need to test if the app OnStart(), the question list will be fetched and binded to the view or not. Since this app also use Observer patter, we also need to test if OnStart(), the controller will be registered as an Listener to get the notification of the changes or not, and OnStop(), the controller will be unregistered. The limitation of the project is we cannot test the functions in the infrastructure layer, for instance, the class Activity, the TestDouble of Activity inherit thousands of codes from Activity, which make test difficult and unreliable.

Critical Evaluation of tools

Android Studio is very convenient, to use coverage test , we just simply right click and chose test with coverage; to use Mockito, we just import Mockito in the beginning;

```
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

import java.util.LinkedList;
import java.util.List;

import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.CoreMatchers.notNullValue;
import static org.hamcrest.CoreMatchers.nullValue;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.verify;
```

There are tools for mutation test in Android studio, for instance, the mutation test can be set up with PIT and Robolectric in Android studio.

Critical Evaluation of testing techniques

The techniques of TestDouble and Mockito are super useful, because almost all systems are object-oriented system nowadays, and there are many interactions between different components and different objects. Use this technique, every component can be tested independently. For example, a app is structured by MVC pattern, one developer is assigned to develop controller, and another is assigned to design model. Once the developer which develop controller finishes, he does not need to wait the model to be fully developed to test the codes in the controller, because he can use TestDouble or Mockito to simulate the functions in model.

Conclusion and summary

To develop test units after the production codes are done, we use TestDouble and Mockito to substitute dependencies, and we use coverage test make sure test units are sufficient to test all the methods and lines, if the coverage is less than 100%, which means there are not enough test units to exercise all the requirement. However, although 100% coverage cannot ensure all requirements are covered, we still need to do mutation test to make sure there is not coverage gap.

However, there is a strategy called Test Drive Development, which can make sure there is not coverage gap, use this strategy, the coverage test and mutation test are not needed.

The strategy is:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to any more of a unit test than is sufficient to fail; and compilation failure is a failure.
3. You are not allowed to write any production code than is sufficient to pass the one failing unit test.

This strategy is the future path we are going to learn and keeping going in the software testing.

Appendix:

The test units of QuestionsListController:

1.

```
//test if the application is turned on, the progress indication would be shown
@Test
public void onStart_progressIndicationShown() throws Exception {
    // Arrange
    // Act
    //call onStart() means the application is invoked
    SUT.onStart();
    // Assert
    //check if the method showProgressIndication() was called, if the application has been invoked
    verify(mQuestionsListViewMvc).showProgressIndication();
}
```

2.

```
//test if the application is turned on for a while, the progress indication would be hidden
@Test
public void onStart_successfulResponse_progressIndicationHidden() throws Exception {
    // Arrange
    success();
    // Act
    SUT.onStart();
    // Assert
    //check if the method hideProgressIndication() was called, if the application has been invoked
    verify(mQuestionsListViewMvc).hideProgressIndication();
}
```

3.

```
//test if the application is turned on, but the questions can not be fetched from EndPoint
// the progress indication would be hidden
@Test
public void onStart_failure_progressIndicationHidden() throws Exception {
    // Arrange
    failure();
    // Act
    SUT.onStart();
    // Assert
    //check if the method hideProgressIndication() was called, if the application has been invoked
    verify(mQuestionsListViewMvc).hideProgressIndication();
}
```

4.

```
//test if the application is turned on, and the questions are fetched successfully from EndPoint
//the questions will be displayed in the view.
@Test
public void onStart_successfulResponse_questionsBoundToView() throws Exception {
    // Arrange
    success();
    // Act
    SUT.onStart();
    // Assert
    //check if the method bindQuestions() was called, if the questions are fetched successfully from EndPoint
    verify(mQuestionsListViewMvc).bindQuestions(QUESTIONS);
}
```

5.

```
//Test if the questions are stored in the cache, questions will be displayed in the view directly, no need to
//call the method fetchLastActiveQuestionsAndNotify()
@Test
public void onStart_secondTimeAfterSuccessfulResponse_questionsBoundToTheViewFromCache() throws Exception {
    // Arrange
    success();
    // Act
    SUT.onStart();
    //call onStart() twice, which means that the questions are still in the cache after calling onStart() first time
    SUT.onStart();
    // Assert
    //check if the method bindQuestion() was called exactly twice
    verify(mQuestionsListViewMvc, times( wantedNumberOfInvocations: 2)).bindQuestions(QUESTIONS);
    //check if it only call the method fetchLastActiveQuestionsAndNotify() once
    assertThat(mUseCaseTd.getCallCount(), is( value: 1));
}
```

6.

```
//test when questions are not fetched, the error toast will be displayed in the view
@Test
public void onStart_failure_errorToastShown() throws Exception {
    // Arrange
    //set condition to be failure.
    failure();
    // Act
    SUT.onStart();
    // Assert
    //check if the method showUseCassError will be called.
    verify(mToastsHelper).showUseCaseError();
}
```

7.

```
//test when questions are not fetched, no questions will be displayed in the view
@Test
public void onStart_failure_questionsNotBoundToView() throws Exception {
    // Arrange
    //set condition to be failure
    failure();
    // Act
    SUT.onStart();
    // Assert
    //check if the method bindQuestions will never be called
    verify(mQuestionsListViewMvc, never()).bindQuestions(any(List.class));
}
```

8.

```
//test if the controller is registered as a listener of the view and the fetchLastActiveQuestionsUseCase when
// the application is invoked
@Test
public void onStart_listenersRegistered() throws Exception {
    // Arrange
    // Act
    SUT.onStart();
    // Assert
    //check if the method registerListener was called
    verify(mQuestionsListViewMvc).registerListener(SUT);
    //check if the controller is the one of the listener of mUseCaseTd
    mUseCaseTd.verifyListenerRegistered(SUT);
}
```

9.

```
//test if the controller is unregistered when the application is stopped
@Test
public void onStop_listenersUnregistered() throws Exception {
    // Arrange
    SUT.onStart();
    // Act
    SUT.onStop();
    // Assert
    //check if the method unregisterListener was called
    verify(mQuestionsListViewMvc).unregisterListener(SUT);
    //check if the controller is not longer the one of the listener of mUseCaseTd
    mUseCaseTd.verifyListenerNotRegistered(SUT);
}
```

10.

```
//test when user click one of the questions, the user will be navigated to the detail of this question
@Test
public void onQuestionClicked_navigatedToQuestionDetailsScreen() throws Exception {
    // Arrange
    // Act
    SUT.onQuestionClicked(QUESTION);
    // Assert
    //check if the method toQuestionDetails() was called
    verify(mScreensNavigator).toQuestionDetails(QUESTION.getId());
}
```

All production codes and test units can be found in My GitHub:

https://github.com/GongVictorFeng/projectOfTesting-