Rambler&Co 85,47
Компания

Подпи

24 марта 2016 в 11:29

Построение Android приложений шаг за шагом, часть третья

Тестирование мобильных приложений*, Разработка под Android*, Разработка мобильных приложений*, Блог компании Rambler&Co

В первой и второй частях статьи мы создали приложение для работы с Github, внедрили Dagger 2 и покрыли код unit тестами. В заключитель части мы напишем интеграционные и функциональные тесты, рассмотрим технику TDD и напишем с ее применением новую функциональнос также подскажем, что читать дальше.

Полное содержание

Введение

В первой части статьи мы в два этапа создали простое приложение для работы с github. Архитектура приложения была разбита на две части

простую и сложную. Во второй части мы внедрили Dagger 2 и покрыли код unit тестами с помощью Robolectric, Mockito, MockWebServer и JaC

Условная схема приложения

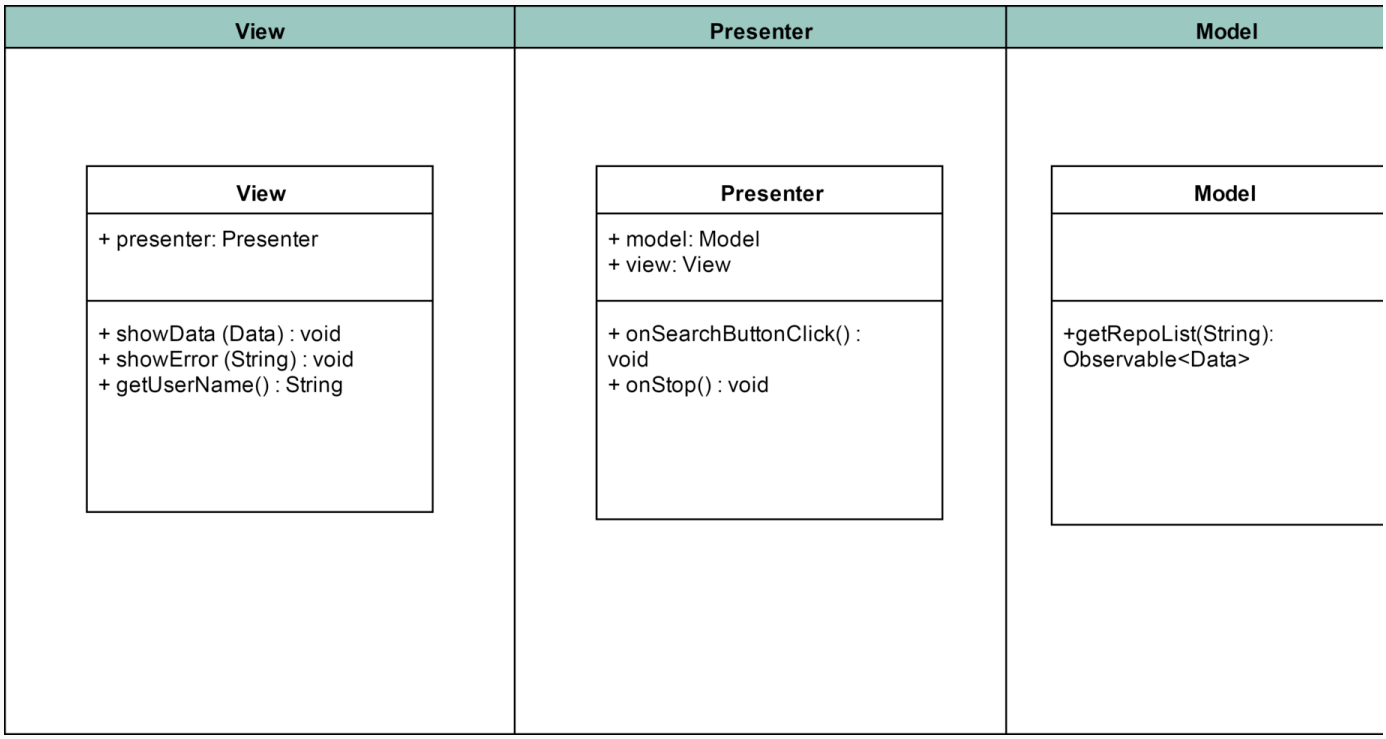
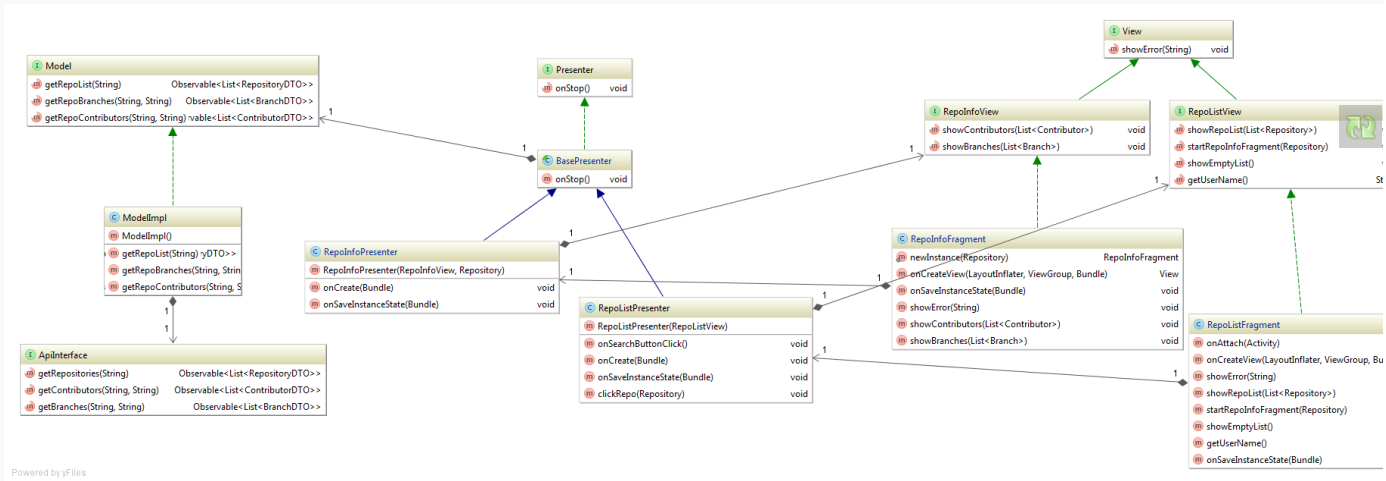


Диаграмма классов



Покрытие тестами

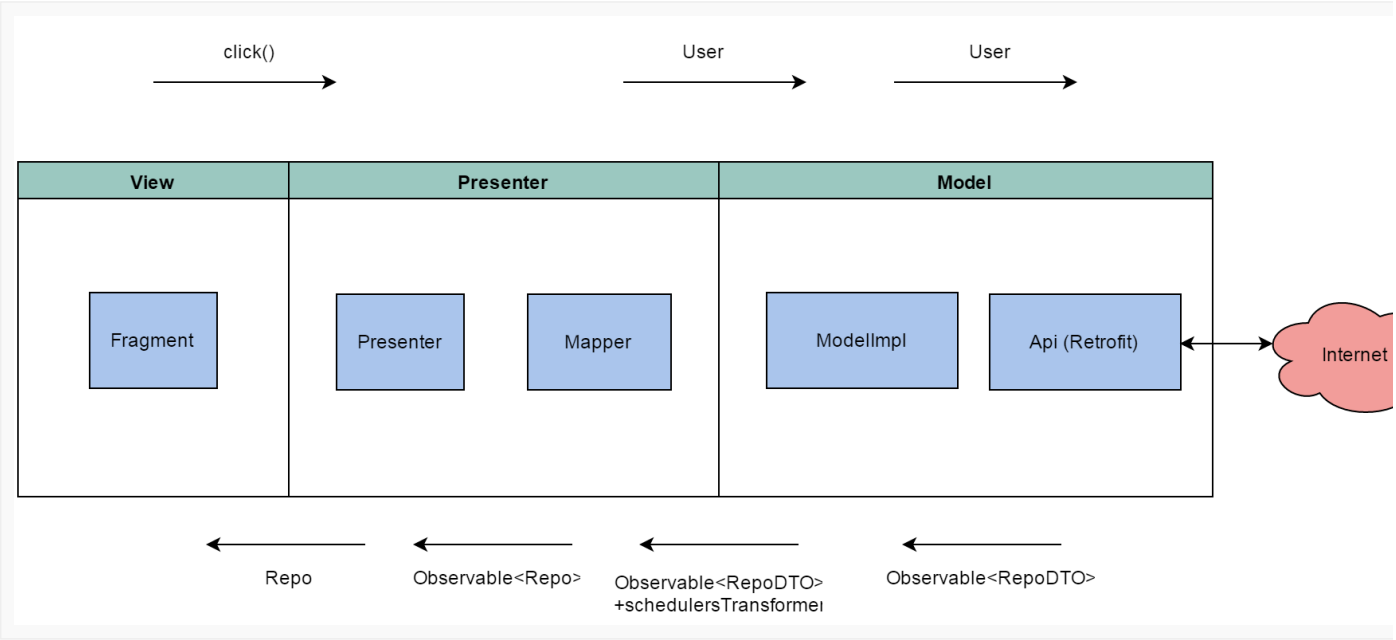
app

app

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Cla
com.andrey7mel.stepbystep.view.adapters		17%		n/a	9	12	24	31	9	12	3	
com.andrey7mel.stepbystep.view.fragments		74%		50%	10	25	19	72	9	24	0	
com.andrey7mel.stepbystep.view		84%		50%	3	7	2	17	1	5	0	
com.andrey7mel.stepbystep.model.api		93%		n/a	1	2	2	13	1	2	0	
com.andrey7mel.stepbystep.presenter		100%		81%	5	39	0	79	0	26	0	
com.andrey7mel.stepbystep.presenter.mappers		100%		100%	0	9	0	30	0	6	0	
com.andrey7mel.stepbystep.model		100%		n/a	0	5	0	14	0	5	0	
Total	168 of 864	81%	8 of 38	79%	28	99	47	256	20	80	3	

Created with JaCoCo 0.7.1.20140

Схема взаимодействия компонентов



Все исходники вы можете найти на Github.

Шаг 5. Интеграционное тестирование

Интеграционное тестирование (Integration testing) — одна из фаз тестирования программного обеспечения, при которой отдельные програм
модули объединяются и тестируются в группе.
Выделяется 3 подхода к интеграционному тестированию:

Снизу вверх (Bottom Up Integration)
Все низкоуровневые модули, процедуры или функции собираются воедино и затем тестируются. После чего собирается следующий уровень
модулей для проведения интеграционного тестирования. Данный подход считается полезным, если все или практически все модули,
разрабатываемого уровня, готовы. Также данный подход помогает определить по результатам тестирования уровень готовности приложения

Сверху вниз (Top Down Integration)
Вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые. Все модули более низкого у
симулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются реальными активными компонента
Таким образом мы проводим тестирование сверху вниз.

Большой взрыв («Big Bang» Integration)
Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится
интеграционное тестирование. Такой подход очень хорош для сохранения времени.

Так как у нас все модули уже готовы, будем использовать подход **снизу вверх**.

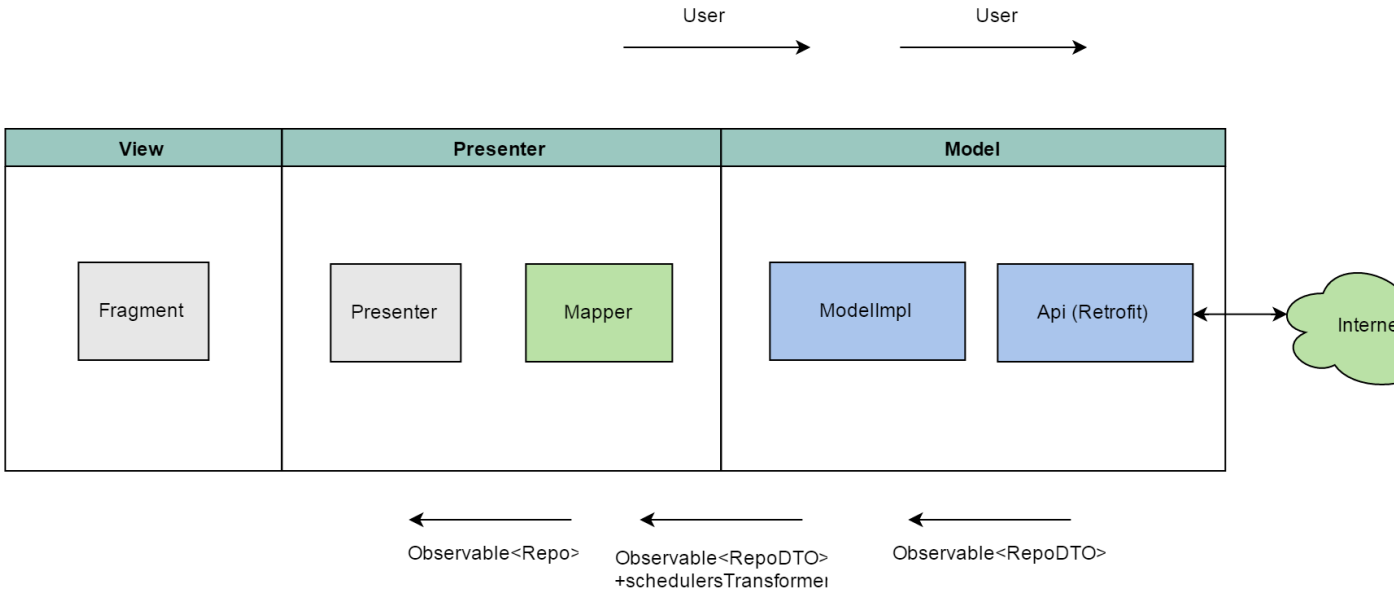
Итеративный подход

Мы будем использовать итеративный подход, т.е будем подключать модули один за одним, снизу вверх. Сначала проверяем связку api + mo
потом api + model + mapper + presenter, затем общую связку api + model mapper + presenter + view

Негативный и позитивный сценарий

Для интеграционных тестов мы должны рассмотреть 2 сценария ответа от сервера: нормальный ответ и ошибка. В зависимости от этого меня
поведение компонентов. Перед каждым тестом мы можем настраивать ответ от сервера (MockWebServer) и проверять результаты.

Схема интеграционного теста (api + model):

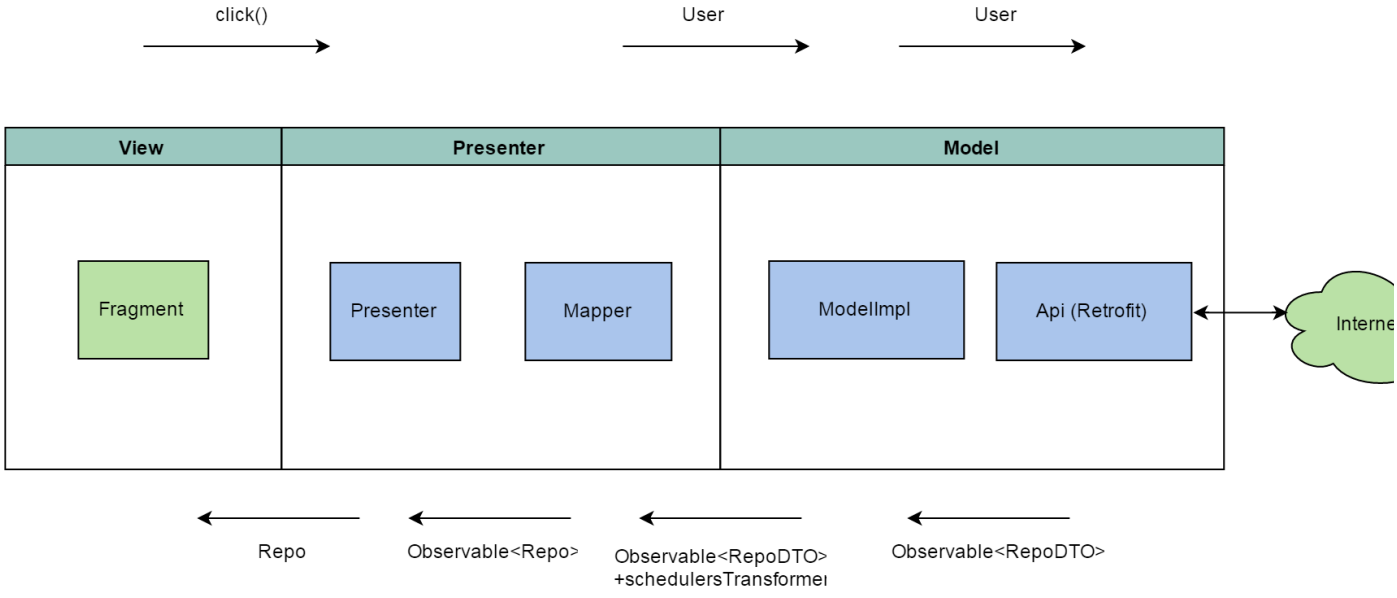


Пример интеграционного теста (api + model), мы проверяем взаимодействие модуля Retrfofit и ModelImpl:

Пример интеграционного теста

```
@Test
public void testGetRepoList() {

    TestSubscriber<List<RepositoryDTO>> testSubscriber = new TestSubscriber<>();
    model.getRepoList(TestConst.TEST_OWNER).subscribe(testSubscriber);

    testSubscriber.assertNoErrors();
    testSubscriber.assertValueCount(1);

    List<RepositoryDTO> actual = testSubscriber.getOnNextEvents().get(0);

    assertEquals(7, actual.size());
    assertEquals("Android-Rate", actual.get(0).getName());
    assertEquals("andrey7mel/Android-Rate", actual.get(0).getFullName());
    assertEquals(26314692, actual.get(0).getId());
}
```

Схема интеграционного теста (api + model + mapper + presenter):



Пример интеграционного теста (api + model + mapper + presenter)

```
@Test
public void testLoadData() {
    repoInfoPresenter.onCreateView(null);
    repoInfoPresenter.onStop();


    verify(mockView).showBranches(branchList);
    verify(mockView).showContributors(contributorList);
}

@Test
public void testLoadDataWithError() {
    setErrorAnswerWebServer();

    repoInfoPresenter.onCreateView(null);
    repoInfoPresenter.onStop();

    verify(mockView, times(2)).showError(TestConst.ERROR_RESPONSE_500);
}
```

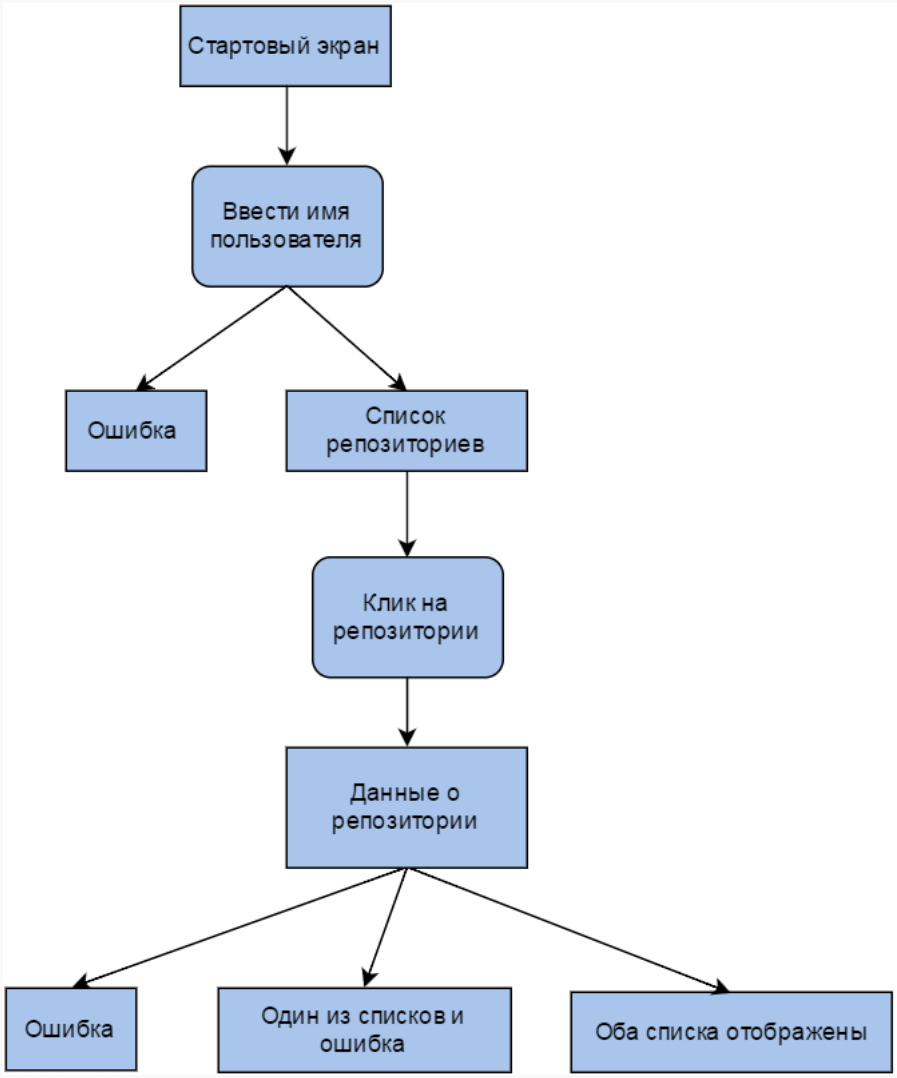
В итоге у нас получится полная проверка взаимодействия всех модулей друг с другом, снизу вверх. Если где то модули будут взаимодейство некорректно, мы быстро увидим это по тестам.

Шаг 6. Функциональное тестирование

Функциональное тестирование — это тестирование ПО в целях проверки реализуемости функциональных требований, то есть способности П определённых условиях решать задачи, нужные пользователям. Функциональные требования определяют, что именно делает ПО, какие зад решает.

В рамках нашего Android приложения мы будем проверять работу приложения с точки зрения пользователя. Для начала составим пользовательскую карту приложения:

▾ Карта приложения



Составим необходимые тест кейсы:

- Открыть приложение, проверить видимость всех элементов
- Ввести тестового пользователя, нажать кнопку Search
  - Данные получены — получить список репозиториев, проверить отображение данных.
  - Данные не получены — проверить отображение ошибки.
- Перейти на второй экран, проверить правильность отображения имени пользователя и названия репозитория.
  - Получить списки бранчей и контрибуторов, проверить отображение данных
  - Какой то из списков не получен (два теста), проверить отображение полученного списка, отображение ошибки
  - Оба списка не получены, проверить отображение ошибки

Для тестирования мы будем использовать Espresso. Также как и для других тестов, изолируем приложение от интернета с помощью моков и заранее подготовленных json файлов. Поможет нам в этом Dagger 2 и подмена компонентов:

▾ Код MockTestRunner и TestApp

```
public class MockTestRunner extends AndroidJUnitRunner {
   @Override
   public Application newApplication(
           ClassLoader cl, String className, Context context)
           throws InstantiationException,
           IllegalAccessException,
           ClassNotFoundException {
       return super.newApplication(
               cl, TestApp.class.getName(), context);
   }
}

public class TestApp extends App {
   @Override
   protected TestComponent buildComponent() {
       return DaggerTestComponent.builder().build();
```

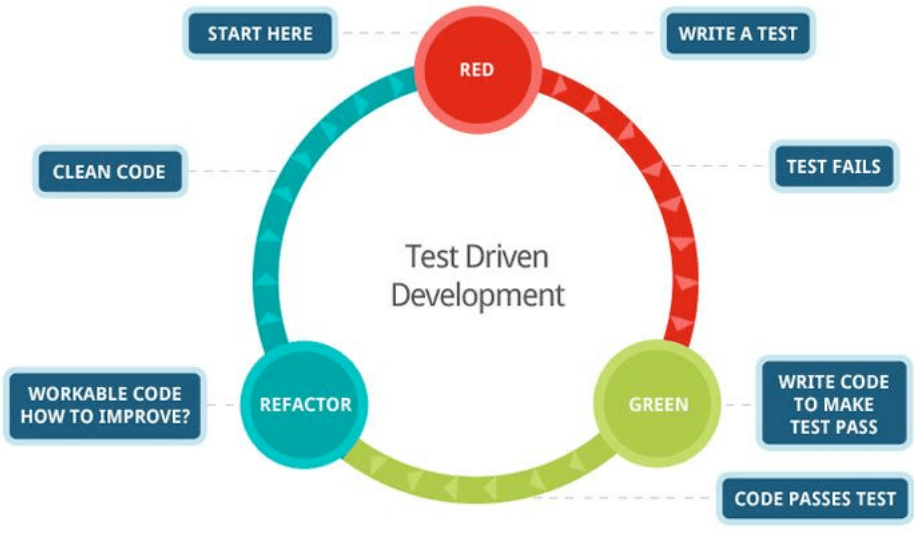
```
    }
}
```

Пример тестов Espresso

```
@Test
public void testGetUserRepo() {
    apiConfig.setCorrectAnswer();
    onView(withId(R.id.edit_text)).perform(clearText());
    onView(withId(R.id.edit_text)).perform(typeText(TestConst.TEST_OWNER));
    onView(withId(R.id.button_search)).perform(click());

    onView(withId(R.id.recycler_view)).check(EspressoTools.hasItemsCount(7));

    onView(withId(R.id.recycler_view)).check(EspressoTools.hasViewWithTextAtPosition(0, "Android-Rate"));
    onView(withId(R.id.recycler_view)).check(EspressoTools.hasViewWithTextAtPosition(1, "android-simple-architecture"));
    onView(withId(R.id.recycler_view)).check(EspressoTools.hasViewWithTextAtPosition(2, TestConst.TEST_REPO));
}

@Test
public void testGetUserRepoError() {
    apiConfig.setErrorAnswer();
    onView(withId(R.id.edit_text)).perform(clearText());
    onView(withId(R.id.edit_text)).perform(typeText(TestConst.TEST_OWNER));
    onView(withId(R.id.button_search)).perform(click());

    onView(allOf(withId(android.support.design.R.id.snackbar_text), withText(TestConst.TEST_ERROR)))
            .check(matches(isDisplayed()));

    onView(withId(R.id.recycler_view)).check(EspressoTools.hasItemsCount(0));
}
```

Аналогично пишем остальные тесты по тест кейсам.
Закончив работу с Espresso, мы полностью покроем приложение модульными, интеграционными и функциональными тестами.

Шаг 7. TDD



Разработка через тестирование (Test-driven development) — техника разработки программного обеспечения, которая определяет разработку написание тестов.

В сущности вам нужно выполнять три простых повторяющихся шага:
— Написать тест для новой функциональности, которую необходимо добавить;
— Написать код, который пройдет тест;
— Провести рефакторинг нового и старого кода.

Если аббревиатура TDD для вас не знакома, рекомендуем почитать статью от наших коллег из iOS отдела или статьи из хаба TDD.

Существуют 3 закона TDD:

- Не пишется production код, прежде чем для него есть неработающий тест;
- Не пишется больше кода юнит теста, чем достаточно для его ошибки.
- Не пишется больше production кода, чем достаточно для прохождения текущего неработающего теста.

Для примера создадим progress bar который будет показывать загрузку из интернета. Он должен появляться когда происходит загрузка дан
исчезать когда данные загружены или появилась ошибка. Всю разработку будем вести по TDD.

Разработка данной функциональности затронет презентеры и фрагменты, мапперы и дата слой остаются без изменений.

Презентеры

Начнем со списка репозиториев. Первым делом дополним интерфейсы:

```
public interface RepoListView extends View {

   void showRepoList(List<Repository> list);

   void showEmptyList();

   String getUserName();

   void startRepoInfoFragment(Repository repository);

  //New

  void showLoading();

  void hideLoading();
}
```

Первый этап.

Сначала пишем тест, который проверит, что в случае нормальной загрузки был вызван метод showLoading у фрагмента:

```
@Test
public void testShowLoading() {
   repoListPresenter.onSearchButtonClick();
   verify(mockView).showLoading();
}
```

Как только получили неработающий тест, пишем код, который пройдет его:

```
public void onSearchButtonClick() {
   String name = view.getUserName();
   if (TextUtils.isEmpty(name)) return;
   view.showLoading();
   // --- some code ---
}
```

Рефакторить пока нечего.

На этом первая итерация разработки по TDD закончилась. Мы получили новую функциональность и тест для нее.

Второй этап.

Напишем тест, который проверит, что после нормальной загрузки был вызван метод hideLoading у фрагмента:

```
@Test
public void testHideLoading() {
   repoListPresenter.onSearchButtonClick();
   verify(mockView).hideLoading();
}
```

Пишем код, который пройдет тест:

```
//--
view.showLoading();
Subscription subscription = model.getRepoList(name)
       .map(repoListMapper)
       .subscribe(new Observer<List<Repository>>() {
           @Override
           public void onCompleted() {
              view.hideLoading();
           }

           @Override
           public void onError(Throwable e) {
               view.showError(e.getMessage());
           }

           @Override
           public void onNext(List<Repository> list) {
               if (list != null && !list.isEmpty()) {
                   repoList = list;
                   view.showRepoList(list);
               } else {
                   view.showEmptyList();
               }
           }
       });
```

Рефакторинг не требуется.

Третий и четвертый этапы.

Теперь напишем тесты, которые проверят, что при возникновении ошибки, были корректны вызваны необходимые методы.

▾ Тесты на обработку ошибки

```
@Test
public void testShowLoadingOnError() {
   doAnswer(invocation -> Observable.error(new Throwable(TestConst.TEST_ERROR)))
           .when(model)
           .getRepoList(TestConst.TEST_OWNER);
   repoListPresenter.onSearchButtonClick();
   verify(mockView).showLoading();
}

@Test
public void testHideLoadingOnError() {
   doAnswer(invocation -> Observable.error(new Throwable(TestConst.TEST_ERROR)))
           .when(model)
           .getRepoList(TestConst.TEST_OWNER);
   repoListPresenter.onSearchButtonClick();
   verify(mockView).hideLoading();
}
```

▾ Код обработки ошибки

```
//--
  @Override
           public void onError(Throwable e) {
                  view.showError(e.getMessage());
                      view.hideLoading();
           }
//--
```

Рефакторинг не требуется. Работа с Repo List Presenter завершена, теперь перейдем к Repo Info Presenter.

Repo Info Presenter

Аналогично предыдущему шагу, пишем тесты и код для корректной загрузки данных.

▾ Тесты для корректной загрузки данных

```
@Test
public void testShowLoading() {
   repoInfoPresenter.onCreateView(null);
   verify(mockView).showLoading();
}

@Test
public void testHideLoading() {
   repoInfoPresenter.onCreateView(null);
   verify(mockView).hideLoading();
}
```

▾ Код для корректной загрузки данных

```
public void loadData() {
   String owner = repository.getOwnerName();
   String name = repository.getRepoName();

   view.showLoading();
   Subscription subscriptionBranches = model.getRepoBranches(owner, name)
           .map(branchesMapper)
           .subscribe(new Observer<List<Branch>>() {
               @Override
               public void onCompleted() {
                   hideInfoLoadingState();
               }

               @Override
               public void onError(Throwable e) {
                   view.showError(e.getMessage());
               }

               @Override
               public void onNext(List<Branch> list) {
                   branchList = list;
                   view.showBranches(list);
               }
           });
   addSubscription(subscriptionBranches);

   Subscription subscriptionContributors = model.getRepoContributors(owner, name)
           .map(contributorsMapper)
           .subscribe(new Observer<List<Contributor>>() {
               @Override
               public void onCompleted() {
                   hideInfoLoadingState();
               }

               @Override
               public void onError(Throwable e) {
                   view.showError(e.getMessage());
               }

               @Override
               public void onNext(List<Contributor> list) {
                   contributorList = list;
                   view.showContributors(list);
               }
           });
```

```
    addSubscription(subscriptionContributors);
}

protected void hideInfoLoadingState() {
    countCompletedSubscription++;

    if (countCompletedSubscription == COUNT_SUBSCRIPTION) {
        view.hideLoading();
        countCompletedSubscription = 0;
    }
}
```

Рефакторинг.
Как видно, используется одинаковый код для двух презентеров (показать и скрыть индикатор загрузки, показать ошибку). Необходимо выне его в общий базовый класс BasePresenter. Выносим методы showLoadingState() hideLoadingState() и showError(Throwable e) в BasePresenter

▾ Код BasePresenter

```
protected abstract View getView();

protected void showLoadingState() {
    getView().showLoadingState();
}

protected void hideLoadingState() {
    getView().hideLoadingState();
}

protected void showError(Throwable e) {
    getView().showError(e.getMessage());
}
```

Рефакторим RepoInfoPresenter и проверим, что проходят все тесты. Не забываем сделать рефакторинг RepoListPresenter для работы с базовы классом.

Далее пишем сначала тесты, а потом код для обработки ошибок во время загрузки (для RepoInfoPresenter).

▾ Тесты для обработки ошибок во время загрузки

```
@Test
public void testShowLoadingOnError() {
    doAnswer(invocation -> Observable.error(new Throwable(TestConst.TEST_ERROR)))
            .when(model)
            .getRepoContributors(TestConst.TEST_OWNER, TestConst.TEST_REPO);
    doAnswer(invocation -> Observable.error(new Throwable(TestConst.TEST_ERROR)))
            .when(model)
            .getRepoBranches(TestConst.TEST_OWNER, TestConst.TEST_REPO);
    repoInfoPresenter.onCreateView(null);
    verify(mockView).showLoading();
}

@Test
public void testHideLoadingOnError() {
    doAnswer(invocation -> Observable.error(new Throwable(TestConst.TEST_ERROR)))
            .when(model)
            .getRepoContributors(TestConst.TEST_OWNER, TestConst.TEST_REPO);
    doAnswer(invocation -> Observable.error(new Throwable(TestConst.TEST_ERROR)))
            .when(model)
            .getRepoBranches(TestConst.TEST_OWNER, TestConst.TEST_REPO);
    repoInfoPresenter.onCreateView(null);
    verify(mockView).hideLoading();
}

@Test
public void testShowLoadingOnErrorBranches() {
```

```
    doAnswer(invocation -> Observable.error(new Throwable(TestConst.TEST_ERROR)))
            .when(model)
            .getRepoBranches(TestConst.TEST_OWNER, TestConst.TEST_REPO);
    repoInfoPresenter.onCreateView(null);
    verify(mockView).showLoading();
}

@Test
public void testHideLoadingOnErrorBranches() {
    doAnswer(invocation -> Observable.error(new Throwable(TestConst.TEST_ERROR)))
            .when(model)
            .getRepoBranches(TestConst.TEST_OWNER, TestConst.TEST_REPO);
    repoInfoPresenter.onCreateView(null);
    verify(mockView).hideLoading();
}
```

▾ Код для обработки ошибок во время загрузки

```
showLoadingState();
Subscription subscriptionBranches = model.getRepoBranches(owner, name)
        .map(branchesMapper)
        .subscribe(new Observer<List<Branch>>() {
            @Override
            public void onCompleted() {
                hideInfoLoadingState();
            }

            @Override
            public void onError(Throwable e) {
                hideInfoLoadingState();
                showError(e);
            }

            @Override
            public void onNext(List<Branch> list) {
                branchList = list;
                view.showBranches(list);
            }
        });
addSubscription(subscriptionBranches);

Subscription subscriptionContributors = model.getRepoContributors(owner, name)
        .map(contributorsMapper)
        .subscribe(new Observer<List<Contributor>>() {
            @Override
            public void onCompleted() {
                hideInfoLoadingState();
            }

            @Override
            public void onError(Throwable e) {
                hideInfoLoadingState();
                showError(e);
            }

            @Override
            public void onNext(List<Contributor> list) {
                contributorList = list;
                view.showContributors(list);
            }
        });
```

На этом разработка презентеров закончена. Переходим к фрагментам.

Фрагменты

Progress bar, как общий элемент, будет лежать в activity, фрагменты будут вызывать у activity методы showProgressBar() и hideProgressBar(), которые покажут или спрячут progress bar. Для работы с activity используем интерфейс ActivityCallback. По опыту презентеров, можем сразу догадаться, что нам будет необходим общий базовый класс — BaseFragment. В нем будет содержаться логика взаимодействия с activity.

Сначала пишем тесты, а потом код, для взаимодействия базового фрагмента с activity:

Тесты Base Fragment

```
@Test
public void testAttachActivityCallback() throws Exception {
     assertNotNull(baseFragment.activityCallback);
}

@Test
public void testShowLoadingState() throws Exception {
   baseFragment.showLoading();
   verify(activity).showProgressBar();
}

@Test
public void testHideLoadingState() throws Exception {
   baseFragment.hideLoading();
   verify(activity).hideProgressBar();
}
```

Код Base Fragment

```
@Override
public void onAttach(Activity activity) {
   super.onAttach(activity);

   try {
       activityCallback = (ActivityCallback) activity;
   } catch (ClassCastException e) {
       throw new ClassCastException(activity.toString()
               + " must implement activityCallback");
   }
}
@Override
public void showLoading() {
   activityCallback.showProgressBar();
}

@Override
public void hideLoading() {
   activityCallback.hideProgressBar();
}
```

Рефакторинг не требуется, переходим к activity.

Acitivity

Последним шагом реализуем интерфейс Activity. Мы будем изменять видимость (setVisibility) progressBar в зависимости от команды. В тестах необходимо проверить что progressBar найден и работу методов showProgressBar и hideProgressBar.

Сначала пишем тесты:

Тесты Activity

```
@Test
public void testHaveProgressBar() throws Exception {
    assertNotNull(progressBar);
}
```

```
@Test
public void testShowProgressBar() throws Exception {
   mainActivity.showProgressBar();
   verify(progressBar).setVisibility(View.VISIBLE);
}

@Test
public void testHideProgressBar() throws Exception {
   mainActivity.hideProgressBar();
   verify(progressBar).setVisibility(View.INVISIBLE);
}
```

Потом пишем код:

▾ Код Activity

```
@Bind(R.id.toolbar_progress_bar)
protected ProgressBar progressBar;

//---- some code ----

@Override
public void showProgressBar() {
   progressBar.setVisibility(View.VISIBLE);
}

@Override
public void hideProgressBar() {
   progressBar.setVisibility(View.INVISIBLE);
}
```

Все достаточно тривиально, рефакторинг не требуется.

На этом мы закончим разработку progress bar с использованием техники TDD.

Шаг 8. Что дальше?

Изучив TDD и разработав отображение загрузки, мы закончим разработку приложения. Для дальнейшего развития рекомендую к прочтению следующие статьи:

Android Clean Architecture
Android Clean Architecture — известная статья от Fernando Cejas, на основе Clean Architecture от Дядюшки Боба. Рассматривается взаимодейс между 3 слоями Presentation Layer, Domain Layer и Data Layer. Есть перевод на habrahabr.

VIPER
VIPER (View, Interactor, Presenter, Entity и Routing) становится все более популярен, познакомится с ним можно в статье Android VIPER на реактивной тяге от @VikkoS. Основные принципы VIPER освещены в статьях и докладах наших коллег из отдела iOS.

Mosby
Mosby — популярная библиотека для создания MVP приложений. Содержит в себе все основные интерфейсы и базовые классы. Сайт: http://hannesdorfmann.com/mosby/ Github: https://github.com/sockeqwe/mosby

Android Application Architecture
Хорошая статья про архитектуру от Ribot team — Android Application Architecture. Рассматривается миграция с AsyncTask на RxJava. Совсем н вышел перевод на habrahabr.

Android Development Culture Document
Android Development Culture Document #qualitymatters от @Artem_zin. Отличная статья и демонстрационный проект от Artem Zinnatullin. В ст рассматриваются 8 принципов разработки android приложений, подкрепляется все это примером на Github.

Заключение

В этом цикле статей мы прошли все этапы разработки приложения. Начали мы с простой архитектуры на основе MVP, усложняя ее по ходу добавления новых фич. Использовали современные библиотеки: RxJava и RxAndroid для реактивного программирования и избавления от cal ов, Retrofit для удобной работы с сетью, Butterknife для быстрого и легкого поиска view. Dagger 2 управлял всеми зависимостями и оказал на

неоценимую поддержку при написании тестов. Сами тесты мы писали с помощью jUnit, Robolectric, Mockito и MockWebServer. А Espresso изба
наших тестировщиков от мук регрессионного тестирования.

Мы полностью покрыли наш проект тестами. Unit тесты изолированно проверяют каждый компонент, интеграционные тесты проверяют их об
взаимодействие, а функциональные тесты смотрят на все это со стороны пользователя. При дальнейшем изменении программы мы можем не
бояться (ну или почти не бояться), что поломаем какие то компоненты, и что то отвалится, а баги пролезут в релиз. Благодаря TDD, большая
нашего кода будет покрыта тестами (нет теста, нет и кода). Не будет проблемы частичного покрытия или "код написали, а на тесты времени
осталось".

Весь код проекта доступен на Github (https://github.com/andrey7mel/android-step-by-step)
Надеюсь, эта серия статей вам понравилась и оказалась полезной, спасибо за внимание!

android, rxjava, архитектура приложений, mvp, архитектура android-приложений, TDD

Реализация псевдо-3D в гоночных играх

+90 13,1k 172 16

Индексы в PostgreSQL — 1

+102 11,7k 306 32

Выбор MQ для высоконагруженного проекта

+30 10k 133 49

Алгоритм Джонкера-Волгенанта + t-SNE = супер-сила

+63 9,6k 148 2

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

PhotoScan: как делать фотографии фотографий без бликов GT

1,6k 14 5

Электроника для ролевых игр в 2016 году GT

3k 6 10

Тысячная избранная статья. Как устроено рецензирование в Википедии GT

3,9k 17 44

SmartPoket. Часть 3, заключительная: мой личный органайзер GT

3,6k 7 12

Разработчик «умных» кредиток Plastc собрал предзаказов на $9 млн и объявил о банкротстве GT

9,4k 11 22