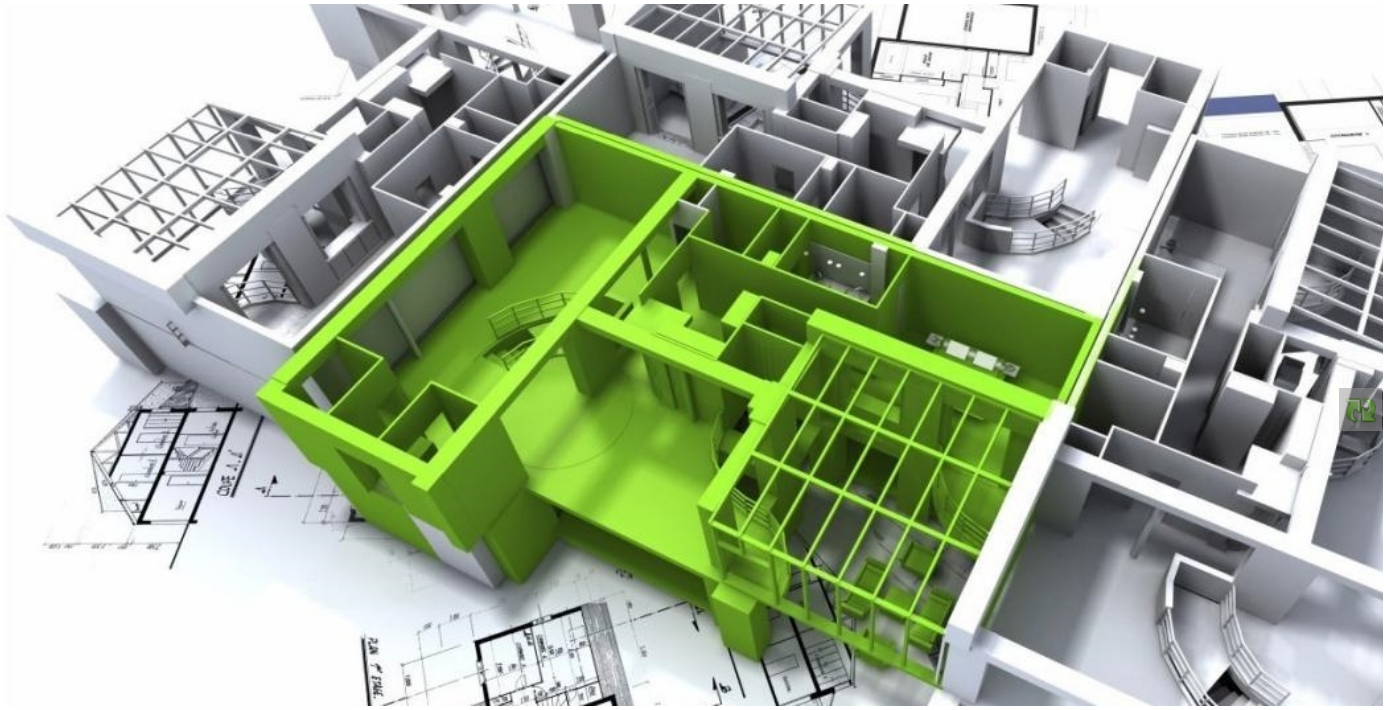
Rambler&Co 85,47
Компания
Подпи

18 февраля 2016 в 15:52

Построение Android приложений шаг за шагом, часть вторая

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



В первой части статьи мы разработали приложение для работы с github, состоящее из двух экранов, разделенное по слоям с применением па MVP. Мы использовали RxJava для упрощения взаимодействия с сервером и две модели данных для разных слоев. Во второй части мы внедр Dagger 2, напишем unit тесты, посмотрим на MockWebServer, JaCoCo и Robolectric.

Содержание:

Полное содержание

Введение

В первой части статьи мы в два этапа создали простое приложение для работы с github.

Условная схема приложения

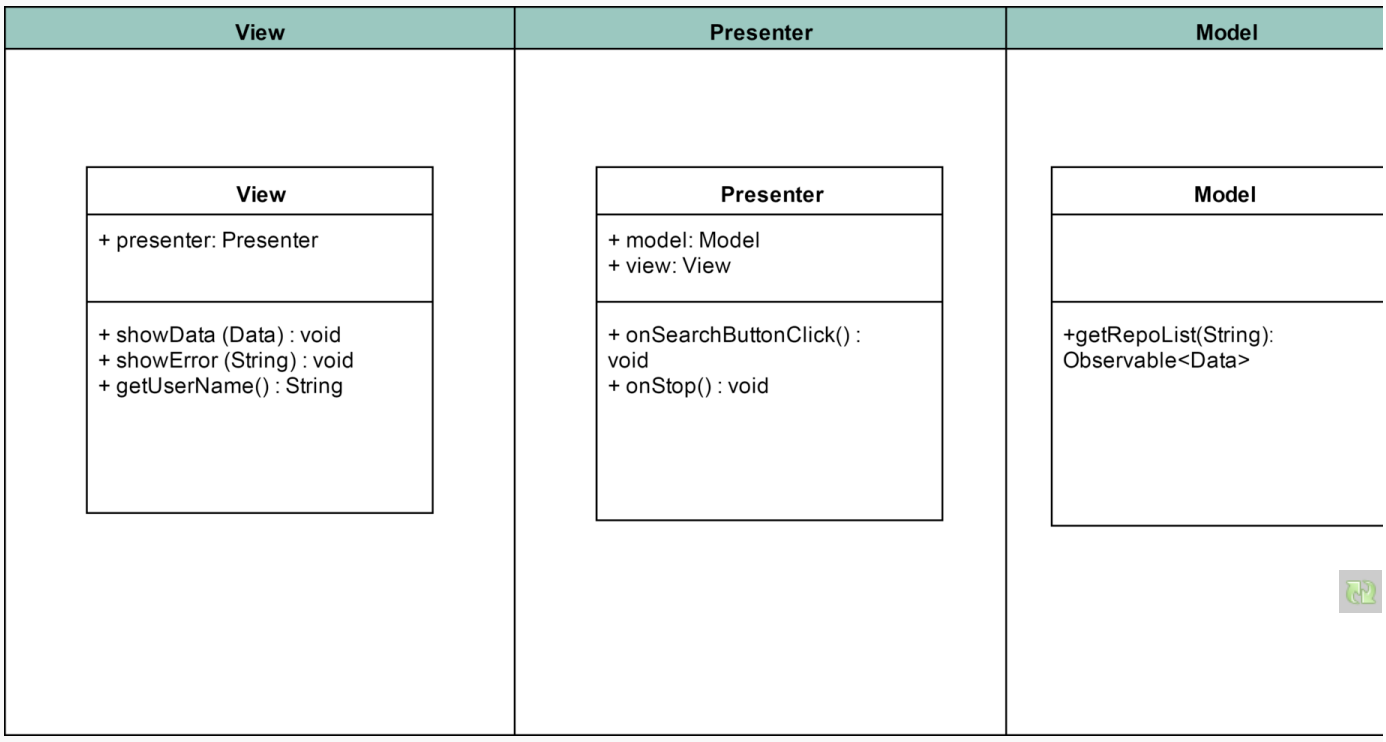
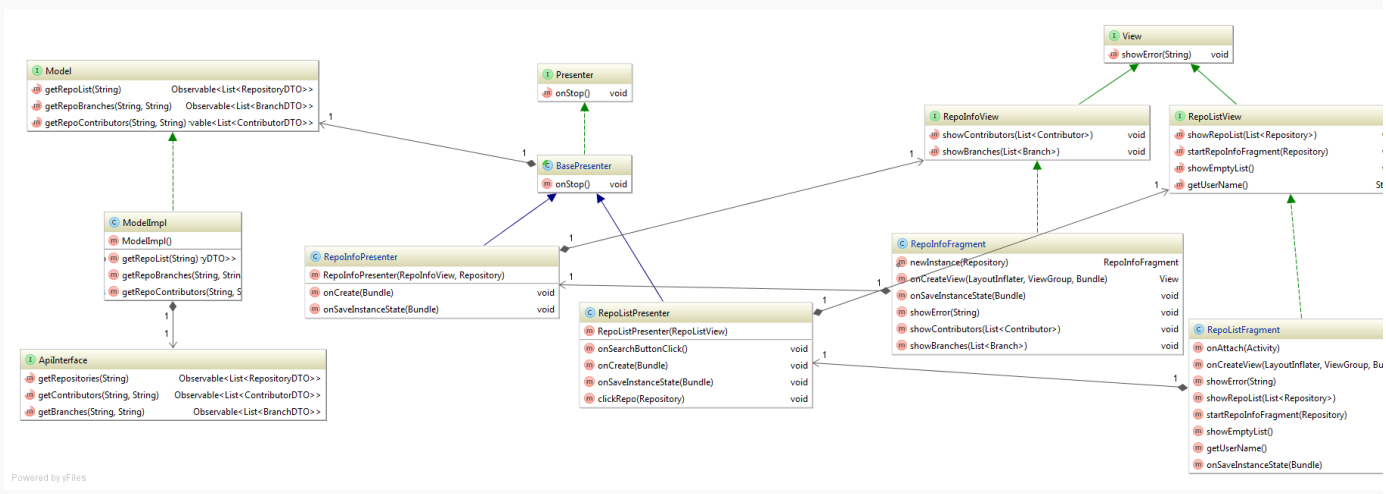


Диаграмма классов



Все исходники вы можете найти на Github. Ветки в репозитории соответствуют шагам в статье: Step 3 Dependency injection — третий шаг, St
Unit tests — четвертый шаг.

Шаг 3. Dependency Injection

Перед тем, как использовать Dagger 2, необходимо понять принцип Dependency injection (Внедрение зависимости).

Представим, что то у нас есть объект A, который включает объект B. Без использования DI мы должны создавать объект B в коде класса A.

Например так:

```
public class A {
   B b;

   public A() {
       b = new B();
   }
}
```

Такой код сразу же нарушает SRP и DRP из принципов SOLID. Самым простым решением является передача объекта B в конструктор класса самым мы реализуем Dependency Injection “вручную”:

```
public class A {
   B b;

   public A(B b) {
       this.b = b;
   }
}
```

Обычно DI реализуется с помощью сторонних библиотек, где благодаря аннотациям, происходит автоматическая подстановка объекта.

```
public class A {
   @Inject
   B b;

   public A() {
       inject();
   }
}
```

Подробнее об этом механизме и его применении на Android можно прочитать в этой статье: Знакомимся с Dependency Injection на примере D

Dagger 2

Dagger 2 — библиотека созданная Google для реализации DI. Ее основное преимущество в кодогенерации, т.е. все ошибки будут видны на э компиляции. На хабре есть хорошая статья про Dagger 2, также можно почитать официальную страницу или хорошую инструкцию на codepa

Для установки Dagger 2 необходимо отредактировать build.gradle:

▾ build.gradle

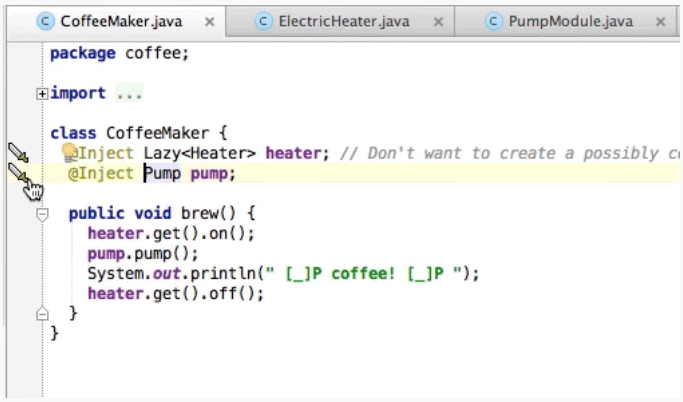
```
apply plugin: 'com.android.application'
apply plugin: 'com.neenbedankt.android-apt'

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:21.0.3'

    compile 'com.google.dagger:dagger:2.0-SNAPSHOT'
    apt 'com.google.dagger:dagger-compiler:2.0-SNAPSHOT'
    provided 'org.glassfish:javax.annotation:10.0-b28'
}
```

Также очень рекомендуется поставить плагин Dagger IntelliJ Plugin. Он поможет ориентироваться откуда и куда происходят инжекции.

▾ Dagger IntelliJ Plugin



Сами объекты для внедрения Dagger 2 берет из методов модулей (методы должны помечаться аннотацией @Provides, модули — @Module) ил создает их с помощью конструктора класса аннотированного @Inject. Например:

```
@Module
public class ModelModule {

   @Provides
   @Singleton
   ApiInterface provideApiInterface() {
       return ApiModule.getApiInterface();
   }
}
```

или

```
public class RepoBranchesMapper

   @Inject
   public RepoBranchesMapper() {}
}
```

Поля для внедрения обозначаются аннотацией @Inject:

```
@Inject
protected ApiInterface apiInterface;
```

Связываются эти две вещи с помощью компонентов (@Component). В них указывается откуда брать объекты и куда их внедрять (методы inje Пример:

```
@Singleton
@Component(modules = {ModelModule.class})
public interface AppComponent {

   void inject(ModelImpl dataRepository);
}
```

Для работы Dagger 2 мы будем использовать один компонент (AppComponent) и 3 модуля для разных слоев (Model, Presentation, View).

▾ AppComponent

```
@Singleton
@Component(modules = {ModelModule.class, PresenterModule.class, ViewModule.class})
public interface AppComponent {

   void inject(ModelImpl dataRepository);

   void inject(BasePresenter basePresenter);

   void inject(RepoListPresenter repoListPresenter);

   void inject(RepoInfoPresenter repoInfoPresenter);
```

```
    void inject(RepoInfoFragment repoInfoFragment);
}
```

Model

Для Model — слоя необходимо необходимо предоставлять ApiInterface и два Scheduler для управления потоками. Для Scheduler необходимо использовать аннотацию @Named, чтобы Dagger разобрался с графом зависимостей.

▾ ModelModule

```
@Provides
@Singleton
ApiInterface provideApiInterface() {
    return ApiModule.getApiInterface(Const.BASE_URL);
}

@Provides
@Singleton
@Named(Const.UI_THREAD)
Scheduler provideSchedulerUI() {
    return AndroidSchedulers.mainThread();
}

@Provides
@Singleton
@Named(Const.IO_THREAD)
Scheduler provideSchedulerIO() {
    return Schedulers.io();
}
```

Presenter

Для presenter слоя нам необходимо предоставлять Model и CompositeSubscription, а также мапперы. Model и CompositeSubscription будем предоставлять через модули, мапперы — с помощью аннотированного конструктора.

▾ Presenter Module

```
public class PresenterModule {

    @Provides
    @Singleton
    Model provideDataRepository() {
        return new ModelImpl();
    }

    @Provides
    CompositeSubscription provideCompositeSubscription() {
        return new CompositeSubscription();
    }
}
```

▾ Пример маппера с аннотированным конструктором

```
public class RepoBranchesMapper implements Func1<List<BranchDTO>, List<Branch>> {

    @Inject
    public RepoBranchesMapper() {
    }

    @Override
    public List<Branch> call(List<BranchDTO> branchDTOs) {
        List<Branch> branches = Observable.from(branchDTOs)
```

```
                .map(branchDTO -> new Branch(branchDTO.getName()))
                .toList()
                .toBlocking()
                .first();
        return branches;
    }
}
```

View

Со View слоем и внедрением презентеров ситуация сложнее. При создании презентера мы в конструкторе передаем интерфейс View. Соответственно, Dagger должен иметь ссылку на реализацию этого интерфейса, т.е на наш фрагмент. Можно пойти и другим путем, изменив интерфейс презентера и передавая ссылку на view в onCreate. Рассмотрим оба случая.

Передача ссылки на view.

У нас есть фрагмент RepoListFragment, реализующий интерфейс RepoListView,
и RepoListPresenter, принимающий на вход в конструкторе этот RepoListView. Нам необходимо внедрить RepoListPresenter в RepoListFragment. реализации такой схемы нам придется создать новый компонент и новый модуль, который в конструкторе будет принимать ссылку на наш интерфейс RepoListView. В этом модуле мы будем создавать презентер (с использованием ссылки на интрефейс RepoListView) и внедрять его фрагмент.

▾ Внедрение во фрагменте

```
@Override
public void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    DaggerViewComponent.builder()
            .viewDynamicModule(new ViewDynamicModule(this))
            .build()
            .inject(this);
}
```

▾ Компонент

```
@Singleton
@Component(modules = {ViewDynamicModule.class})
public interface ViewComponent {

    void inject(RepoListFragment repoListFragment);
}
```

▾ Модуль

```
@Module
public class ViewDynamicModule {

    RepoListView view;

    public ViewDynamicModule(RepoListView view) {
        this.view = view;
    }

    @Provides
    RepoListPresenter provideRepoListPresenter() {
        return new RepoListPresenter(view);
    }
}
```

В реальных приложениях у вас будет множество инжекций и модулей, поэтому создание различных компонентов для различных сущностей - отличная идея для предотвращения создания god object.

Изменение кода презентера.

Приведенный выше метод требует создания нескольких файлов и множества действий. В нашем случае, есть гораздо более простой способ, изменим конструктор и будем передавать ссылку на интерфейс в onCreate.
Код:

▾ Внедрение во фрагменте

```
@Inject
RepoInfoPresenter presenter;

@Override
public void onCreate(@Nullable Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   App.getComponent().inject(this);
   presenter.onCreate(this, getRepositoryVO());
}
```

▾ Модуль

```
@Module
public class ViewModule {

   @Provides
   RepoInfoPresenter provideRepoInfoPresenter() {
       return new RepoInfoPresenter();
   }
}
```

Завершив внедрение Dagger 2, перейдем к тестированию приложения.

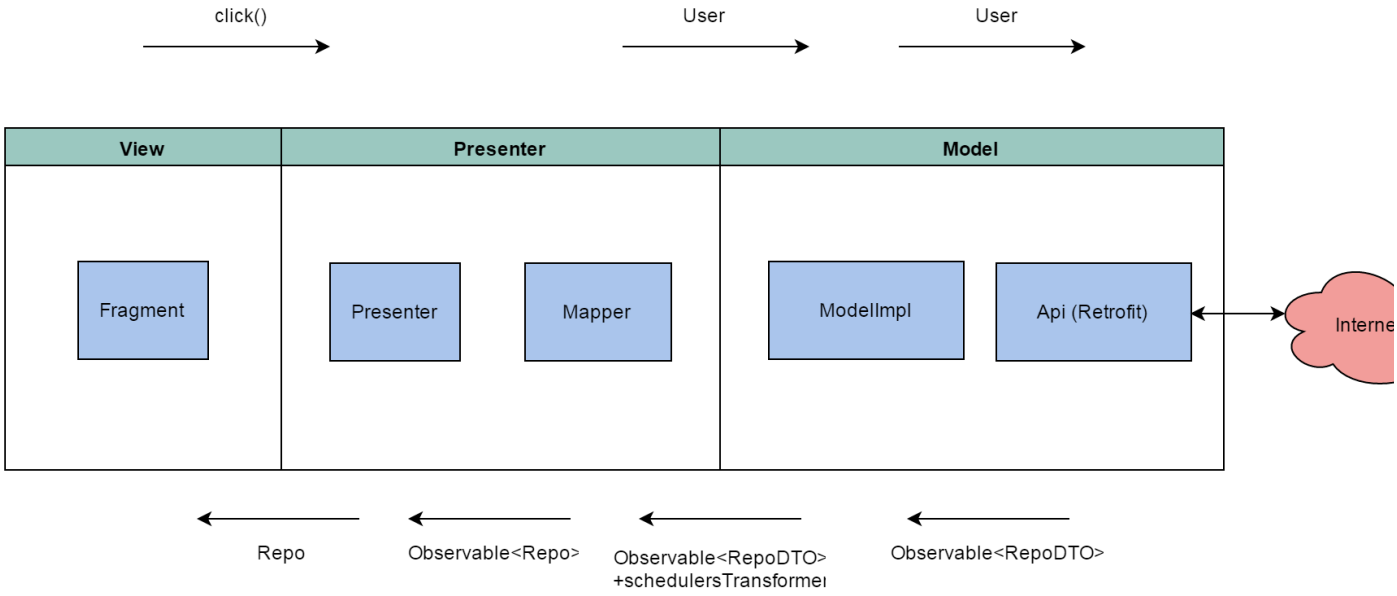
Шаг 4. Модульное тестирование

Тестирование давно стало неотъемлемой частью процесса разработки ПО.
Википедия выделяет множество видов тестирования, в первую очередь разберемся с модульным (unit) тестированием.

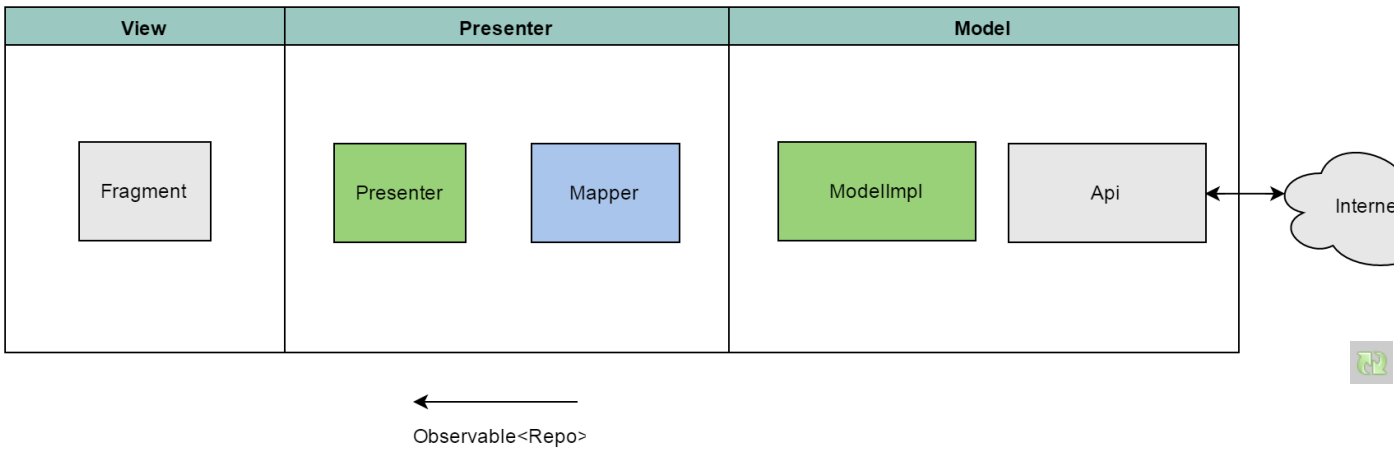
Модульное тестирование процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода програм
Идея состоит в том, чтобы писать тесты для каждого нетривиального метода. Это позволяет достаточно быстро проверить, не привело ли оче
изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устра
таких ошибок.

Написать полностью изолированные тесты у нас не получится, потому что все компоненты взаимодействуют друг с другом. Под unit тестами, будем понимать проверку работы одного модуля окруженного моками. Взаимодействие нескольких реальных модулей будем проверять в интеграционных тестах.

Схема взаимодействия модулей:



Пример тестирования маппера (серые модули — не используются, зеленые — моки, синий — тестируемый модуль):



Инфраструктура

Инструменты и фреймворки повышают удобство написания и поддержки тестов. CI сервер, который не даст вам сделать merge при красных
резко уменьшает шансы неожиданной поломки тестов в master branch. Автоматический запуск тестов и ночные сборки помогают выявить пр
на самом раннем этапе. Этот принцип получил название fail fast.
Про тестовое окружение вы можете почитать в статье Тестирование на Android: Robolectric + Jenkins + JaCoCo. В дальнейшем мы будем
использовать Robolecric для написания тестов, mockito для создания моков и JaCoCo для проверки покрытия кода тестами.

Паттерн MVP позволяет быстро и эффективно писать тесты на наш код. С помощью Dagger 2 мы сможем подменить настоящие объекты на те
моки, изолировав код от внешнего мира. Для этого используем тестовый компонент с тестовыми модулями. Подмена компонента происходит
тестовом application, который мы задаем с помощью аннотации @Config(application = TestApplication.class) в базовом тестовом классе.

JaCoCo Code Coverage

Перед началом работы, нужно определить какие методы тестировать и как считать процент покрытия тестами. Для этого используем библио
JaCoCo, которая генерирует отчеты по результатам выполнения тестов.
Современная Android Studio поддерживает code coverage из коробки или можно настроить его, добавив в build.gradle следующие строки:

build.gradle

```
apply plugin: 'jacoco'

jacoco {
   toolVersion = "0.7.1.201405082137"
}

def coverageSourceDirs = [
       '../app/src/main/java'
]
```

```
task jacocoTestReport(type: JacocoReport, dependsOn: "testDebugUnitTest") {
    group = "Reporting"

    description = "Generate Jacoco coverage reports"

    classDirectories = fileTree(
            dir: '../app/build/intermediates/classes/debug',
            excludes: ['**/R.class',
                       '**/R$*.class',
                       '**/*$ViewInjector*.*',
                       '**/*$ViewBinder*.*',   //DI
                       '**/*_MembersInjector*.*',  //DI
                       '**/*_Factory*.*',  //DI
                       '**/testrx/model/dto/*.*', //dto model
                       '**/testrx/presenter/vo/*.*', //vo model
                       '**/testrx/other/**',
                       '**/BuildConfig.*',
                       '**/Manifest*.*',
                       '**/Lambda$*.class',
                       '**/Lambda.class',
                       '**/*Lambda.class',
                       '**/*Lambda*.class']
    )

    additionalSourceDirs = files(coverageSourceDirs)
    sourceDirectories = files(coverageSourceDirs)
    executionData = files('../app/build/jacoco/testDebugUnitTest.exec')

    reports {
        xml.enabled = true
        html.enabled = true
    }
}
```

Обратите внимание на исключенные классы: мы удалили все что связано с Dagger 2 и нашими моделями DTO и VO.

Запустим jacoco (gradlew jacocoTestReport) и посмотрим на результаты:

app

app

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

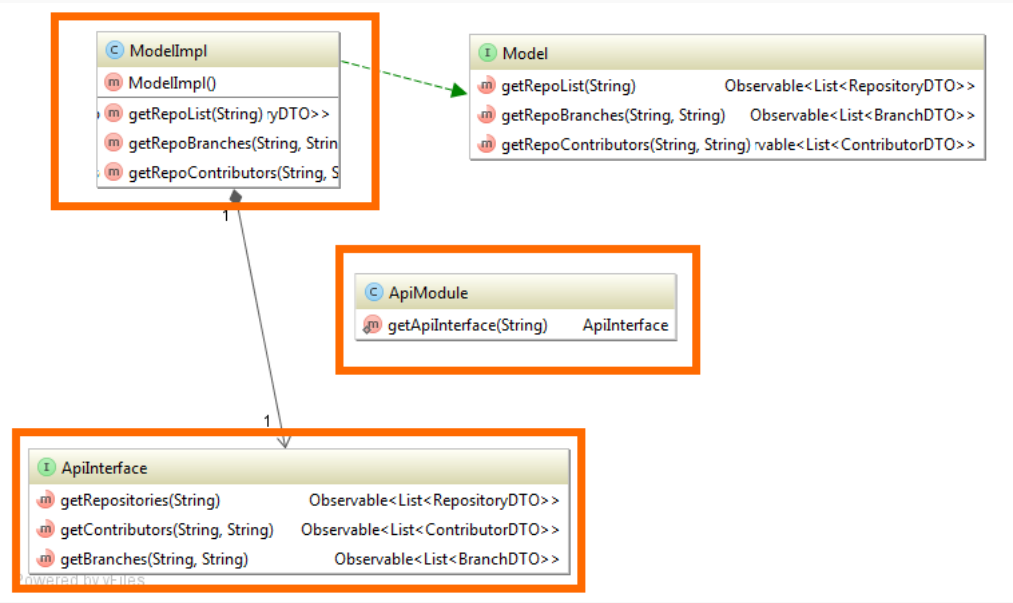
Created with JaCoCo 0.7.1.201

Сейчас у нас процент покрытия идеально совпадает с нашим количеством тестов, т.е 0% =) Давайте исправим эту ситуацию!

Model

В model слое нам необходимо проверить правильность настройки retrofit (ApiInterface), корректность создания клиента и работу ModelImpl. Компоненты должны проверяться изолированно, поэтому для проверки нам нужно эмулировать сервер, в этом нам поможет MockWebServer. Настраиваем ответы сервера и проверяем запросы retrofit.

▾ Схема Model слоя, классы требующие тестирования помечены красным



Тестовый модуль для Dagger 2

```
@Module
public class ModelTestModule {

    @Provides
    @Singleton
    ApiInterface provideApiInterface() {
        return mock(ApiInterface.class);
    }

    @Provides
    @Singleton
    @Named(Const.UI_THREAD)
    Scheduler provideSchedulerUI() {
        return Schedulers.immediate();
    }

    @Provides
    @Singleton
    @Named(Const.IO_THREAD)
    Scheduler provideSchedulerIO() {
        return Schedulers.immediate();
    }
}
```

Примеры тестов

```
public class ApiInterfaceTest extends BaseTest {

    private MockWebServer server;
    private ApiInterface apiInterface;

    @Before
    public void setUp() throws Exception {
        super.setUp();
        server = new MockWebServer();
        server.start();
        final Dispatcher dispatcher = new Dispatcher() {

            @Override
            public MockResponse dispatch(RecordedRequest request) throws InterruptedException {

                if (request.getPath().equals("/users/" + TestConst.TEST_OWNER + "/repos")) {
                    return new MockResponse().setResponseCode(200)
```

```
                        .setBody(testUtils.readString("json/repos"));
                } else if (request.getPath().equals("/repos/" + TestConst.TEST_OWNER + "/" + TestConst.TEST_REPO + "/branches")) {
                    return new MockResponse().setResponseCode(200)
                            .setBody(testUtils.readString("json/branches"));
                } else if (request.getPath().equals("/repos/" + TestConst.TEST_OWNER + "/" + TestConst.TEST_REPO + "/contributors")) {
                    return new MockResponse().setResponseCode(200)
                            .setBody(testUtils.readString("json/contributors"));
                }
                return new MockResponse().setResponseCode(404);
            }
        };

        server.setDispatcher(dispatcher);
        HttpUrl baseUrl = server.url("/");
        apiInterface = ApiModule.getApiInterface(baseUrl.toString());
    }


    @Test
    public void testGetRepositories() throws Exception {

        TestSubscriber<List<RepositoryDTO>> testSubscriber = new TestSubscriber<>();
        apiInterface.getRepositories(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());
    }

   @After
     public void tearDown() throws Exception {
         server.shutdown();
     }
}
```

Для проверки модели мокаем ApiInterface и проверяем корректность работы.

▾ Пример тестов для ModelImpl

```
@Test
public void testGetRepoBranches() {

    BranchDTO[] branchDTOs = testUtils.getGson().fromJson(testUtils.readString("json/branches"), BranchDTO[].class);

    when(apiInterface.getBranches(TestConst.TEST_OWNER, TestConst.TEST_REPO)).thenReturn(Observable.just(Arrays.asList(branchDTOs)));

    TestSubscriber<List<BranchDTO>> testSubscriber = new TestSubscriber<>();
    model.getRepoBranches(TestConst.TEST_OWNER, TestConst.TEST_REPO).subscribe(testSubscriber);

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

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

    assertEquals(3, actual.size());
    assertEquals("QuickStart", actual.get(0).getName());
    assertEquals("94870e23f1cfafe7201bf82985b61188f650b245", actual.get(0).getCommit().getSha());
}
```

Проверим покрытие в Jacoco:

app > com.andrey7mel.stepbystep.model > ModelImpl

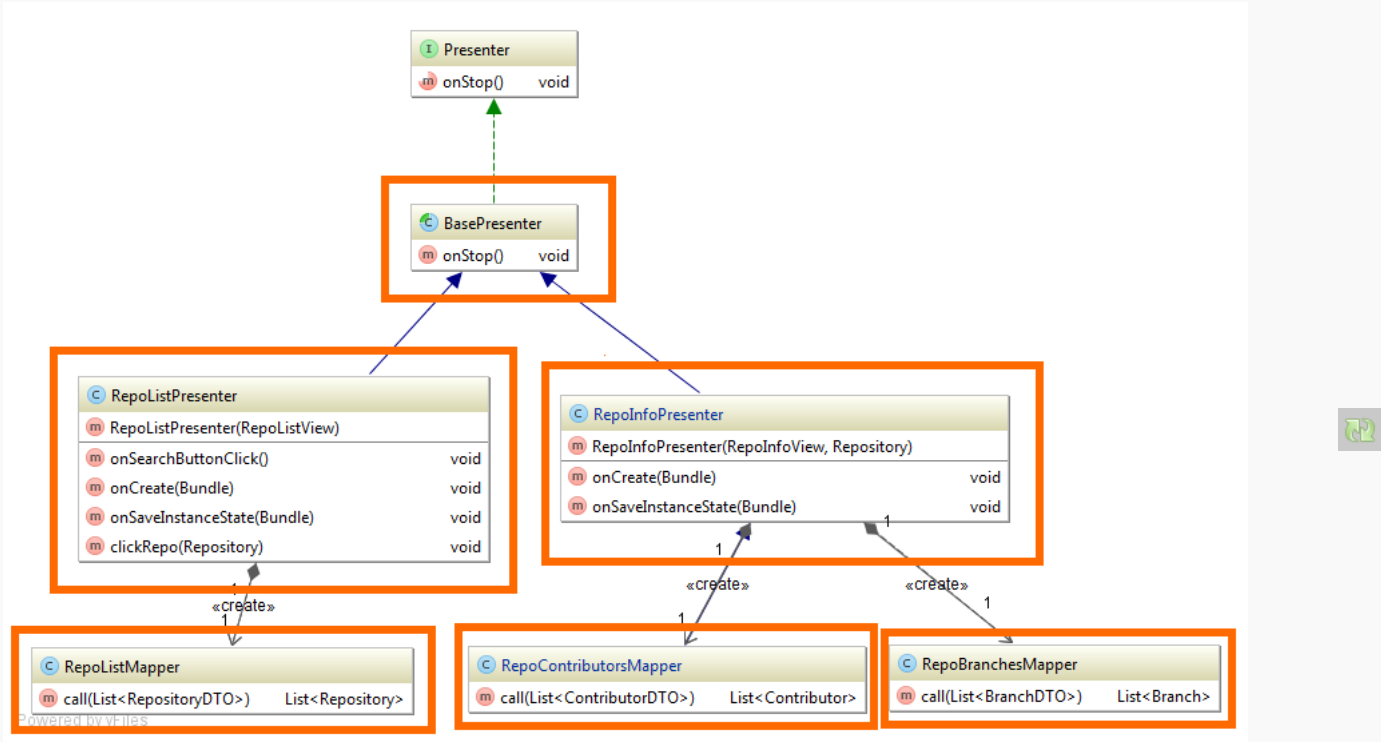
ModelImpl

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Meth
ModelImpl()		100%		n/a	0	1	0	4	0	
getRepoBranches(String, String)		100%		n/a	0	1	0	3	0	
getRepoContributors(String, String)		100%		n/a	0	1	0	3	0	
getRepoList(String)		100%		n/a	0	1	0	3	0	
applySchedulers()		100%		n/a	0	1	0	1	0	
Total	0 of 39	100%	0 of 0	n/a	0	5	0	14	0	

Presenter

В presenter слое нам необходимо протестировать работу мапперов и работу презентеров.

Схема Presenter слоя, классы требующие тестирования помечены красным



С мапперами все достаточно просто. Считываем json из файлов, преобразуем и проверяем.
С презентерами — мокаем model и проверяем вызовы необходимых методов у view. Также необходимо проверить корректность onSubscribe и onStop, для этого перехватываем подписку (Subscription) и проверяем isUnsubscribed

Пример тестов в presenter слое

```
@Before
public void setUp() throws Exception {
    super.setUp();
    component.inject(this);

    activityCallback = mock(ActivityCallback.class);

    mockView = mock(RepoListView.class);
    repoListPresenter = new RepoListPresenter(mockView, activityCallback);

    doAnswer(invocation -> Observable.just(repositoryDTOs))
            .when(model)
            .getRepoList(TestConst.TEST_OWNER);

    doAnswer(invocation -> TestConst.TEST_OWNER)
            .when(mockView)
```

```
                .getUserName();
    }


    @Test
    public void testLoadData() {
        repoListPresenter.onCreateView(null);
        repoListPresenter.onSearchButtonClick();
        repoListPresenter.onStop();

        verify(mockView).showRepoList(repoList);
    }

    @Test
    public void testSubscribe() {
        repoListPresenter = spy(new RepoListPresenter(mockView, activityCallback)); //for ArgumentCaptor
        repoListPresenter.onCreateView(null);
        repoListPresenter.onSearchButtonClick();
        repoListPresenter.onStop();

        ArgumentCaptor<Subscription> captor = ArgumentCaptor.forClass(Subscription.class);
        verify(repoListPresenter).addSubscription(captor.capture());
        List<Subscription> subscriptions = captor.getAllValues();
        assertEquals(1, subscriptions.size());
        assertTrue(subscriptions.get(0).isUnsubscribed());
    }
```

Смотрим изменение в JaCoCo:

app > com.andrey7mel.stepbystep.presenter Source Files

com.andrey7mel.stepbystep.presenter

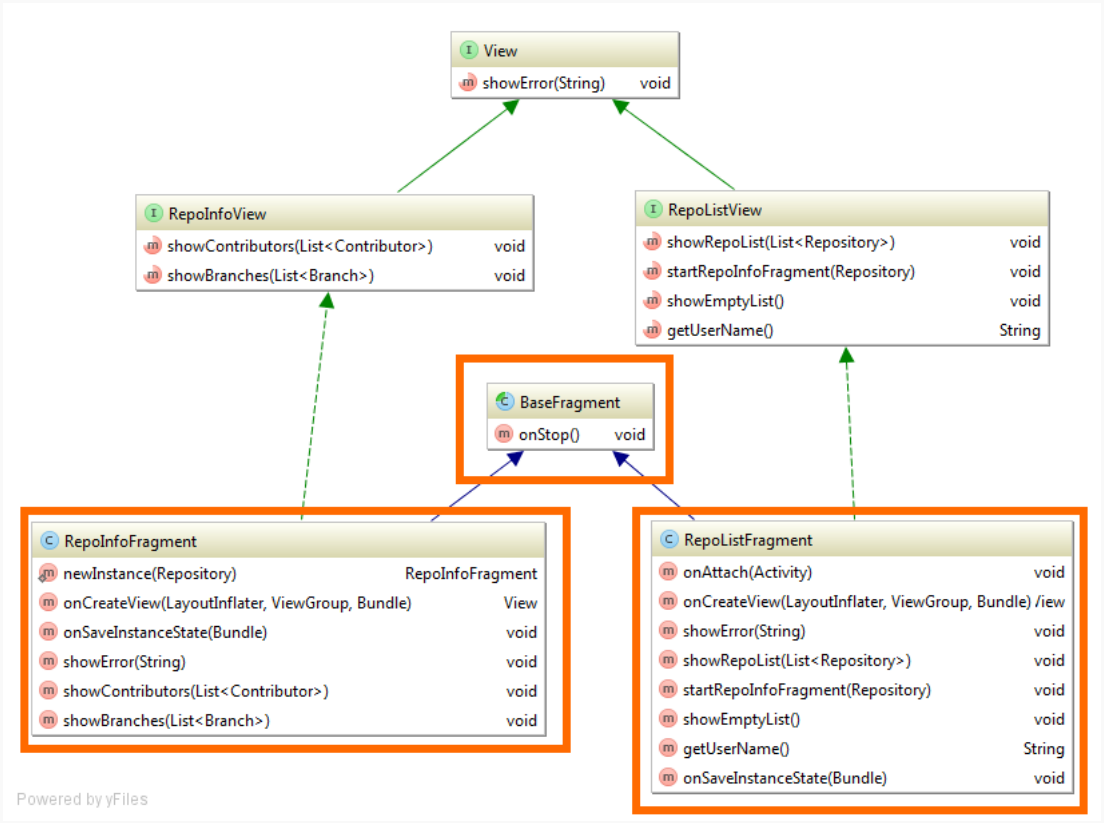
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
RepoInfoPresenter		100%		70%	3	10	0	29	0	5	0	1
RepoListPresenter		100%		83%	2	12	0	23	0	6	0	1
RepoListPresenter.new Observer() {...}		100%		100%	0	6	0	9	0	4	0	1
RepoInfoPresenter.new Observer() {...}		100%		n/a	0	4	0	7	0	4	0	1
RepoInfoPresenter.new Observer() {...}		100%		n/a	0	4	0	7	0	4	0	1
BasePresenter		100%		n/a	0	3	0	7	0	3	0	1
Total	0 of 295	100%	5 of 26	81%	5	39	0	79	0	26	0	6

Created with JaCoCo 0.7.1.201

View

При тестирование View слоя, нам необходимо проверить только вызовы методов жизненного цикла презентера из фрагмента. Вся логика содержится в презентерах.

Схема View слоя, классы требующие тестирования помечены красным



▾ Пример тестирования фрагмента

```
@Test
public void testOnCreateViewWithBundle() {
   repoInfoFragment.onCreateView(LayoutInflater.from(activity), (ViewGroup) activity.findViewById(R.id.container), bundle);
   verify(repoInfoPresenter).onCreateView(bundle);
}

@Test
public void testOnStop() {
   repoInfoFragment.onStop();
   verify(repoInfoPresenter).onStop();
}

@Test
public void testOnSaveInstanceState() {
   repoInfoFragment.onSaveInstanceState(null);
   verify(repoInfoPresenter).onSaveInstanceState(null);
}
```

Финальное покрытие тестами:

app

app

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	C
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 0.7.1.201

Заключение или to be continued…

Во второй части статьи мы рассмотрели внедрение Dagger 2 и покрыли код unit тестами. Благодаря использованию MVP и подмене инжекций смогли быстро написать тесты на все части приложения. Весь код доступен на github. Статья написана при активном участии @nnesterov. В следующей части рассмотрим интеграционное и функциональное тестирование, а также поговорим про TDD.

UPDATE

Построение Android приложений шаг за шагом, часть третья

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

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

Arvifox
Профиль
Трекер
Настройки

Разделы
Публикации
Хабы
Компании
Пользователи
Песочница

Информация
О сайте
Правила
Помощь
Соглашение
Помощь стартапам

Услуги
Реклама
Тарифы
Контент
Семинары

Приложения
Загрузите в App Store
ДОСТУПНО в Google

Служба поддержки
Мобильная версия