Unit testing Android apps

Igor Filippov

About me

- Developing mobile apps for more than 5 years
- Primarily Android, but also iOS
- Experience with other languages and frameworks: Go, Ruby, Python, JavaScript, Node

Working at Memorado - leading brain training app

memorado.com

Why bother at all?

- Manual testing doesn't scale
- CI and releases are easier when app is well-tested
- Test-first approach results in better code

What is a unit test?

- Unit tests focus on single class
- Unaware of external configuration, environment
- Real collaborators replaced with test doubles

Why should you focus on unit tests?

- Fast to execute, unit test run directly on your machine
- Easier to maintain
- Faster and easier to develop(compared to integration, UI tests)
- TDD

What do you need to develop unit tests on Android?

1. Separate business logic from platform code

2. SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion)

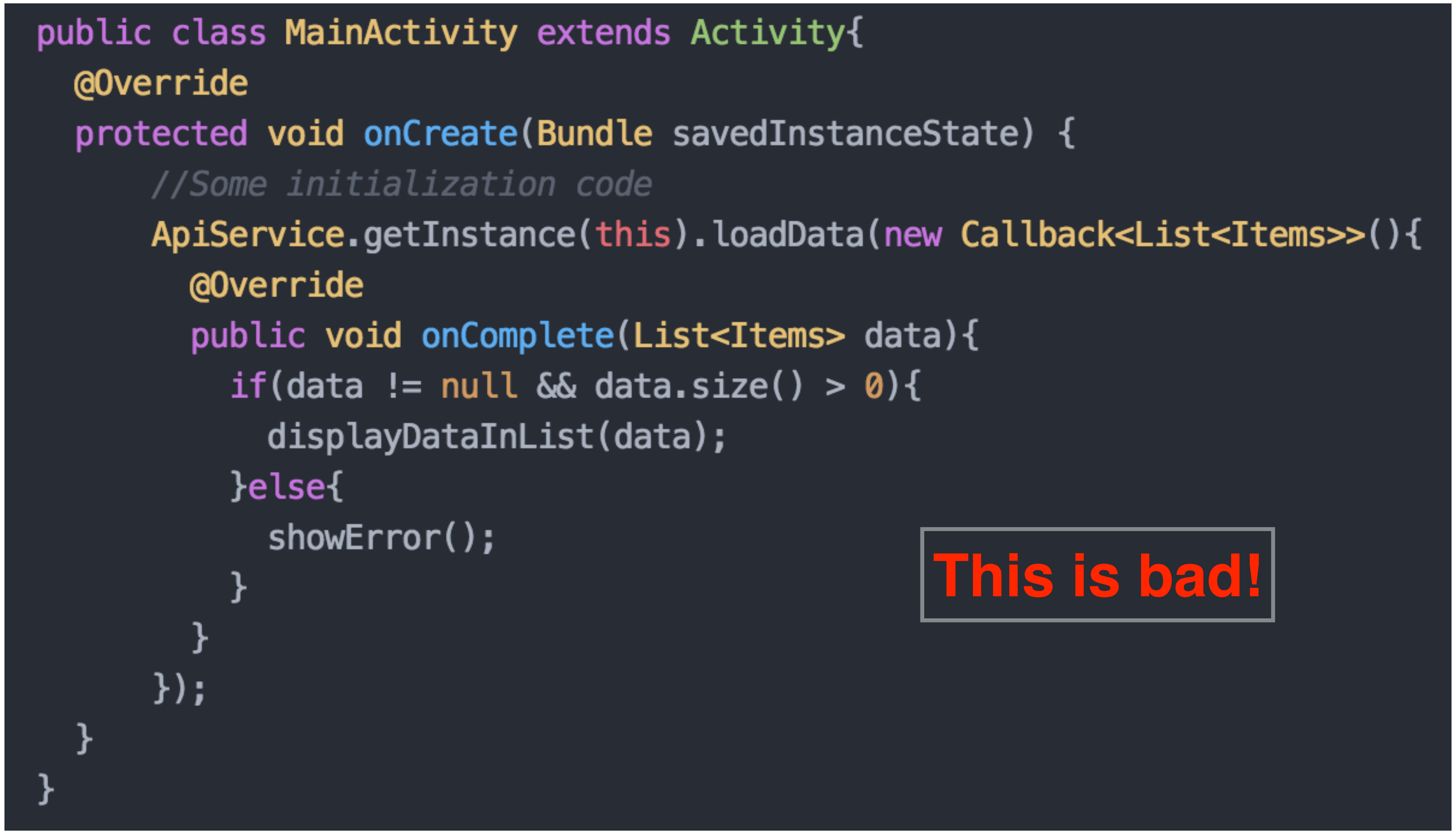
3. Use the tools! JUnit, Mockito, Robolectric

Decouple business logic from platform code

It’s very important to keep your business logic **away** from Activities, Fragments, Services, BroadcastReceivers.

Decouple business logic from platform code

Use **Model-View-Presenter**. Your Fragments and Activities should only contain *view* logic.



```
public class MainActivity extends Activity{
  @Override
  protected void onCreate(Bundle savedInstanceState) {
      //Some initialization code
      ApiService.getInstance(this).loadData(new Callback<List<Items>>(){
        @Override
        public void onComplete(List<Items> data){
          if(data != null && data.size() > 0){
            displayDataInList(data);
          }else{
            showError();
          }
        }
      });
  }
}
```

Decouple business logic from platform code

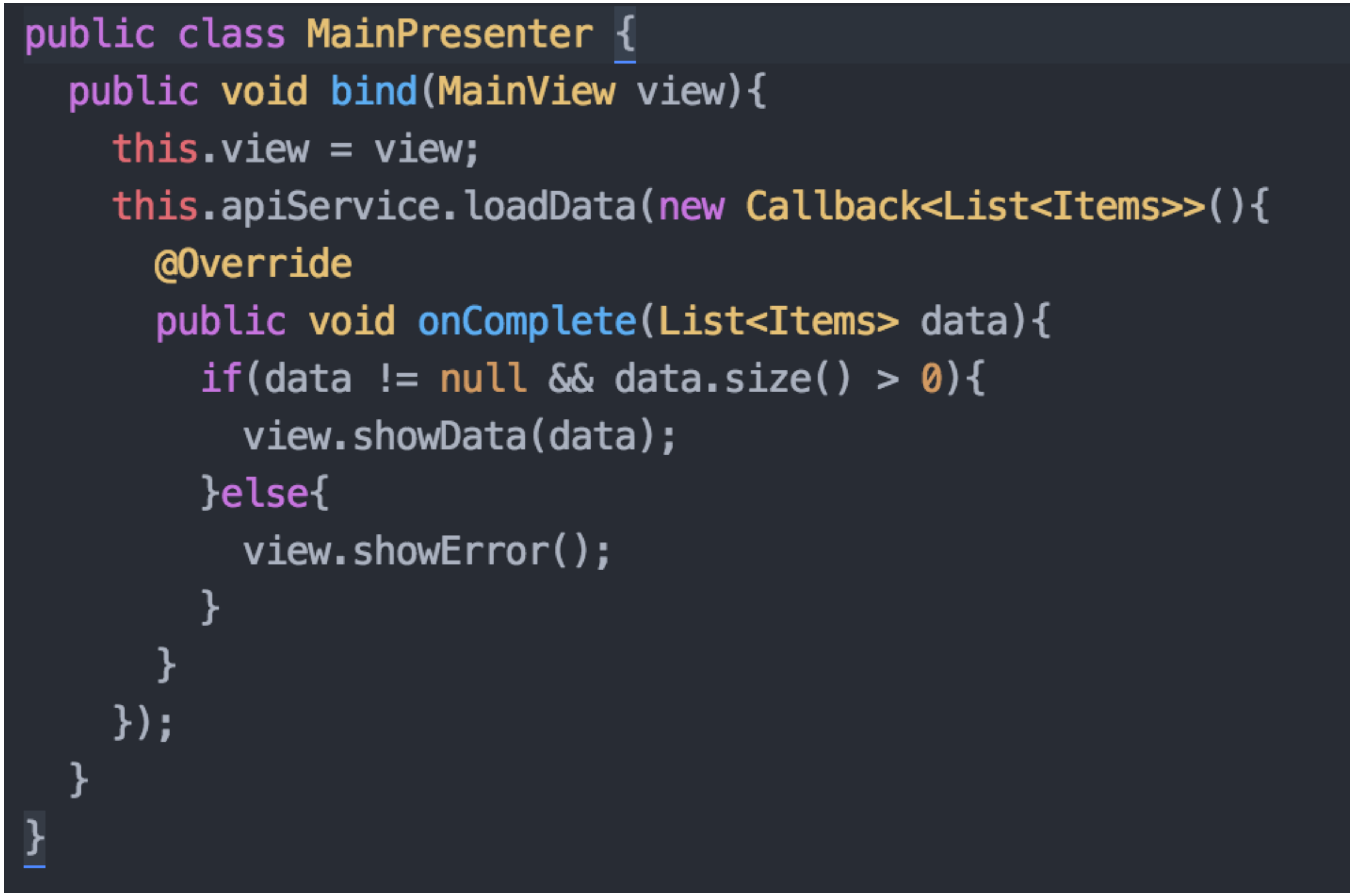
View is only responsible for displaying data

```java
public interface MainView {
  void showData(List<Data> data);
  void showError();
}
```

```java
public class MainActivity extends Activity implements MainView {
  @Override protected void onCreate(Bundle savedInstanceState) {
    this.mainPresenter = new mainPresenter();
    mainPresenter.bind();
  }
  @Override void showData(List<Data> data){
    //Display data in a list
  }
  @Override void showError(){
    //Show error view
  }
}
```

Decouple business logic from platform code

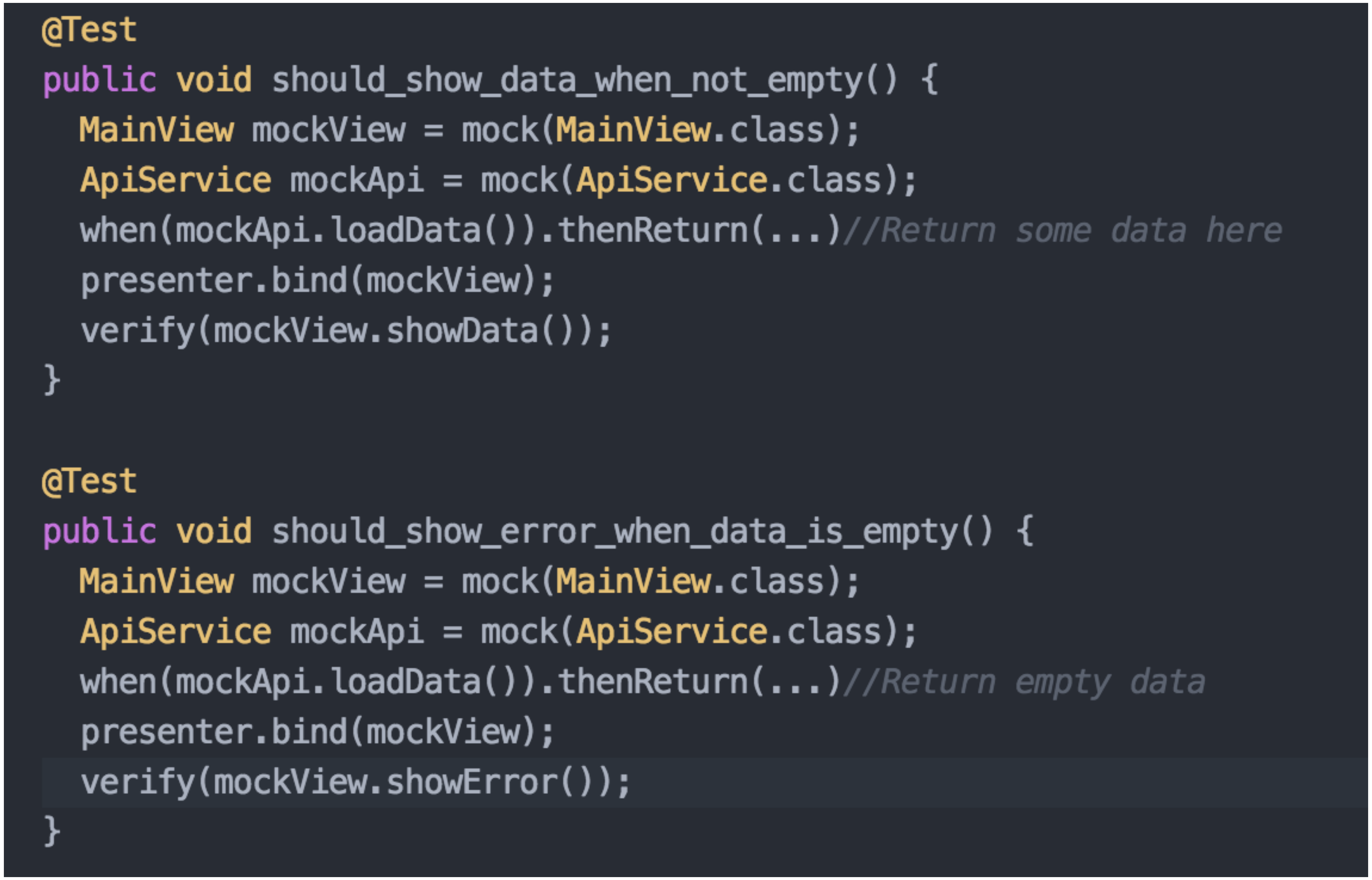
Presenter decides what and when to show



```java
public class MainPresenter {
  public void bind(MainView view){
    this.view = view;
    this.apiService.loadData(new Callback<List<Items>>(){
      @Override
      public void onComplete(List<Items> data){
        if(data != null && data.size() > 0){
          view.showData(data);
        }else{
          view.showError();
        }
      }
    });
  }
}
```

Decouple business logic from platform code

Your **Presenter** test might look like this:



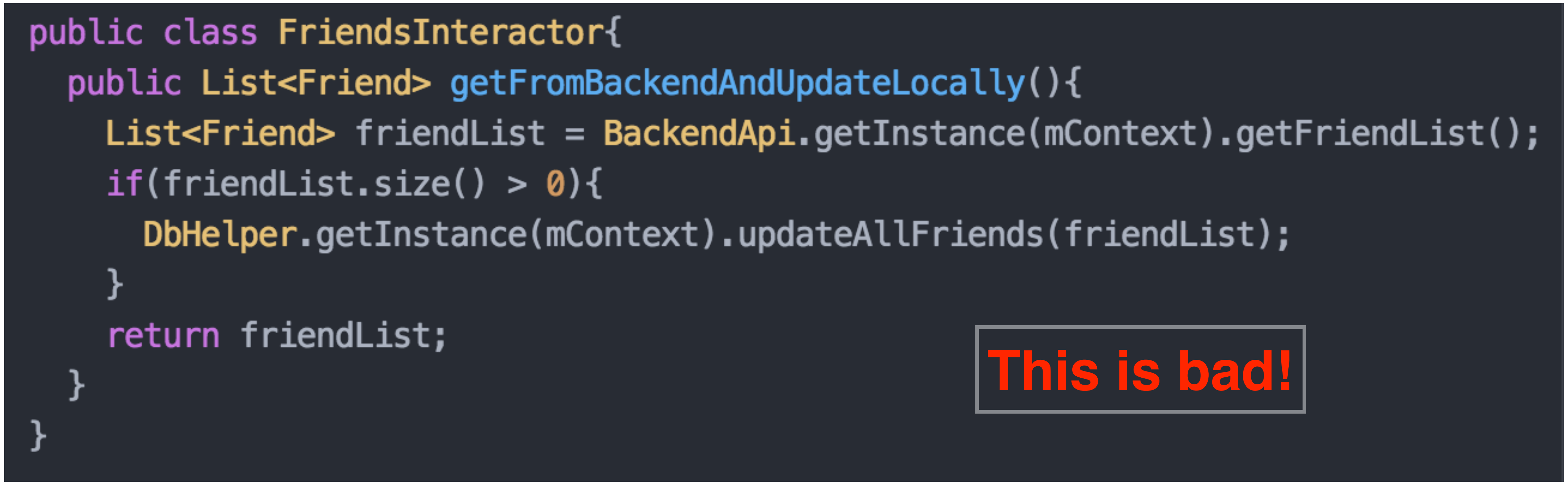
```
@Test
public void should_show_data_when_not_empty() {
  MainView mockView = mock(MainView.class);
  ApiService mockApi = mock(ApiService.class);
  when(mockApi.loadData()).thenReturn(...)//Return some data here
  presenter.bind(mockView);
  verify(mockView.showData());
}

@Test
public void should_show_error_when_data_is_empty() {
  MainView mockView = mock(MainView.class);
  ApiService mockApi = mock(ApiService.class);
  when(mockApi.loadData()).thenReturn(...)//Return empty data
  presenter.bind(mockView);
  verify(mockView.showError());
}
```

Managing dependencies

- It should be clear which dependencies your class has
- You should be able to replace dependencies easily
- Try not to use DI(Dagger, Roboguice) frameworks right away, get some feeling for DI first

Managing dependencies

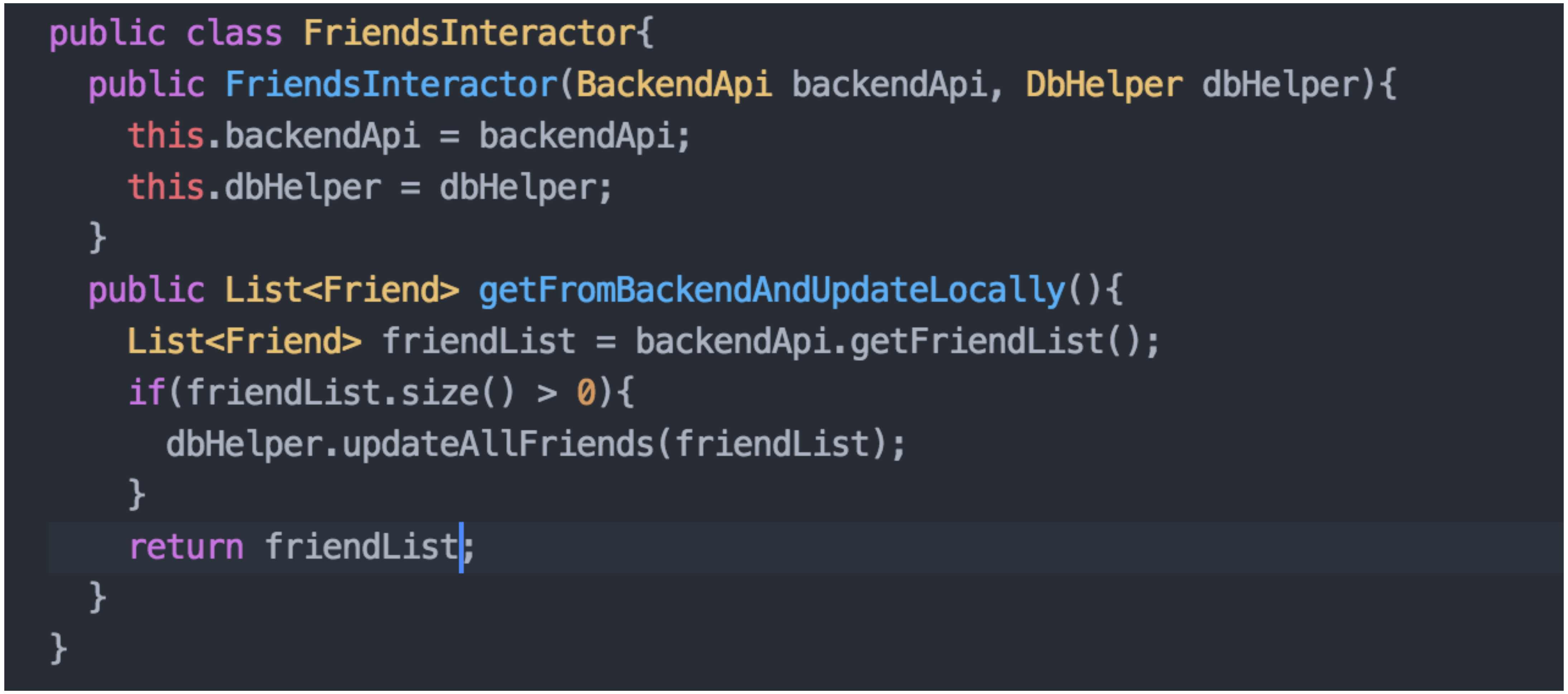
It’s hard to test code which extensively uses singletons



```
public class FriendsInteractor{
  public List<Friend> getFromBackendAndUpdateLocally(){
    List<Friend> friendList = BackendApi.getInstance(mContext).getFriendList();
    if(friendList.size() > 0){
      DbHelper.getInstance(mContext).updateAllFriends(friendList);
    }
    return friendList;
  }
}
```

Managing dependencies

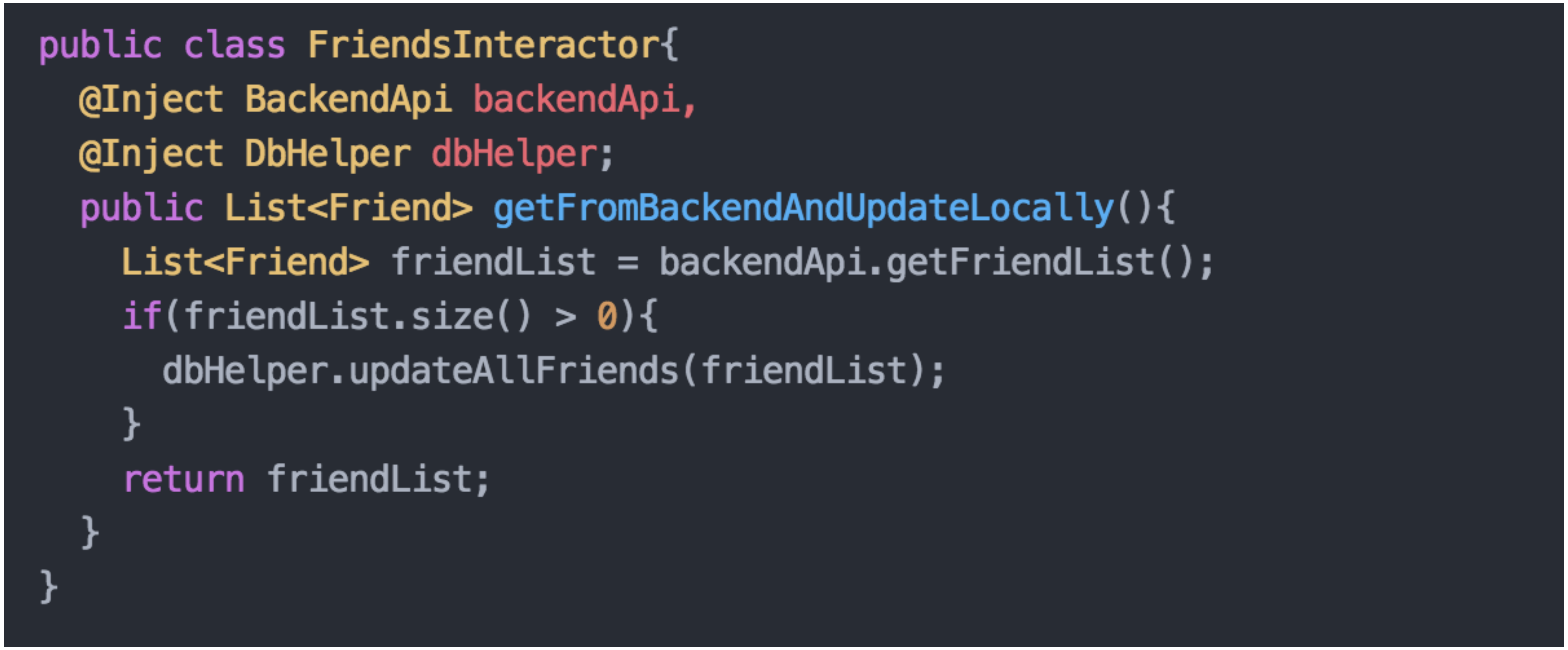
Constructor injections helps you see what kind of dependencies given class has



```java
public class FriendsInteractor{
  public FriendsInteractor(BackendApi backendApi, DbHelper dbHelper){
    this.backendApi = backendApi;
    this.dbHelper = dbHelper;
  }
  public List<Friend> getFromBackendAndUpdateLocally(){
    List<Friend> friendList = backendApi.getFriendList();
    if(friendList.size() > 0){
      dbHelper.updateAllFriends(friendList);
    }
    return friendList;
  }
}
```

Managing dependencies

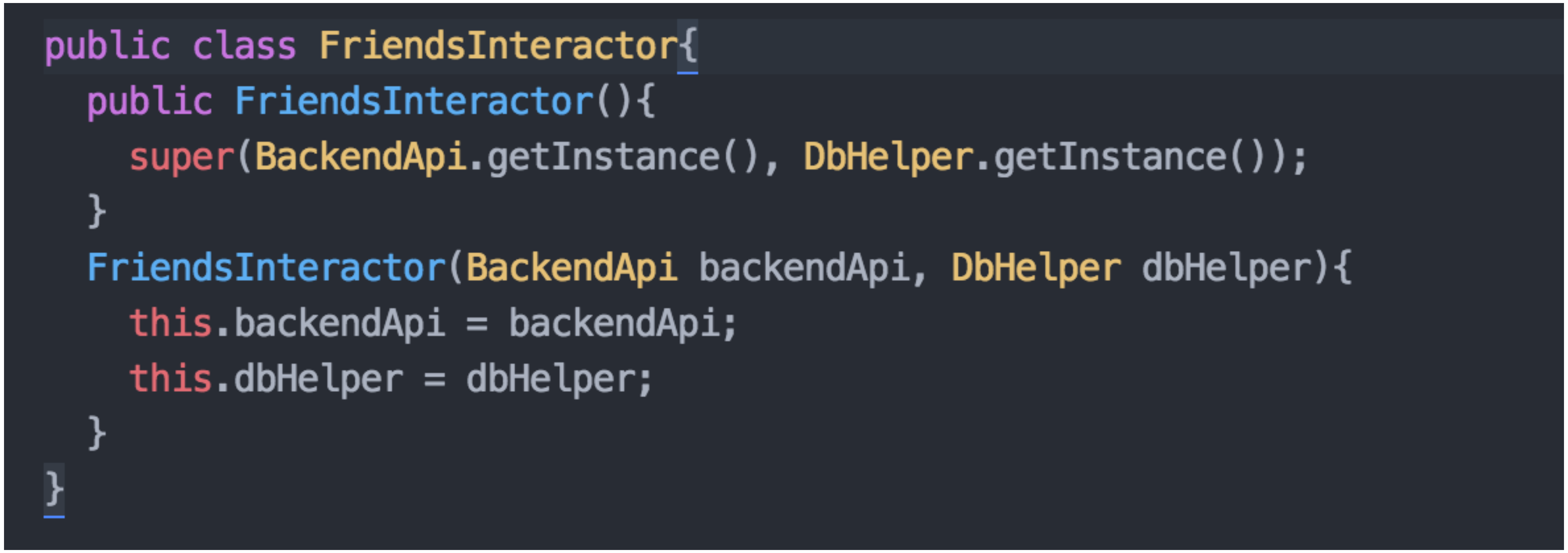
DI frameworks(Dagger, Roboguice) add overhead. Only use them as last resort



```
public class FriendsInteractor{
  @Inject BackendApi backendApi,
  @Inject DbHelper dbHelper;
  public List<Friend> getFromBackendAndUpdateLocally(){
    List<Friend> friendList = backendApi.getFriendList();
    if(friendList.size() > 0){
      dbHelper.updateAllFriends(friendList);
    }
    return friendList;
  }
}
```

Managing dependencies

Use package private constructor in tests



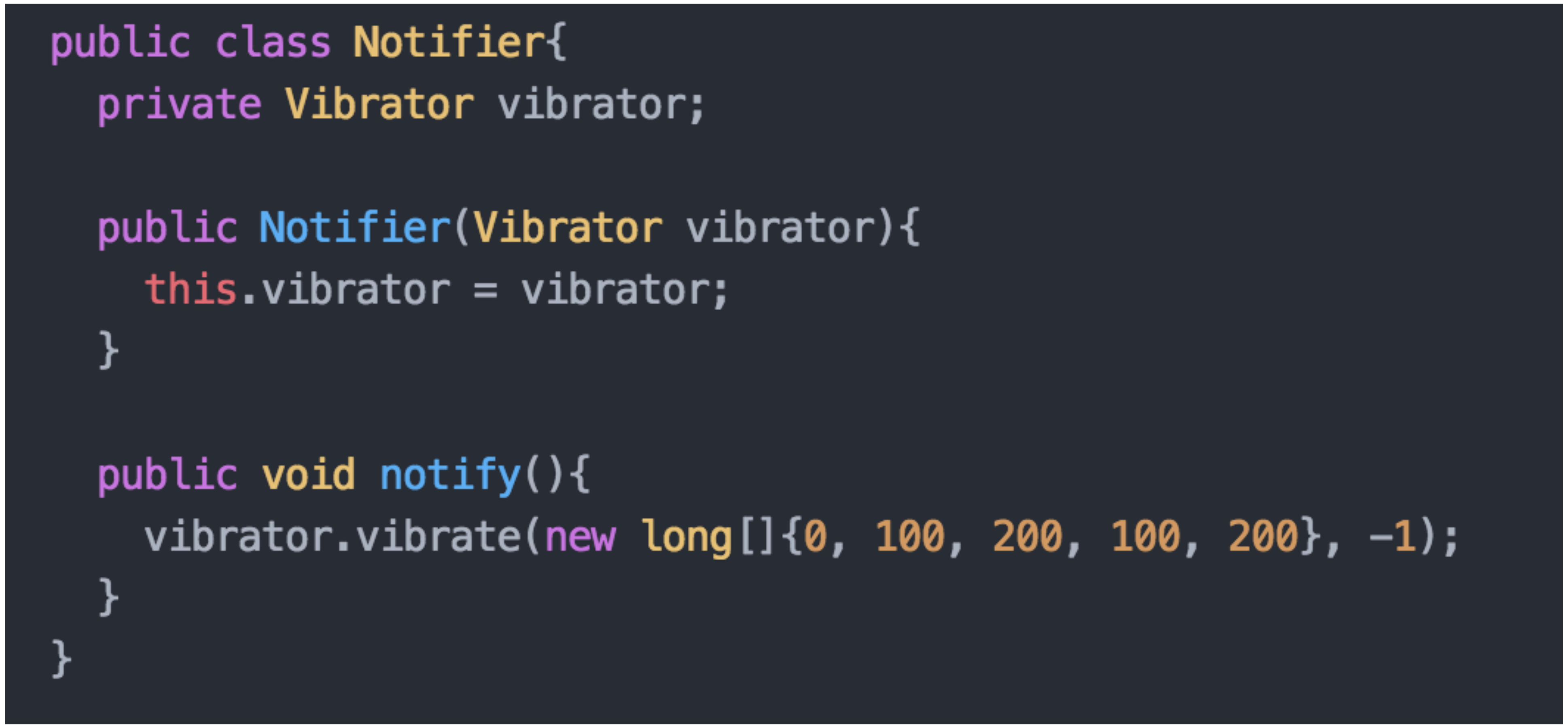
```java
public class FriendsInteractor{
  public FriendsInteractor(){
    super(BackendApi.getInstance(), DbHelper.getInstance());
  }
  FriendsInteractor(BackendApi backendApi, DbHelper dbHelper){
    this.backendApi = backendApi;
    this.dbHelper = dbHelper;
  }
}
```

Use the tools!

- JUnit - testing framework
- Robolectric - use to deal with Android dependencies(Activities, Fragments)
- Mockito - create test doubles, verify behaviour
- AssertJ - fluent assertions

Mockito

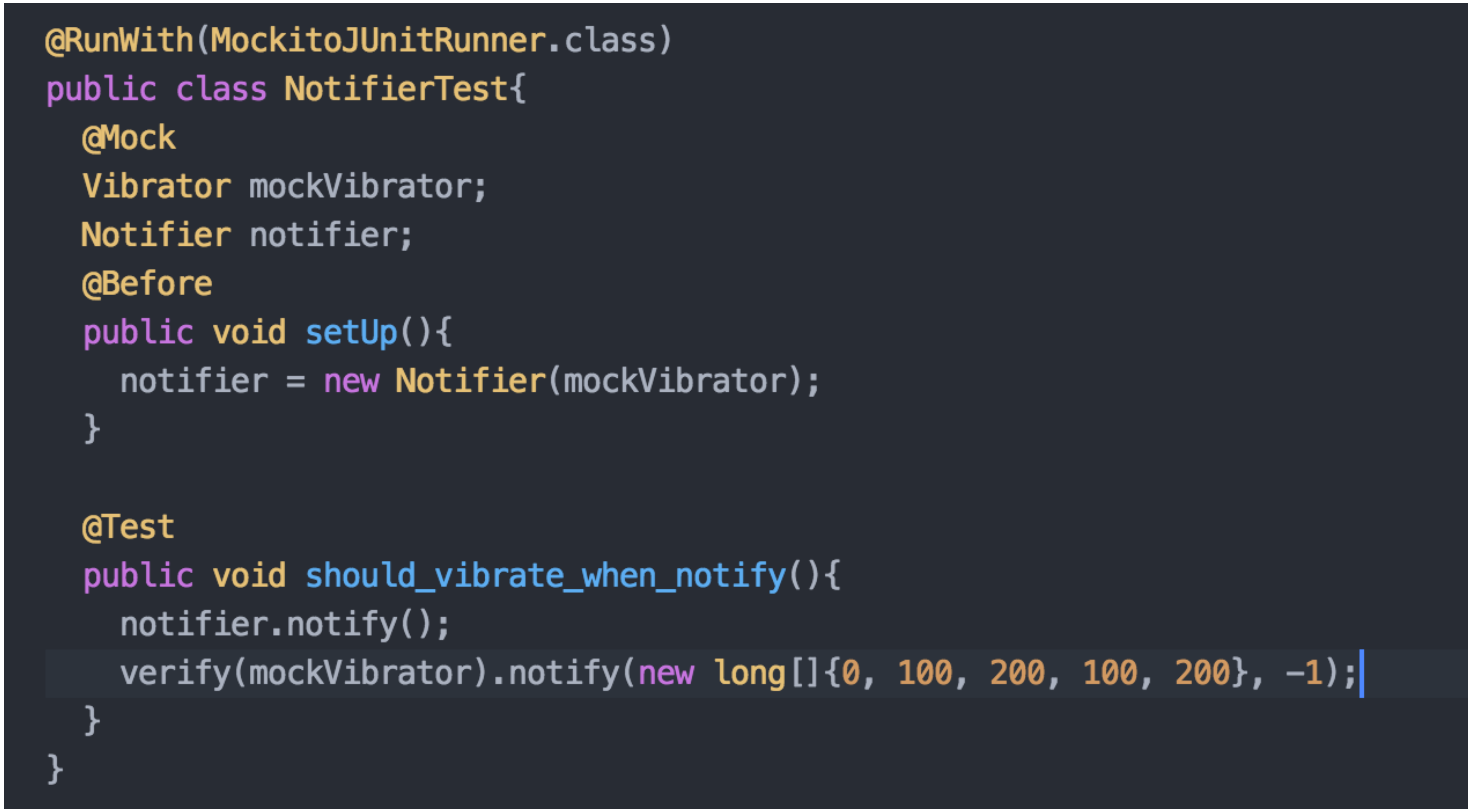
Verify interaction with dependencies



```
public class Notifier{
  private Vibrator vibrator;

  public Notifier(Vibrator vibrator){
    this.vibrator = vibrator;
  }

  public void notify(){
    vibrator.vibrate(new long[]{0, 100, 200, 100, 200}, -1);
  }
}
```

Mockito

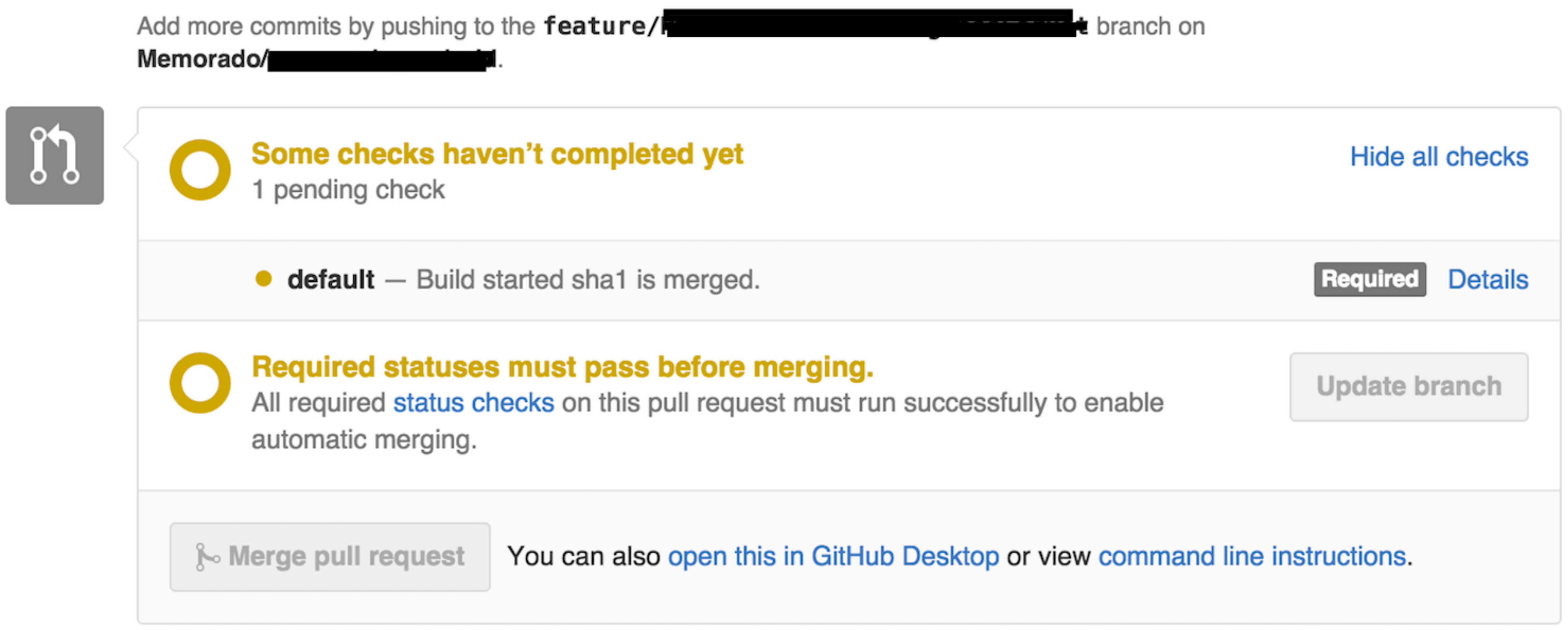
Verify interaction with dependencies



```
@RunWith(MockitoJUnitRunner.class)
public class NotifierTest{
  @Mock
  Vibrator mockVibrator;
  Notifier notifier;
  @Before
  public void setUp(){
    notifier = new Notifier(mockVibrator);
  }

  @Test
  public void should_vibrate_when_notify(){
    notifier.notify();
    verify(mockVibrator).notify(new long[]{0, 100, 200, 100, 200}, -1);
  }
}
```

Quality-driven process

Run all the tests on CI before merging feature



Add more commits by pushing to the `feature/`[redacted] branch on **Memorado/**[redacted].

Some checks haven't completed yet
1 pending check

Hide all checks

● **default** — Build started sha1 is merged.

Required Details

Required statuses must pass before merging.
All required status checks on this pull request must run successfully to enable automatic merging.

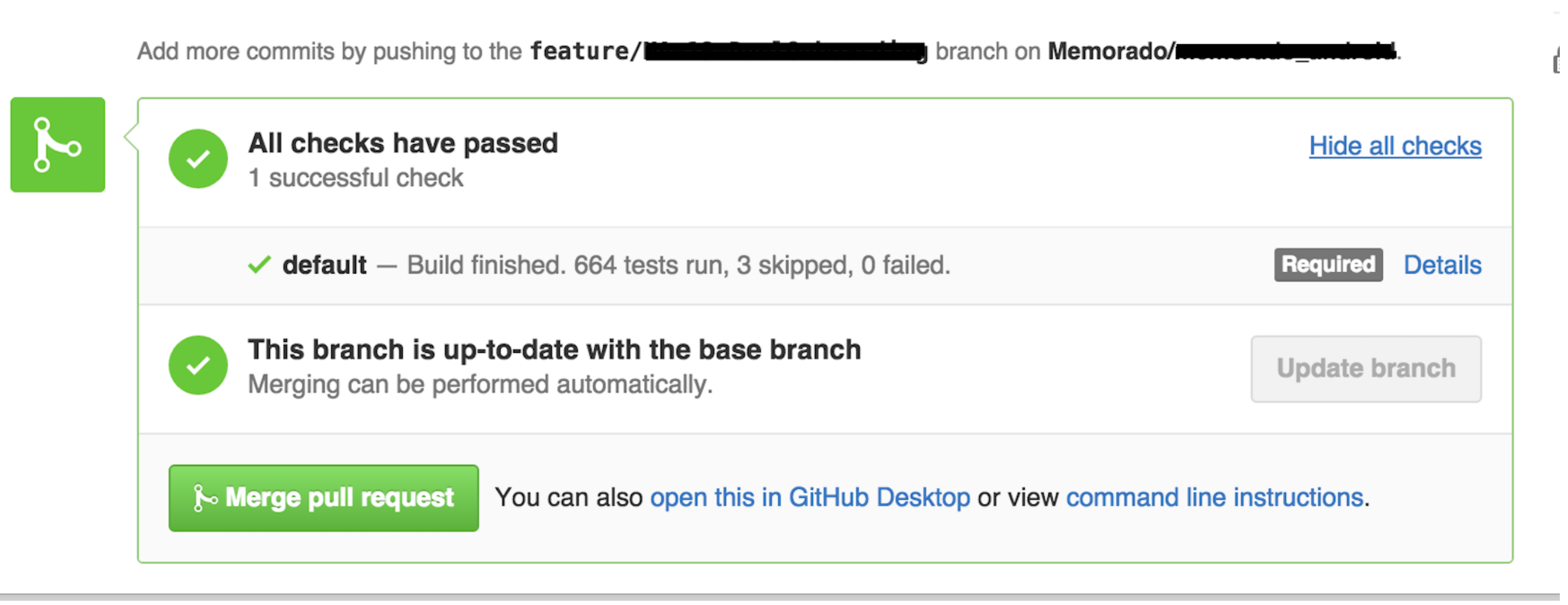
Update branch

Merge pull request

You can also open this in GitHub Desktop or view command line instructions.

Quality-driven process

Run all the tests on CI before merging feature



Add more commits by pushing to the **feature/**[redacted] branch on **Memorado/**[redacted].

All checks have passed
1 successful check

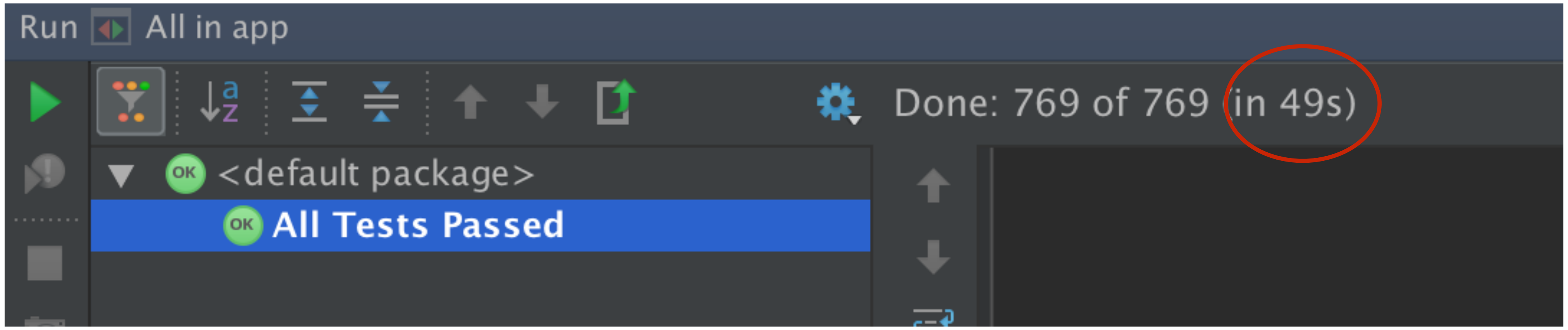
✓ **default** — Build finished. 664 tests run, 3 skipped, 0 failed.

This branch is up-to-date with the base branch
Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

Quality-driven process

Unit tests are really fast, no need to deploy to device



Run All in app

Done: 769 of 769 (in 49s)

<default package>

All Tests Passed

Closing remarks

- In the end it's all about your architecture
- If you follow software development good practices, then you should be able to write unit tests
- https://github.com/f6v/android-good-practices

Join Memorado!

We're making people smarter and we're looking for Android developers!

igor.filippov@memorado.com