AutSoft


# Roland Mostoha

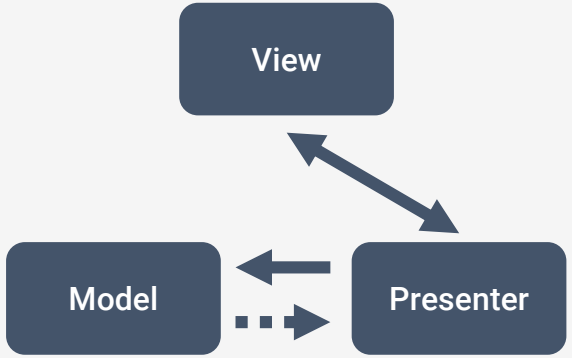
Squad Lead @ AutSoft


## Effective Automated Testing

# Motivation




Dependency Injection

# Agenda

1

## Testing principles

- What to test?
- Advantages of Unit tests

2

## Tools & Techniques

- Hermetic testing
- DIP + DI
- Mocking

3

## JUnit 5

- JUnit 5 on Android
- JUnit 5 features

# What to test?

- Business requirements defined in the Specification or User Story
- In lower levels
  - Own business logic, algorithms etc.

**User Story**

*"After the user entered a valid PIN code, we navigate him to home screen"*

→

```
@Test
public void givenValidPin_whenWeSubmit_thenMainScreenShouldShown … {
    // Given
    onView(ViewMatchers.withId(R.id.input_pin))
            .perform(typeText("123456"))
            .perform(closeSoftKeyboard());
    // When
    onView(ViewMatchers.withId(R.id.btn_submit))
            .perform(click());
    // Then
    onView(ViewMatchers.withText(R.string.home_title))
            .check(matches(isDisplayed()));
}
```

# What not to test?

- Don’t test source code, that not contains any business logic
- Don’t test external dependencies, framework’s functions
  - We don’t own the source code, it is not our responsibility

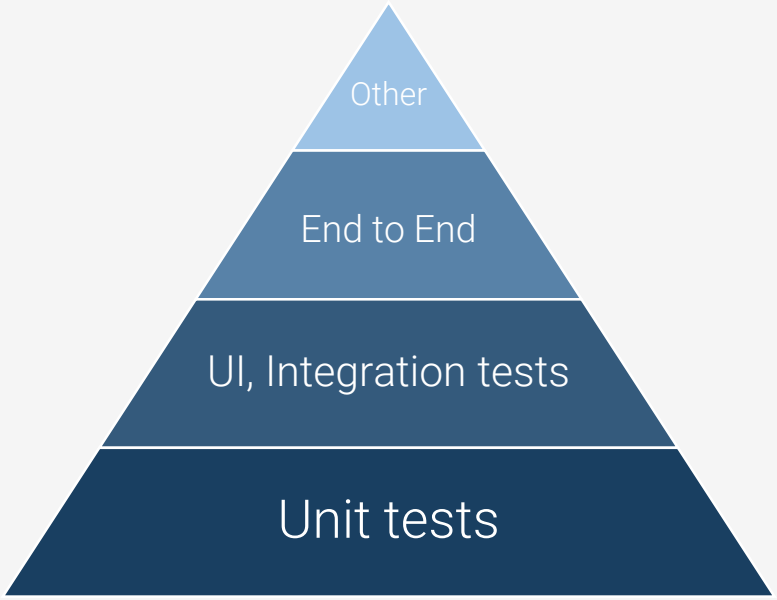
```
// THIS TEST IS NOT USEFUL FOR US!
@Test
public void givenInput_whenWeTypeIt_thenInputFieldHasText()…{
   // Given
   String input = "Text";
   // When
   onView(withId(R.id.input_name)).perform(typeText(input));
   // Then
   onView(withId(R.id.input_name)).check(matches(withText(input)));
}
```

# Unit Tests Above All

# Advantages of Unit tests

Debugging

Source Maintainability




Duration

Cost, resource

Reliability

# Agenda

1

## Testing principles

- What to test?
- Testing levels
- Testing pyramid

2

## Tools & Techniques

- Hermetic testing
- DIP + DI
- Mocking

3

## JUnit 5

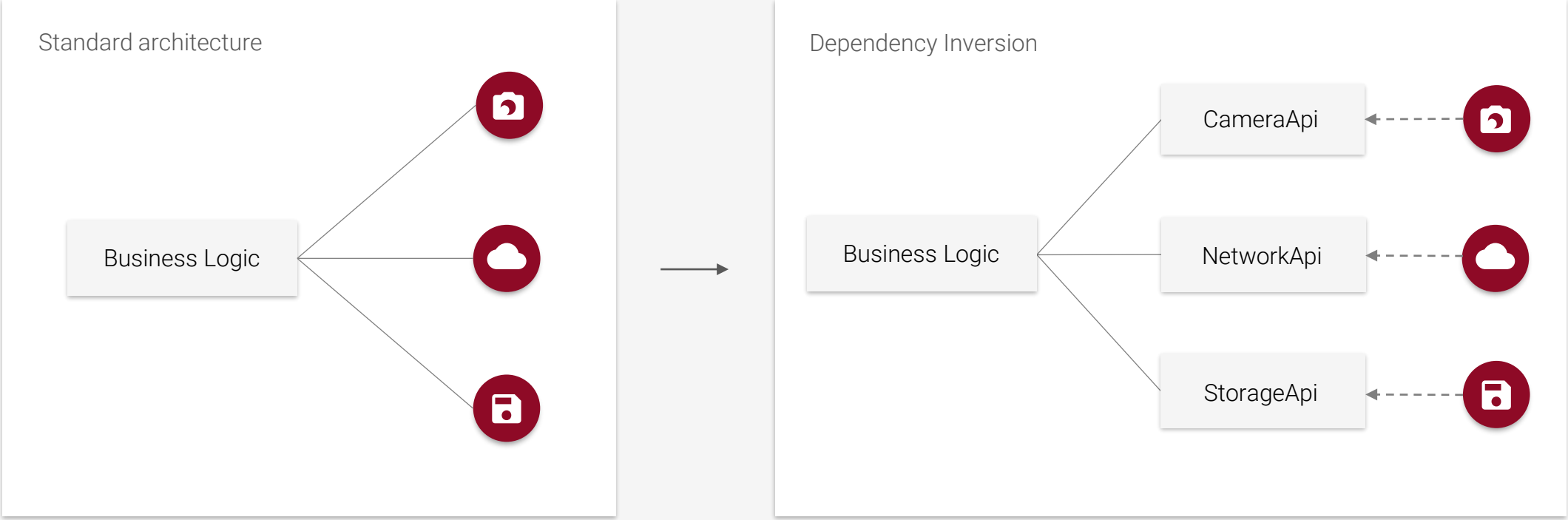
- JUnit 5 on Android
- JUnit 5 features

# JUnit


- Running in the JVM
  - Pro: We can run tests at milliseconds in any Java Environment
  - Contra: Can't work with network, database, sensors, camera etc.
- **Problem:** We have a lot of external dependencies in Android platform
- **Solution**: Decouple the behavior of external dependencies
- JUnit **force us** to use hermetic testing for unit tests

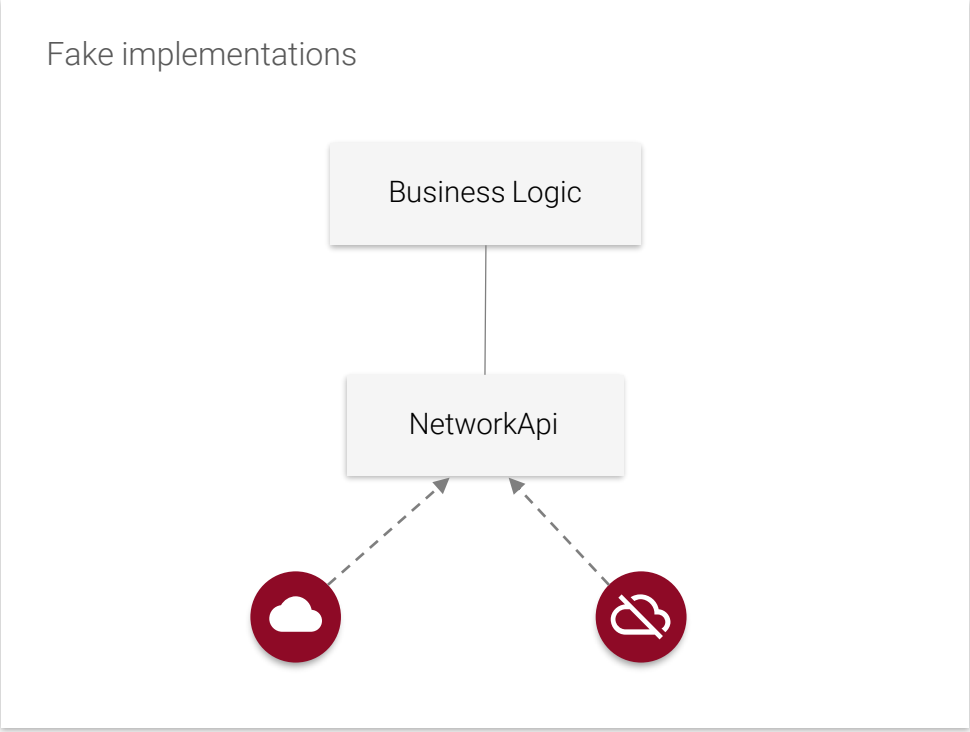
```
java.lang.RuntimeException: Method isEmpty in android.text.TextUtils not mocked.
See http://g.co/androidstudio/not-mocked for details.
```

# Dependency Inversion Principle



# Dependency Inversion Principle

- Low coupling



# Dependency Injection

- A class supplies its external dependencies
- Decreasing the responsibility - Increasing the cohesion

```
public PinPresenter(PinView pinView) {
   this.pinView = pinView;
   this.authenticationApi = new DefaultAuthenticationService();
   this.preferenceApi = new InMemoryPreferences();
}
```

↓

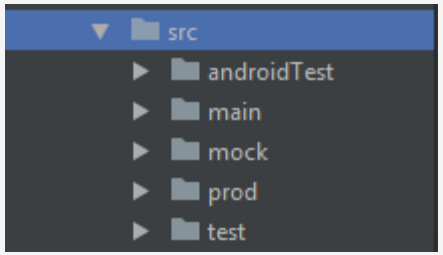
```
public PinPresenter(PinView pinView, AuthenticationApi authenticationApi, PreferenceApi preferenceApi) {
   this.pinView = pinView;
   this.authenticationApi = authenticationApi;
   this.preferenceApi = preferenceApi;
}
```

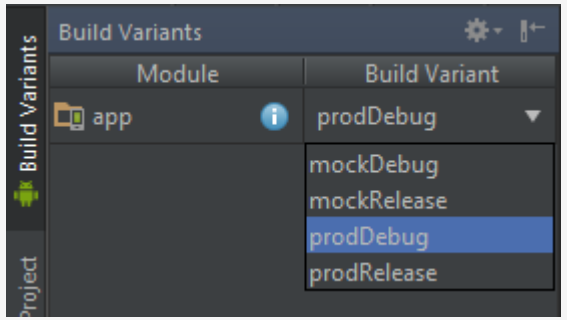
Business Logic can be initiated by fake implementations.

Tools & Techniques

# DIP + DI on Android

```
productFlavors {
    mock {
        applicationIdSuffix = ".mock"
    }
    prod {
    }
}
```



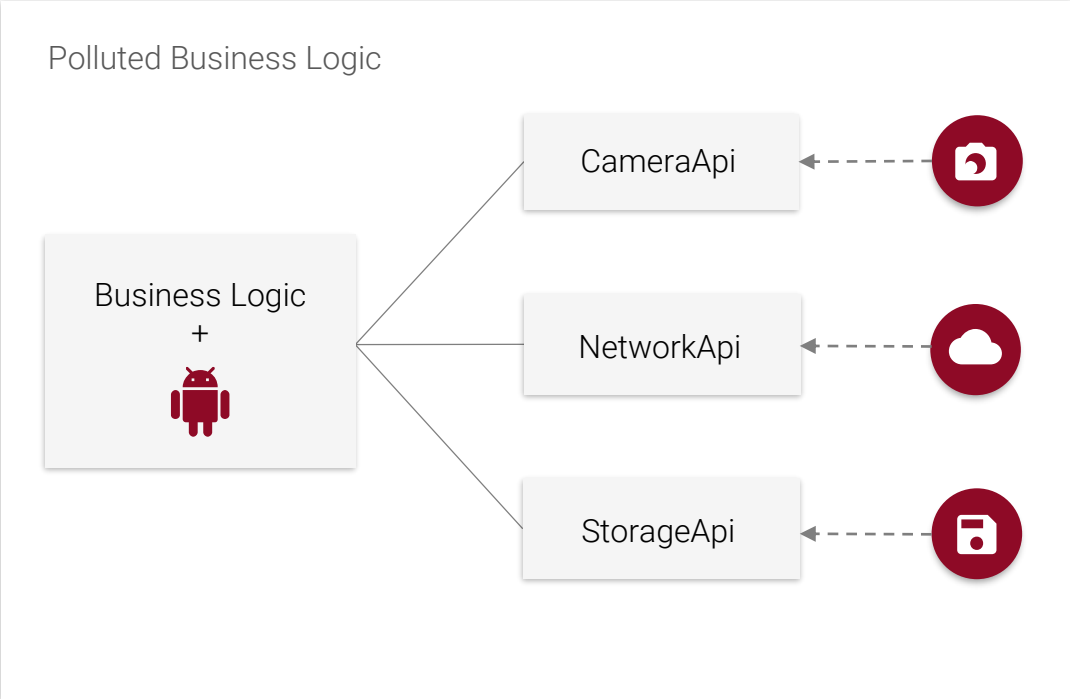



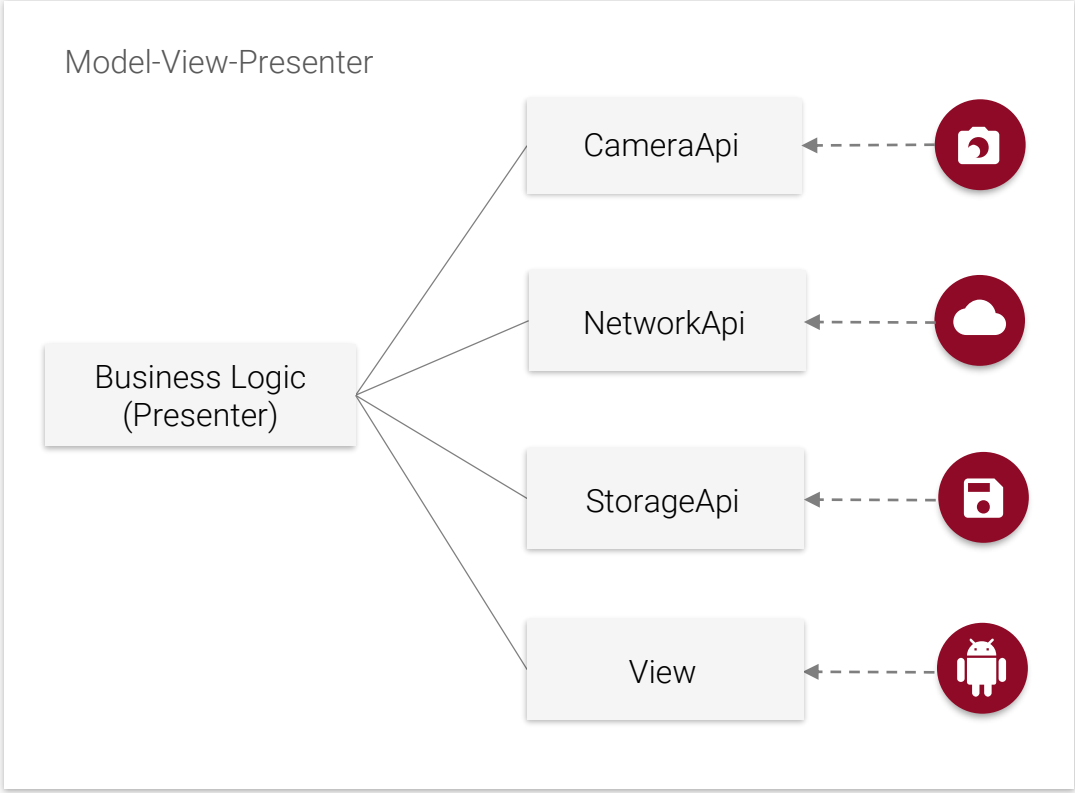

```
/**
 * Provides Mock APIs.
 */
public class Injection {
    public static PreferenceApi providePreferenceApi() {
        return new InMemoryPreferences();
    }
    public static AuthenticationApi provideAuthenticationApi() {
        return new MockAuthenticationService();
    }
}
```

```
/**
 * Provides active network and database APIs.
 */
public class Injection {
    public static PreferenceApi providePreferenceApi() {
        return new DefaultPreferences();
    }
    public static AuthenticationApi provideAuthenticationApi() {
        return new DefaultAuthenticationService();
    }
}
```

Tools & Techniques

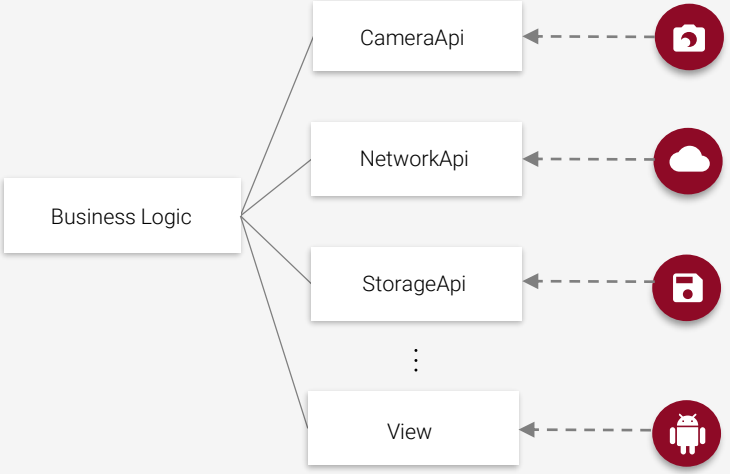
# Model-View-Presenter



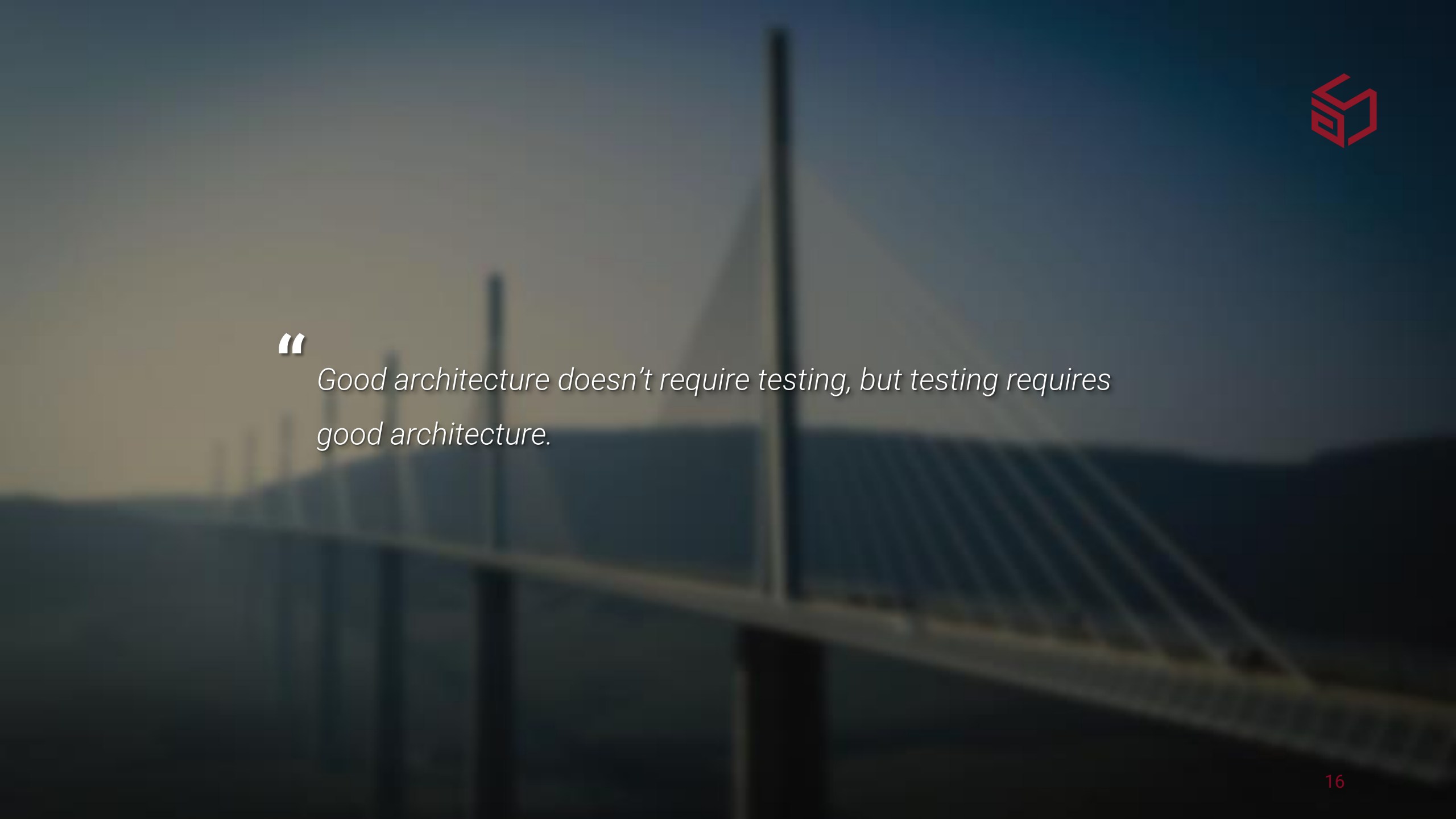
Tools & Techniques

# Unit testable architecture




- Business logic is clean to test
- Maintainable source code
  - Low coupling, high cohesion
- We can apply Test-First approach

How we get here? We only wanted to Unit test our Android application.



> *Good architecture doesn't require testing, but testing requires good architecture.*

# Mockito

- Create fake implementations by interfaces, classes
- Verify behavior by ensuring method calls

```
public interface PinView {
   void showPinErrorMessage(@StringRes int resourceId);
}
```

```
@Test
public void givenEmptyPin_whenWeSubmit_thenEmptyErrorMessageShouldShown…{
   // Given
   String pin = "";
   // When
   pinPresenter.submitPin(pin);
   // Then
   verify(pinView).showPinErrorMessage(R.string.pin_error_empty);
}
```

Tools & Techniques

# Robolectric


- Bundled Android SDK with “fake” implementations
- We can Unit test our application without changing the architecture
  - Is this a good thing?

Tools & Techniques

# Robolectric

## Design Smell

- We can ignore something that our standard unit tests are trying to tell us
  - We are tightly coupled to Android SDK
- Robolectric is a set of mocks for a set of types we don't own.
  - "Don't mock type you don't own!" – Mockito: How to write good tests

**If possible, we should decouple the Android behavior from our business logic instead.**

# Agenda

1

## Testing principles

- What to test?
- Testing levels
- Testing pyramid

2

## Tools & Techniques

- Hermetic testing
- DIP + DI
- Mocking

3

## JUnit 5

- JUnit 5 on Android
- JUnit 5 features

# JUnit 5 on Android

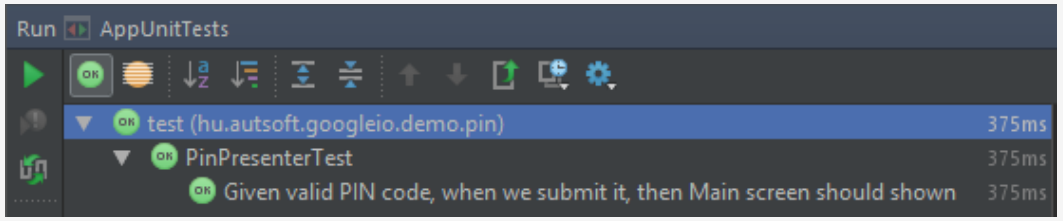
- JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage
- Supported Java 8 features
- A bit hacky on Android yet
  - JUnit 5 doesn't support the Android Gradle task flow
  - We can use android-junit5 plugin

JUnit 5

# Display Name

- No need for method naming conventions
- Test report is readable

```
@DisplayName("Given valid PIN code, when we submit it, then Main screen should shown")
@Test
public void mainScreenShouldShown() throws Exception {
    // Given
    String pin = "123456";
    // When
    pinPresenter.submitPin(pin);
    // Then
    verify(view).navigateToMainActivity();
}
```

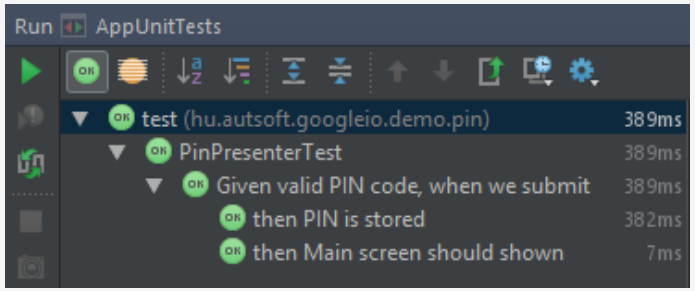


JUnit 5

# Nested tests

- Grouped tests in source and report level

```
@Nested
@DisplayName("Given valid PIN code, when we submit")
class ValidPinTests {
    @DisplayName("then Main screen should shown")
    @Test
    public void mainScreenShouldShown() throws Exception {
        …
    }
    @DisplayName("then PIN is stored")
    @Test
    public void pinIsStoredInPreferences() throws Exception {
        …
    }
}
```



# Assumptions

- We can ensure a state what is necessary to run specific tests
- If an assumption is not fulfilled: **TestAbortedException**
- Great to avoid waiting for long-running test fails

```
@Test
public void mainScreenShouldShown() throws Exception
{
    assumeTrue(pinPresenter != null);
    …
}
```

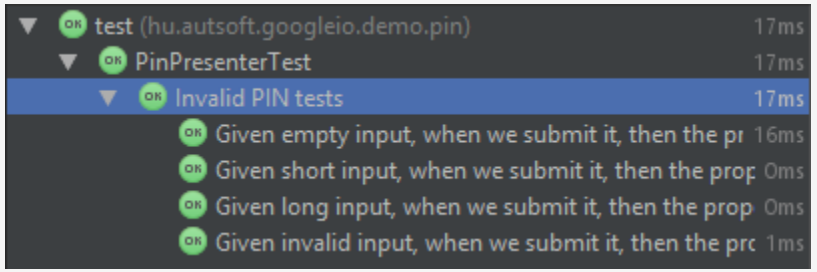
JUnit 5

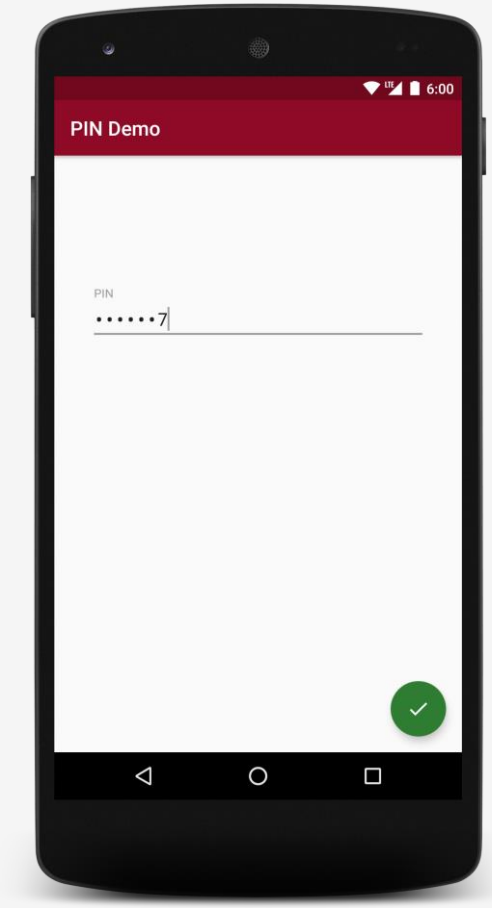
# Dynamic tests

- We can generate tests from test

```
@TestFactory
@DisplayName("Invalid PIN tests")
Collection<DynamicTest> givenInvalidPins_WhenWeSubmitThem_ThenProperErrorMessagesShouldShown() {
   Set<DynamicTest> tests = new LinkedHashSet<>();
   tests.add(createInvalidPinTest("", R.string.pin_error_empty, "empty"));
   tests.add(createInvalidPinTest("123", R.string.pin_error_length, "short"));
   tests.add(createInvalidPinTest("123455", R.string.pin_error_invalid, "long"));
   tests.add(createInvalidPinTest("1234567", R.string.pin_error_length, "invalid"));
   return tests;
}
```

```
return dynamicTest(displayName, new Executable() {
   @Override
   public void execute() throws Throwable {
      // When
      pinPresenter.submitPin(pin);
      // Then
      verify(view).showPinErrorMessage(expectedResourceId);
   }
});
```






Effective Automated Testing

# Demo project

- Master: Initial state with UI tests
- Step 1: Hermetic testing, DIP + DI
- Step 2: Decoupling Android, MVP, Unit tests
- Step 3: Upgrade to JUnit 5

https://github.com/RolandMostoha/pin-demo

AutSoft

# Thank you for your attention!

mostoha.roland@autsoft.hu
@RolandMostoha
blog.autsoft.hu