Espresso Patronum:
The Magic of the Robot Pattern

Adam McNeilly: Android Engineer - OkCupid

@AdamMc331
@NYAndroidMeetup

What is Espresso?

Use Espresso to write concise, beautiful, and reliable Android UI tests[1].

[1] https://developer.android.com/training/testing/espresso/index.html

Three Classes To Know

1. ViewMatchers
2. ViewActions
3. ViewAssertions

ViewMatchers

- `withId(...)`
- `withText(...)`
- `isFocusable()`
- `isChecked()`

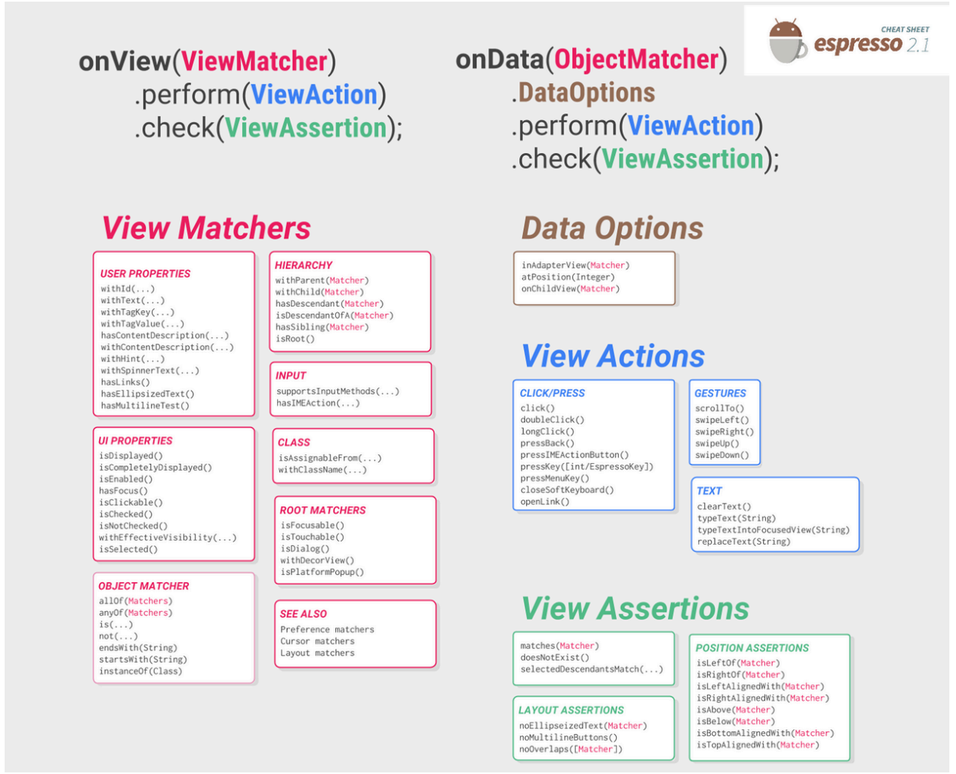
ViewActions

- `typeText(...)`
- `scrollTo()`
- `swipeLeft()`
- `click()`

ViewAssertions

- `matches(Matcher)`
- `isLeftOf(Matcher)`
- `doesNotExist()`

Espresso Cheatsheet[2]



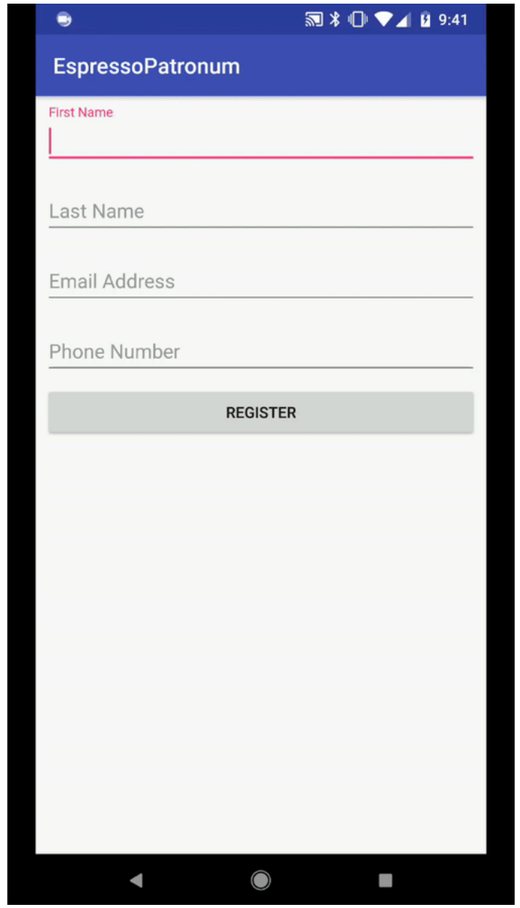
[2] https://developer.android.com/training/testing/espresso/cheat-sheet.html

Espresso Example

```
// onView gives us a ViewInteraction where we can perform an action
// or check an assertion.
onView(ViewMatcher)
    .perform(ViewAction)
    .check(ViewAssetion)

// Type into an EditText, verify it appears in a TextView
onView(withId(R.id.etInput)).perform(typeText("Adam"))
onView(withId(R.id.tvOutput)).check(matches(withText("Adam")))
```

Sample Project



Test Successful Registration

```
@Test
fun testSuccessfulRegistration() {
    onView(withId(R.id.etFirstName)).perform(typeText("Adam"))
    onView(withId(R.id.etLastName)).perform(typeText("McNeilly"))
    onView(withId(R.id.etEmail)).perform(typeText("amcneilly@okcupid.com"))
    onView(withId(R.id.etPhone)).perform(typeText("1234567890"))
    onView(withId(R.id.registerButton)).perform(click())

    onView(withId(R.id.tvFullName)).check(matches(withText("Adam McNeilly")))
    onView(withId(R.id.tvEmailAddress)).check(matches(withText("amcneilly@okcupid.com")))
    onView(withId(R.id.tvPhoneNumber)).check(matches(withText("(123)-456-7890")))
}
```

Test A Missing Field

```
@Test
fun testMissingEmailError() {
    onView(withId(R.id.etFirstName)).perform(typeText("Adam"))
    onView(withId(R.id.etLastName)).perform(typeText("McNeilly"))
    // onView(withId(R.id.etEmail)).perform(typeText("amcneilly@okcupid.com"))
    onView(withId(R.id.etPhone)).perform(typeText("1234567890"))
    onView(withId(R.id.registerButton)).perform(click())

    onView(withId(R.id.etEmail)).check(matches(hasErrorText("Must enter an email address.")))
}
```

One More Negative Test

```
@Test
fun testInvalidEmailError() {
    onView(withId(R.id.etFirstName)).perform(typeText("Adam"))
    onView(withId(R.id.etLastName)).perform(typeText("McNeilly"))
    onView(withId(R.id.etEmail)).perform(typeText("blahblah"))
    onView(withId(R.id.etPhone)).perform(typeText("1234567890"))
    onView(withId(R.id.registerButton)).perform(click())

    onView(withId(R.id.etEmail)).check(matches(hasErrorText("Must enter a valid email address.")))
}
```

All Together

```
@Test
fun testSuccessfulRegistration() {
    onView(withId(R.id.etFirstName)).perform(typeText("Adam"))
    onView(withId(R.id.etLastName)).perform(typeText("McNeilly"))
    onView(withId(R.id.etEmail)).perform(typeText("amcneilly@okcupid.com"))
    onView(withId(R.id.etPhone)).perform(typeText("1234567890"))
    onView(withId(R.id.registerButton)).perform(click())

    onView(withId(R.id.tvFullName)).check(matches(withText("Adam McNeilly")))
    onView(withId(R.id.tvEmailAddress)).check(matches(withText("amcneilly@okcupid.com")))
    onView(withId(R.id.tvPhoneNumber)).check(matches(withText("(123)-456-7890")))
}

@Test
fun testMissingEmailError() {
    onView(withId(R.id.etFirstName)).perform(typeText("Adam"))
    onView(withId(R.id.etLastName)).perform(typeText("McNeilly"))
    onView(withId(R.id.etPhone)).perform(typeText("1234567890"))
    onView(withId(R.id.registerButton)).perform(click())

    onView(withId(R.id.etEmail)).check(matches(hasErrorText("Must enter an email address.")))
}

@Test
fun testInvalidEmailError() {
    onView(withId(R.id.etFirstName)).perform(typeText("Adam"))
    onView(withId(R.id.etLastName)).perform(typeText("McNeilly"))
    onView(withId(R.id.etEmail)).perform(typeText("blahblah"))
    onView(withId(R.id.etPhone)).perform(typeText("1234567890"))
    onView(withId(R.id.registerButton)).perform(click())

    onView(withId(R.id.etEmail)).check(matches(hasErrorText("Must enter a valid email address.")))
}
```

Downfalls Of This Approach

1. Extremely Verbose
2. Unreadable
3. Not Easily Maintainable - What if a view changes?

Introducing Robots

A robot is the middle man between your view and your code. This is a way of separating concerns just like an MVC/MVP/MVWTF architecture does with your application's code.

Usage

```
@Test
fun testSuccessfulRegistration() {
    RegistrationRobot()
            .firstName("Adam")
            .lastName("McNeilly")
            .email("amcneilly@okcupid.com")
            .phone("1234567890")
            .register()
            .assertFullNameDisplay("Adam McNeilly")
            .assertEmailDisplay("amcneilly@okcupid.com")
            .assertPhoneDisplay("(123)-456-7890")
}
```

Define ViewMatchers

```
class RegistrationRobot {

    companion object {
        private val FIRST_NAME_INPUT_MATCHER = withId(R.id.etFirstName)
        private val LAST_NAME_INPUT_MATCHER = withId(R.id.etLastName)
        private val EMAIL_INPUT_MATCHER = withId(R.id.etEmail)
        private val PHONE_INPUT_MATCHER = withId(R.id.etPhone)
        private val REGISTER_INPUT_MATCHER = withId(R.id.registerButton)

        private val FULL_NAME_DISPLAY_MATCHER = withId(R.id.tvFullName)
        private val EMAIL_DISPLAY_MATCHER = withId(R.id.tvEmailAddress)
        private val PHONE_DISPLAY_MATCHER = withId(R.id.tvPhoneNumber)
    }
}
```

Each Action As A Method

```
class RegistrationRobot {

    fun firstName(firstName: String): RegistrationRobot {
        onView(FIRST_NAME_MATCHER).perform(clearText(), typeText(firstName), closeSoftKeyboard())
        return this
    }

    fun register(): RegistrationRobot {
        onView(REGISTER_INPUT_MATCHER).perform(click())
        return this
    }

    fun assertFullNameDisplay(fullName: String): RegistrationRobot {
        onView(FULL_NAME_DISPLAY_MATCHER).check(matches(withText(fullName)))
        return this
    }

    ...
}
```

Benefits

1. Readibility
2. Maintainability

What Else?

Better Test Reporting

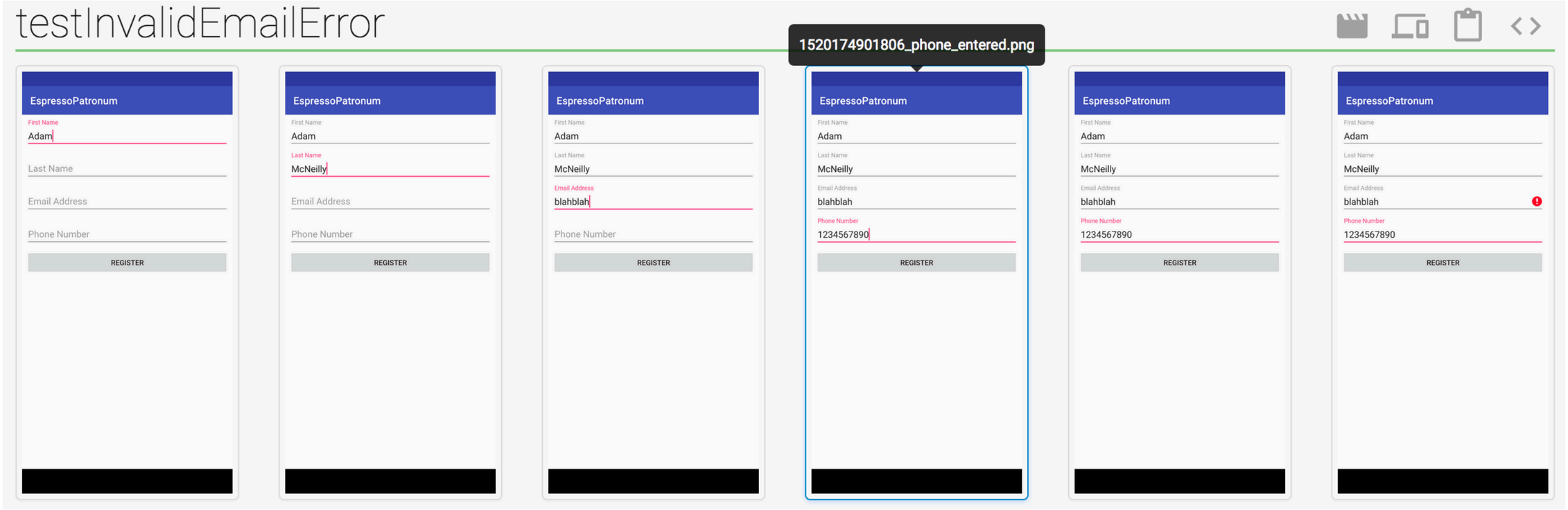
Now that we've established the robot pattern for clean and maintainable tests, let's take it a step further.

Better Test Reporting Using Spoon[3]

Spoon will run all of our instrumentation tests and build us a static HTML report at the end.

[3] https://github.com/square/spoon

Example Spoon Report



When To Take Screenshots

- After assertions
- After actions - unless that action leads to another screen
- On failure

Adding Screenshots To Our Robot

```
fun firstName(firstName: String): RegistrationRobot {
    onView(FIRST_NAME_INPUT_MATCHER).perform(clearText(), typeText(firstName), closeSoftKeyboard())
    takeScreenshot(spoon, "first_name_entered")
    return this
}

fun register(): RegistrationRobot {
    takeScreenshot(spoon, "register_clicked")
    onView(REGISTER_INPUT_MATCHER).perform(click())
    return this
}

fun setFailureHandler(spoon: SpoonRule, context: Context) {
    Espresso.setFailureHandler { error, viewMatcher ->
        takeScreenshot(spoon, "test_failed")
        DefaultFailureHandler(context).handle(error, viewMatcher)
    }
}
```

Why screenshots?

— Human readable output

— Shows exactly what was tested and how

— Diagnose failures faster

Takeaways

1. Use the robot pattern to make your tests more maintainable.
2. Your actual tests become easier and quicker to write once you've created a robot.
3. Robots can be leveraged for additional and more thorough reporting.
4. This idea is not specific to Espresso or Spoon.

Contact

— Twitter - @AdamMc331

— https://github.com/AdamMc331/EspressoPatronum