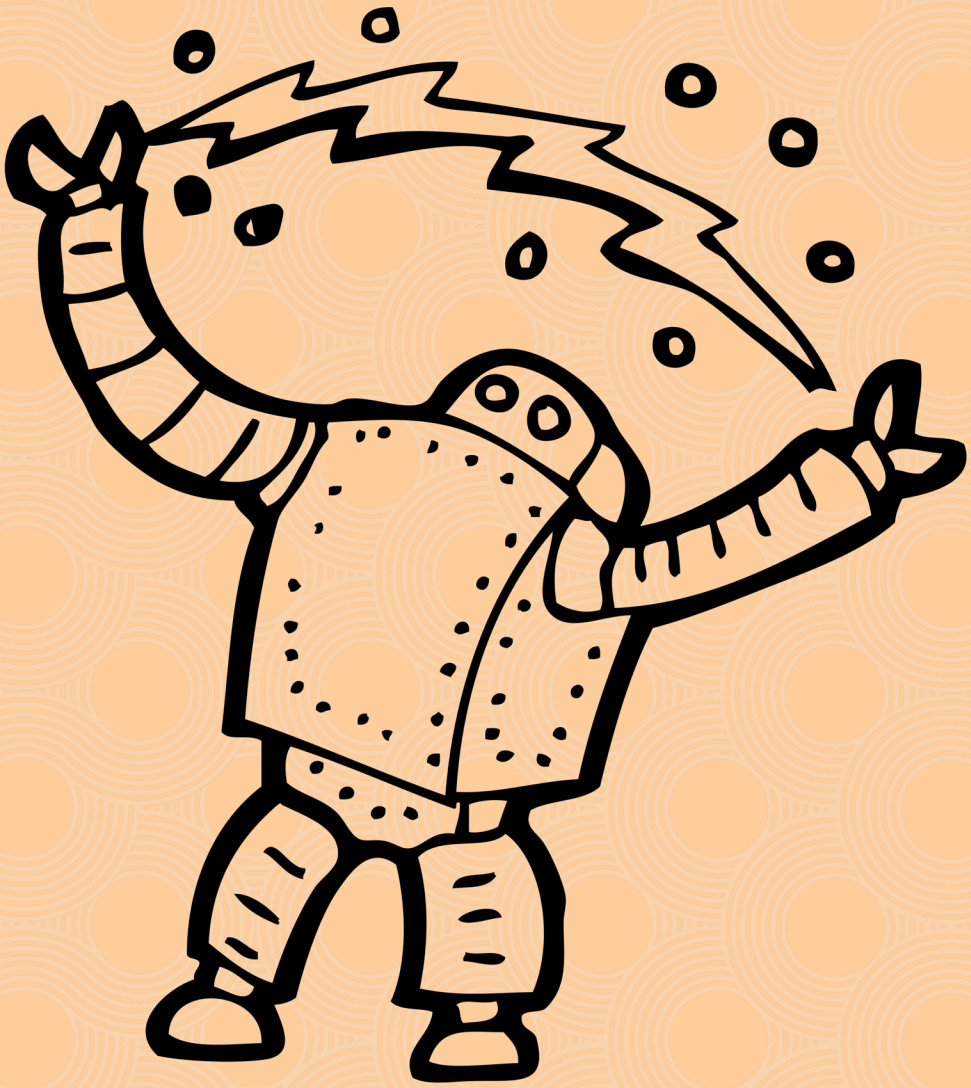


TDD for Android

Time-tested tricks to do TDD well

By @xpmatteo and @carlobellettini

TDD for Android

Time-tested tricks to do Test-Driven Development well

Matteo Vaccari and Carlo Bellettini

Contents

1. Introduction

What will we learn?

- Driving development from acceptance tests
- Writing end-to-end tests
- Driving development from true unit tests
- How to keep the pure logic of the application separate from Android APIs (aka the ports-and-adapters architecture aka hexagonal architecture)
- How to use mocks properly (aka the London Style)
- Presenter-first development
- What is the "main partition" and why it's useful
- A bit of object-oriented design

We will not discuss continuous integration or version control.

Our philosophy

This will not be a "stream-of-consciousness". We will present the samples of code as if we got them right the first time. We didn't, but we think that reading a detailed account of our mistakes and false starts would be tedious. We show you instructive examples, presented in a logical flow, so that you get our idea of what good code looks like. Just don't worry if you can't get your code right the first time like it seems we are doing. We don't usually get it right the first time either!

What this book is not about

We will not provide extensive information on how to test your application on different versions of Android, different devices or different form factors. This is a book about software design, not testing. We believe that TDD is both a design method and a way to write correct code; however in this book we want to emphasize design.

Assumptions

- You know Java
- You have built and run an Android application before
- You have a basic understanding of what testing is and how JUnit works
- You use Android Studio

2. Why doing TDD with Android is challenging

Android problem #1: the OS does the instantiating

The usual way to write a unit test for a Java object is to

1. create the object, bringing it to a desired state
2. perform an action on it
3. check the results

For instance, suppose we wanted to test a simple "dictionary" object. We could do it by means of this test:

```
// 1. create the object and bring it to a desired state
Dictionary dictionary = new Dictionary();
dictionary.define("cat", "gatto");
dictionary.define("dog", "cane");

// 2. perform an action
String translation = dictionary.translationFor("cat");

// 3. check the results
assertEquals("gatto", translation);
```

The problem for Android programmers is that in Android, all of the important objects: the Activity, the Views, the Services, the Content providers, are instantiated by the system. This makes it very difficult to do a unit test on such objects, because we cannot even instantiate the objects; we must get the OS to instantiate them for us.

The solution: don't try to unit test them. Regard them as a kind of "main" program.

A non-Android example

Let's illustrate the principle with an example. You have a simple Java program that reads a text file from standard input, and outputs the same file with line numbers in front. The program, being so simple, can be written as a single "main" program, like this:

```
public class LineNumbersFilter {
  public static void main(String ... args) throws IOException {
    String line;
    int count = 0;
    BufferedReader reader = new BufferedReader(
        new InputStreamReader(System.in));
    while ((line = reader.readLine()) != null) {
      count++;
      System.out.println(String.format("%d %s", count, line));
    }
  }
}
```

It's extremely difficult to unit test this program. The difficulty lies in the fact that the program accesses `System.in` and `System.out` directly. It's difficult to set up the contents of System.in within a unit test, and it's difficult to check the contents of System.out in a unit test. You might try, but it's going to be messy, and very *expensive* in terms of your time.

On the other hand, if you break down the program in two parts, the "main" part and the "logic" part, you will find that the logic can be unit tested easily.

```
class LineNumbersFilter {
  private InputStream in;
  private PrintStream out;
  public LineNumbersFilter(InputStream in, PrintStream out) {
    this.in = in;
    this.out = out;
  }
  public void apply() throws IOException {
    String line;
    int count = 0;
    BufferedReader reader = new BufferedReader(new InputStreamReader(in));
    while ((line = reader.readLine()) != null) {
      count++;
      out.println(String.format("%d %s", count, line));
    }
  }
  // The "logic" part is above

  // The "main"  part is below
  public static void main(String ... args) throws IOException {
    LineNumbersFilter filter = new LineNumbersFilter(System.in, System.out);
    filter.apply();
  }
}
```

Now we can set up the LineNumbersFilter with in-memory streams, that can be easily constructed and checked within a unit test.

```
@Test
public void oneLineOfInput() throws Exception {
  ByteArrayInputStream inputStream =
        new ByteArrayInputStream("one line".getBytes());
  ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
  PrintStream printStream = new PrintStream(outputStream);

  new LineNumbersFilter(inputStream, printStream).apply();

  assertEquals("1 one line\n", outputStream.toString());
}
```

You may object: "you are saying that there is no way to test main, and we risk letting defects seep into our main". We have two answers for this.

The first answer is that if we properly separate logic from creation, there will be no "IFs" in the main. So it will be easy to check by hand that it works; a single manual test will do it. In other words: if there is an error in the main, it will be readily apparent as soon as we try to run it.

The second answer is that it's actually quite easy to do an **end-to-end** test of the main. The Bash script below does it quite nicely.

```
cat > /tmp/expected.txt <<EOF
1 first line
2 second line
EOF

java LineNumbersFilter > /tmp/actual.txt <<EOF
first line
second line
EOF

if diff /tmp/expected.txt /tmp/actual.txt > /tmp/difference.txt
then
  echo "ok"
else
  echo "ERROR:"; cat /tmp/difference.txt
fi
```

This, however, is not a unit test. It does not just test "main"; it tests main and everything else. It can be a very good thing to write such an end-to-end test. However, such tests usually take a long time to write, and it usually takes a long time to get them to pass. For this reason, although they can be very useful in general, they are not very useful for TDD.

What's the consequence for Android programmers

In Android, we can't just instantiate Activities or Views. We must let the system instantiate them for us. Therefore it's not practical to unit-test them. So we don't! We treat the Activity, View, Service or Content Provider as **our "main"**. We take care to move all the logic away from them, into objects that we can create as we please.

Android problem #2: all tests run on the device

The second big problem is that all the tooling that goes with Android assumes that the JUnit tests will run *inside the device*. We cannot even assert that `3+4` is `7` without downloading the test code to the device and executing it there.

But the problem is even worse than that. The "unit testing" toolkit that Google provides is a mess. It's poorly documented: the examples were written for who knows which old version of the OS and they don't always work. Some kind of tests work reasonably well; in our experience, tests for simple forms with input and output text fields tend to work. Other things are absolutely difficult to test; if you want a challenge, try writing a test that starts more than one activity.

Even if you get the test to work at all, it's going to be very slow to run. It will take about `30` seconds to download the code to the device. This is too much.

Additional toolkits of various kinds, such as Robolectric, Espresso or Robotium try to ameliorate the situation, but fail. They are all *extremely* difficult to setup. The official instructions are very long to start with and they are out-of-date most of the time. Then you must follow a trail of blog posts hoping to find up-to date information or hacks. These tools are a major time sink and we recommend that TDDers avoid them. They might be useful for *testers*, but they are not so useful for *programmers*.

Our solution is to avoid running unit tests in the device. Move all the logic from the android-dependent module of your project to a new, android-independent module that contains pure Java logic. This the same "no logic in main" technique we saw in the previous section.

Our job is to build solutions for our customers in reasonable time. We cannot afford to spend major time nursing the test environments. This would be *pure waste*. Not only that; these tools push us to work at the lowest level of abstraction, that is, the level of the Android API. A much more productive use of our time is to reason on how to separate the logic of our application from the Android programming environment. This will push us to find a more abstract level of APIs that will be more productive to us.

We still recommend to use tools like Monkey Runner or Robotium for end-to-end testing. Just don't use them for TDD.

3. The Android Development Process

Here we describe in general terms the development process that we propose. We start with a high-level description, then we follow with a more detailed description of the various parts of the process. Many of the things that we talk about have been described at length in other books; for instance, TDD or User Stories. Yet there are many different ways to see these subjects. In our work experience, it always pays to recap the fundamentals. For it usually happens that the people in the room have different ideas of what TDD or User Stories are. So bear with us, so that when we talk of TDD, or other fundamental parts of the process, we are all on the same page.

At a glance

You have an idea for a great app. What steps do you take to make it real? The framework that we describe will orient your actions.

Planning

1. Break the project in user stories
2. Select the first user story you want to deliver
3. Write concrete *examples* of how the user story will work.

Execution

1. [Optional] Start with a Spike
2. Create the project
3. [Optional] Automate an example with an End-to-end Acceptance Test
4. Implement the example, in TDD, usually using the *presenter-first* technique
5. Connect the presenter with the view
6. Test the example manually on the device
7. If you have an end-to-end test, check that it works.
8. Choose the next example and go to 4.

This is a rough outline. It will rarely be this linear in practice; for instance it will happen that when you test the application manually, you will notice things that you want to change. You will add them to your todo-list.

The loop between steps 4 and 8 should be executed in minutes, not hours… don't spend too much time without checking that your code works on the device. Conversely, don't rely only on the device for getting feedback on your code.

Break down a project in User Stories

Do we need a plan? Yes we do. Without a plan, we risk to lose *focus*. The app we have in mind will need many features and details. If we work on more than one thing at once, we disperse our time and energy. So the main reason for planning is that we want to focus on a single thing at a time. Focus and prioritize:

1. Make a list of the things you want to do
2. Choose the first thing you want to do
3. Do that thing and nothing else until it's done.

This is the Agile way to work. And one more thing: make it so that the things you work on are so *small* that you can build them in hours. This way you will produce a steady stream of small successes.

So, how do we produce a list of things to do? We write user stories.

A user story is simply a brief description of something that a user can do with the application. A user story has:

1. A Title
2. [Optional] a Description
3. [Eventually] an Acceptance Criterion

When you start thinking about the user stories for a project, you will probably start with just a bunch of titles. Some stories can be further clarified with a brief description. Before you implement them, you will need to clarify the Acceptance Criterion. An Acceptance Criterion can be further clarified with Examples.

Example: a Todo-List application

If I wanted to implement a Todo-List app, I would probably want the following user stories:

- Create new lists
- Show the items of a list
- Add an item to a list
- Mark an item as done

This is a good set of stories to start. As we keep thinking, many more stories will come up, related to renaming lists and items, deleting, sharing, notification, etc.

Exercise: write the list of user stories for an app that reads blog feeds.

Rules for user stories

- Demoable: when the story is done, there should be something *visible* to prove that it's done.
- Small: it should be possible to implement a story in hours, or at most in a day or two. Break down big stories in small ones.
- Independent: it's usually possible to build user stories in any order. For instance: you can demo the "Show the items of a list" user story before the "Add an item to a list" story: just show a few canned items that you instantiate inside the application code.

Concrete examples

An Acceptance Criterion can be a general statement of what the application should do. For instance, the AC for the user story "Hide items when done" could read like

```
When an item is marked as 'done' it will be hidden after one week.
```

To make sure that we understand correctly what this means, we translate it to concrete Examples. Usually we do this through a conversation with our customer.

> Given Item "Buy milk" that was marked "done" on 1st March 2015 at 14:00:00 \
>
> **Example**: **item not yet hidden**
> When the current date and time is 8th March 2015 at 13:59:59
> Then the item is shown \
>
> **Example**: **item hidden** When the current date and time is 8th March 2015 at 14:00:00 Then the item is NOT shown

As you can see, where the AC talks in general terms "An item marked as *done*", the examples are about concrete things: a specific item "Buy Milk" that was marked done in some specific instant in time.

Writing the examples is an useful thing, because

1. They remove ambiguity: what does "after one week" means exactly? Now we know: it's not just the calendar day, it's also the time of day that matters.
2. The examples become tests: we can execute them both manually (for instance, by artificially changing the clock of the device) or automatically.

Another name for Examples is Scenarios.

The concept of Example/Scenario is crucial, because it drives development. A scenario is the smallest unit of functionality that can be delivered.

Start with a Spike

What is a spike? A *spike* is an experiment that you do in order to explore how to do a feature. A spike will usually not have tests, will be quick and dirty, will not be complete, will not follow our usual rules for good quality. It's just an exploration.

Spikes are important because we do often have to work with APIs that we don't know well. This is particularly true in the Android environment, where we have to deal with complex, rich components whose behaviour is not clear until we start playing with them.

Important point: don't try to write TDD with APIs that you don't know well. There is the risk of wasting a lot of time writing ineffective tests.

So, whenever we start working with a bit of Android APIs that we don't know well, it pays to start with a little experiment.

The rules for spikes are:

1. Goal: you have a learning goal for the spike
2. Timebox: you set yourself a time limit; for instance, two hours.
3. New project: start the spike in a new project (not by hacking into your production code)
4. Throw away: after you're done, you *throw away* the spike code. You may keep the spike around as a *junkyard* of bits to copy from; but you never turn the spike into your production project. Start production code with a fresh project.

Test-Driven Development

TDD is about writing code in *small increments*, driven by an *automated test*, keeping the code *as simple as possible* at all times. This process has a dual nature: it is a way to *design* programs, and also is a way to build a good *suite of tests*.

The process was described first by Kent Beck in Test-Driven Development: By Example. It's probably still the best book on the subject. It's a subtle book: it seems like it's glossing over many things, but when you go back to it and read it again, you find it has new answers to your questions.

The micro-cycle

The process is easy to describe. Kent Beck in Test-Driven Development: By Example writes the following sequence:

1. Quickly write a test
2. Run all the tests, see the new one fail
3. Make a small change
4. Run all the tests, see them all succeed
5. Refactor to remove duplication

Note the emphasis on "small": the cycle is meant to be repeated every few minutes, not hours. The point of TDD is to provide quick feedback on the quality of your code.

If you've never seen an accomplished TDD practitioner at work, it's difficult to grasp how the TDD cycle should be done. We suggest you to watch good TDD videos, like the TDD Videos by Kent Beck

Presenter First

How do you TDD a GUI application?

The Presenter First style of TDD is usually appropriate when we write GUI applications. The idea is that you start by postulating that you will have at least two objects: a *Presenter* and a *View*. The *Presenter* is an object that represents your whole application, or a significant subset of your application. The *View* represent the GUI screen that your application will use.

The "View" ambiguity

The word "view" in the Android world has a specific meaning, but when we talk of presenters and views, we mean something different. A view in Android is a subclass of class `View`. In the context of presenters instead, a "view" is a role that is played sometimes by Android activities, sometimes by actual Android views. We trust that context will make it clear what we mean in each case.

One key idea is that all application logic goes in the Presenter, while all technical details of how to show windows etc to the user go in the view. The presenter is notified by the view that something happened. In response, the presenter calls methods on the view to change what the user sees.

Another key idea is that you start TDD with the presenter.

J.B. Rainsberger popularized the concept in his video The World's Best Intro To TDD and in his book Responsible Design For Android

The presenter-first method works by defining (at least) two objects:

When we describe what a GUI application does, we usually reason in terms of "when the user does THIS, then the application shows THAT". For instance, consider an application that shows a counter that can be incremented by the user by pressing a button. Our Example will say:

> **Example**: **increment** Given that the current counter value is 0 When the user presses the "INC" button Then the view will show 1

The GUI of the Counter application

First version: not yet quite a presenter

A simple test for this example could be

```
CounterApp app = new CounterApp(0);
app.increment();
assertEquals(1, app.valueToDisplay());
```

This test is adequate. You can use it to develop your application. It's easy to connect this `CounterApplication` logic to a GUI written in Android. In the following code, we show how to use the `CounterApp` in an activity:

```
1  public class CounterActivity extends Activity {
2    private CounterApp app = new CounterApp();
3
4    @Override
5    protected void onCreate(Bundle savedInstanceState) {
6      super.onCreate(savedInstanceState);
7      setContentView(R.layout.activity_counter);
8
9      Button incrementButton = (Button) findViewById(R.id.increment);
10     incrementButton.setOnClickListener(new View.OnClickListener() {
11       @Override
12       public void onClick(View v) {
13         app.increment();
14         TextView textView = (TextView) findViewById(R.id.counter);
15         textView.setText(String.valueOf(app.valueToDisplay()));
```

```
16        }
17      });
18    }
19  }
```

In lines 9-10 we make sure that when the user clicks the "increment" button, Android will call us back. In line 13 we increment the counter, and in lines 14-15 we update the text label.

A proper presenter

The previous example works and is adequate for most purposes; yet it's not completely satisfactory. We need to write many lines of "glue" to make sure that we are really getting the right data from the CounterApp and using correctly in the activity. We are calling the CounterApp twice, one for communicating the fact that the user has pressed the button, and another time for asking back the CounterApp for the value to display.

This is adequate for such a small application, but becomes boring when the number of bits of user interface that *could* change increases. If we had 100 text fields on the GUI, we wouldn't like to ask 100 questions to the app so that we can update them all.

```
// Something we would definitely NOT want to do
public void onClick(View v) {
  app.doSomething();
  // ask the app the value of ALL fields in case one of them
  // is changed...
  findViewById(R.id.display0).setText(app.valueToDisplay0());
  findViewById(R.id.display1).setText(app.valueToDisplay1());
  findViewById(R.id.display2).setText(app.valueToDisplay2());
  findViewById(R.id.display3).setText(app.valueToDisplay3());
  findViewById(R.id.display4).setText(app.valueToDisplay4());
  findViewById(R.id.display5).setText(app.valueToDisplay5());
  // etc etc ...
}
```

It would be much better if the app could update directly the fields it wishes to change. What if it was simply like this:

```
public class CounterActivity extends Activity {
  private CounterApp app = new CounterApp(this);

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_counter);

    View incrementButton = findViewById(R.id.increment);
    incrementButton.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        // Just notify the app, it will do the rest
        app.increment();
      }
    });
  }
}
```

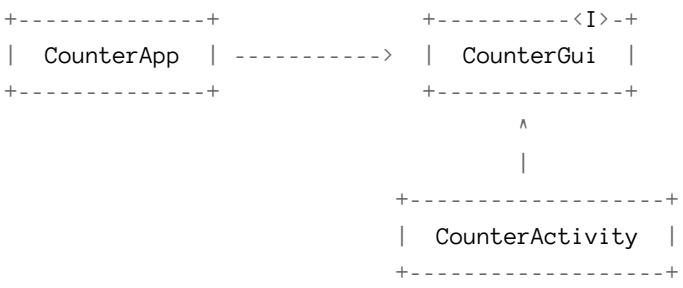
The activity is vastly simplified: all we have to do is call `app.increment()` whenever the user clicks the button.

```
public class CounterApp {
  private int value;
  private CounterActivity counterActivity;

  public CounterApp(CounterActivity counterActivity) {
    this.counterActivity = counterActivity;
  }

  // We're not quite there yet!
  // We shouldn't mess with the internals of the activity!
  public void increment() {
    value++;
    TextView view = (TextView) counterActivity.findViewById(R.id.counter);
    view.setText(String.valueOf(value));
  }
}
```

The changing of the value displayed on the GUI has become a responsibility of the CounterApp. We don't really like this; we'd like to be able to tell the activity "show this value!" and let the activity deal with the details of which element of the view to update.

We have created a circular dependency between the activity and the app

And this is bad. We would much prefer that the CounterApp be independent of the CounterActivity. Luckily, there is a standard way to break circular dependencies: introduce an interface!



We introduce an interface we call `CounterGui` (We could have chosen the name `CounterView`, but that could create confusion with the way Android uses the word "view")

So let's start again with the presenter. We write a test of what the CounterApp should do when the increment method is called. The test should do, in pseudo code:

```
Given the CounterApp holds a reference of the CounterGui as a collaborator
when we call increment on the CounterApp
then the CounterGui receives a call to display "1"
```

We want to test CounterApp without ever referring to the CounterActivity, that in our intentions will be the one real implementation of CounterGui. Therefore, we need a fake implementation of CounterGui. Our fake implementation will be passed as a collaborator to the CounerApp, and it does not need to implement a real GUI; all we need is that we can check that the proper call(s) to it have been made. This type of fake implementation of an interface is called a "Mock".

Definition: a "mock" is a fake implementation of an interface that can be used to verify that certain calls have been made to it.

There are more than one way to write this mock. The simplest way needs no particular mocking frameworks. Just let the test class implement the interface that we want to use.

```
// Our test class implements CounterGui
public class CounterAppTest implements CounterGui {

  @Test
  public void increment() throws Exception {
    // We create the app, passing the test class itself as a collaborator
    CounterApp app = new CounterApp(this);

    // Whenever we call
    app.increment();

    // We expect display(1) to have been called
    assertEquals(Integer.valueOf(1), displayedNumber);
  }

  // we use this variable to check that CounterGui#display(1) was called
  Integer displayedNumber = null;

  // and this is our fake implementation of CounterGui#display
  @Override
  public void display(int number) {
    this.displayedNumber = number;
  }
}
```

Note that writing this test forces us to define the one method that the CounterGui needs to have, namely `display`. We use a trick to verify that the display method has really been called. If it is not called, the value of `displayedNumber` remains null. If it is called, the value of `displayedNumber` is the value of the argument to the call.

This is enough to allow us to see the test fail, and then build the right functionality within CounterApp to make it pass. It's easily done:

```
public class CounterApp {
  private int value;
  private CounterGui gui;

  public CounterApp(CounterGui gui) {
    this.gui = gui;
  }

  public void increment() {
    value++;
    gui.display(value);
  }
}
```

Note, however, that if CounterApp made more than one call to CounterGui, we would not be able to detect this. We only retain the argument of the last call.

Using a mocking framework such as JMock or EasyMock solves this problem. It also makes it easier to specify precisely what we expect: "just ONE call to CounterGui#display with the argument 1". The price we pay is that we need to use more sophisticated machinery (see chapter How JMock Works).

The following is the same test, implemented with JMock.

```
public class CounterAppTest {

  // This is the JMock machinery that we need
  @Rule public JUnitRuleMockery context = new JUnitRuleMockery();

  public void testIncrement() throws Exception {
    // We ask JMock to make a mock of the CounterGui interface
    final CounterGui gui = context.mock(CounterGui.class);

    // We create the app, passing the gui as a collaborator
    CounterApp app = new CounterApp(gui);

    // We setup our expectations
    context.checking(new Expectations() {{
      // Exactly one time, gui will be called with display(1)
      oneOf(gui).display(1);
    }});

    // Whenever we call
    app.increment();
  }
}
```

The amazing thing is that the presenter test, whether written with hand-made mocks or with a mocking framework, allows us to define the `CounterGui` interface, that is to find all the methods that we need from this interface, even before that an implementation exists! Here is the interface that emerged:

```
public interface CounterGui {
  void display(int number);
}
```

Our next task is to define a real implementation of `CounterGui`. The obvious choice here is to let `CounterActivity` implement it. We do it here:

```
public class CounterActivity extends Activity implements CounterGui {
  private CounterApp app = new CounterApp(this);

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_counter);

    View incrementButton = findViewById(R.id.increment);
    incrementButton.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        app.increment();
      }
    });
  }

  @Override
  public void display(int number) {
    TextView textView = (TextView) findViewById(R.id.counter);
    textView.setText(String.valueOf(number));
  }
}
```

Notice the nice separation of concerns: the presenter deals with the application logic; the view deals with presentation logic.

Model, View, Presenter

A Presenter usually deals with a view, as we have seen, and a *model*. What's a model? A "model" in this context is a model of the problem domain. In other words, it's a pure logic implementation of the application logic, independent of any infrastructural issues such as GUIs or persistence.

Wait, didn't we say the same thing about the presenter, just a few pages ago? Well, sort of. You see, the Presenter that we have built so far really has two responsibilities:

1. Implement application logic (in this case, it's the simple counter logic of incrementing the value)
2. Connecting application logic with the view

You see it in the implementation of method `CounterApp#increment`:

```
public void increment() {
  value++;
  gui.display(value);
}
```

It is doing two things: incrementing the counter and updating the view. In this particular case, the application logic is so simple that it might make sense to leave it like this. In general, however, application logic is complex. For this reason, it's usually a good idea to separate the model from the presenter. If we wanted to do so, we would create a `Counter` object that does nothing but counting, and pass it to the presenter as a collaborator.

The result would be a clearer separation of concerns:

```
public class CounterApp {
  private Counter counter;
  private CounterGui gui;

  public CounterApp(Counter counter, CounterGui gui) {
    this.counter = counter;
    this.gui = gui;
  }

  public void increment() {
    counter.increment();
    gui.display(counter.value());
  }
}
```

The obvious implementation of `Counter` would be

```
public class Counter {
  int value = 0;

  public void increment() {
    value++;
  }

  public int value() {
    return value;
  }
}
```

Acceptance-TDD (ATDD)

One pattern that is mentioned in the original TDD book by Kent Beck is "Child Test". It means that when it takes me too long to get a test to pass, it probably means that the test represents too big a step for me. In that

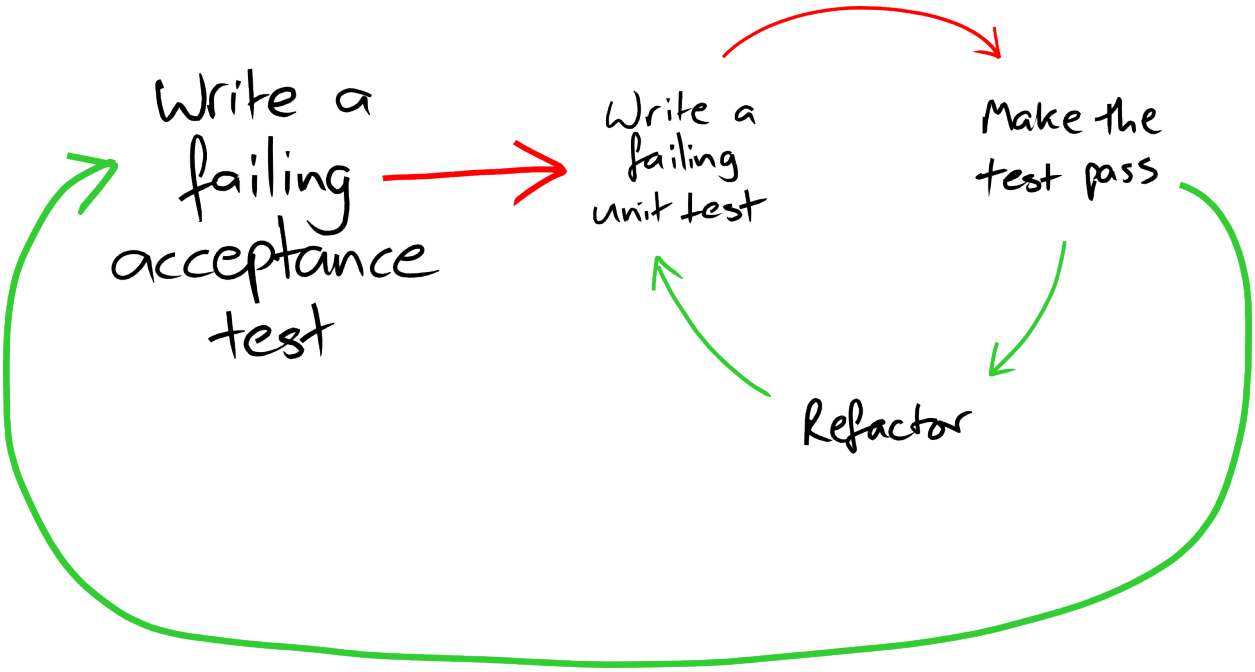
case, Beck suggests that you comment out the problematic test, and start writing a simpler test, a "child test". The child test lets me take a smaller step in the direction of making the larger test pass. After a while, and probably after several TDD cycles, we are ready to make the big test pass.

One instance of the Child Test pattern is when we write an end-to-end acceptance test as a first step in getting a new feature to work. That test is likely to take more than a few minutes to pass. In Android, an end-to-end test will also take minutes to run. Yet, the end-to-end test is very useful.

1. It makes me think hard about *examples* of the desired functionality
2. It makes sure that the configurations are correct and that all the objects inside the application talk to each other correctly

For this reason, it pays to apply Child Test to end-to-end tests. We start every feature by writing on paper some examples (scenarios) of how the feature will work. Then we translate those scenarios into end-to-end acceptance tests. When we are satisfied that they fail, and that they fail for the correct reason, then we comment them out.

This style of work is sometimes called "Acceptance Test-Driven Development" or ATDD. It was popularized by Freeman and Pryce in the book Growing Object-Oriented Software. Below you can see the picture from their book:



The ATDD cycle; image cc-by-sa courtesy of Freeman and Pryce

When we do TDD we often exercise objects in isolation from each other. The mocks approach explained in GOOS is particularly good in this respect. The end-to-end acceptance test (AT) helps making sure all the objects that we TDDed in isolation talk to each other correctly. Thus we mitigate the risk of mock-based tests making false assumptions on how the real (non-mocked) collaborators really work.

Mock objects

Quick, what are mock objects good for? If you answered "for isolating dependencies" then Bzzzzzzt! You got it wrong!

The real reason why mocks are useful is that they help us developing *protocols*, that is a set of messages that an object must understand in order to fulfill a *role* in an object-oriented system.

Tell, don't ask

An object works by sending and receiving *messages*. When an object receives a message, it can react by sending messages to other objects. How do we test an object then? The simple way is to send a message to an object and then use *getters* to access the object internal state. One problem with this is that the getters will force us to expose at least part of the object's internal representation. This will make it harder to change the object.

Not only that. Using getters will push us to separate data and behaviour. Consider

```
if (point.isPolar()) {
  x = point.getRadius() * Math.cos(point.getAngle());
  y = point.getRadius() * Math.sin(point.getAngle());
} else {
  x = point.getX();
  y = point.getY();
}
canvas.drawPoint(x, y);
```

We ask the point a question: *are you represented with polar coordinates?* Then depending on the question we do one thing or another. What we have is that the point is a dumb data structure with no behaviour. The code that uses the point is doing all the reasoning and has all the behaviour; it converts from polar coordinates to cartesian if necessary, then draws the point on a canvas.

Why do we open up the object and see its internals? We don't have to! We could ask the point to give us the cartesian coordinates, doing the conversion internally if necessary. The above code becomes

```
x = point.getX();
y = point.getY();
canvas.drawPoint(x, y);
```

But we can go one step further. What if the code was

```
if (!point.isInvisible()) {
  canvas.drawPoint(point.getX(), point.getY());
}
```

Why do we have to ask all this questions to the point object? Couldn't we just ask it to draw itself unless it's invisible?

```
point.drawYourselfUnlessInvisibleOn(canvas);
```

Now the calling code is much simpler! And it's a lot easier to maintain. Suppose we add a *color* to the point. The old calling code would have to change to

```
if (!point.isInvisible()) {
  canvas.drawPoint(point.getX(), point.getY(), point.getColor());
}
```

but the new calling code does not change much

```
point.drawYourselfUnlessInvisibleWithAppropriateColorOn(canvas);
```

We can increase this isolation by making the name of the message simpler:

```
point.drawYourselfOn(canvas)
```

Here we don't want to know *any* detail: nothing about color, shape, visibility, or anything else. This isolation makes code **much easier to change**, because changes in the protocol between point and canvas will not impact the callers of either.

This preference for telling objects to do things rather than asking objects to return values was called "Tell, don't ask!" in a famous paper by Andy Hunt and Dave Thomas (TBD - reference).

Mocks

I hope I've convinced you that *Tell, don't ask* is good. Now you have a problem: how do you test an object that does not have getters? Ha!

The only way to do test such an object is to *observe its behaviour*. That is, we send it a message, and observe what other messages it sends to its neighbours. I send a `drawYourselfOn(canvas)` message to the point, and I want to test that the canvas was used correctly. How could I test that? Should I ask the canvas? That would require adding *getters* to the canvas.

```
// Bleah!  Don't do this!
for (int x=0; x<=canvas.getMaxX(); x++) {
  for (int y=0; y<=canvas.getMaxY(); y++) {
    if (x == point.getX() && y == point.getY())
      assertEquals(point.getColor(), canvas.getColorAt(x, y));
    else
      assertEquals(WHITE, canvas.getColorAt(x, y));
  }
}
```

I'd rather express my test (in pseudocode) this way:

```
I expect that
  canvas will receive drawPoint(x, y)
whenever I do
  point.drawYourselfOn(canvas)
```

So instead of an *assertion* we have an **expectation**. We expect that the canvas will be called in a certain way, *and nothing else*. The nice thing in this test is that *we don't care what's the behaviour of the canvas*. For all we care, we can just assume that `canvas` is just an interface. This means that we can develop an object with TDD before its collaborators even exist!

The above test, expressed in JMock, would look like the following:

```
Point point = new Point(10, 20);
Canvas canvas = context.mock(Canvas.class);
context.checking(new Expectations() {{
  oneOf(canvas).drawPoint(10, 20);
}});
point.drawYourselfOn(canvas);
```

The point of this test is that we can use it to define how the point should talk to its collaborator.

The syntax looks a bit esoteric at first, but it will all make sense. The details on how JMock works are in the appendix. More about mocks in [GOOS].

Skin & Wrap the API

TBD Reference Working Effectively With Legacy Code

4. Some tests are more useful, some tests are less

Why do we write tests? To me, the value I get from tests is mainly two things:

- To make sure that the software works as I expect
- To get help design the software

(both being equally important to me.) The overall goal behind all of this is to be able to deliver software faster (as Dan North says in his book https://leanpub.com/software-faster).

Now, if you read some Android testing literature, you are being encouraged to write tests like this one:

```
public class GuiTest extends ActivityInstrumentationTestCase2<MainActivity> {
    public GuiTest() {
        super(MainActivity.class);
    }

    public void testMessageGravity() throws Exception {
        TextView myMessage = (TextView) getActivity().findViewById(R.id.myMessage);
        assertEquals(Gravity.CENTER, myMessage.getGravity());
    }
}
```

This is test of the GUI layout. We write it, we run it (which requires launching an Android emulator, or downloading the code to a real device), and we get a red bar:

```
junit.framework.AssertionFailedError: expected:<17> but was:<8388659>
```

Now we go to the `content_main.xml` file and set the proper gravity:

```
<TextView
    android:text="Hello World!"
    android:id="@+id/myMessage"
    android:gravity="center"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

We run the code again, and look! It passes!

What have we accomplished here? Did we get value out of this testing? Let's see:

- Maing sure that the software works as I expect: do we get any extra assurance now? Not really. We just added a declaration in the XML file. What we really care about is that the message *looks right*. All we know is that it has a certain property.
- Getting help in designing the software: not at all. It's not like we're using the test to understand what line of code we should write. We knew perfectly well that we wanted to write that `android:gravity="center"` line of code, before writing the test. Writing the test was done just to grant ourselves *permission* to write the line of code we already knew we wanted.

I call tests like this one "bureaucratic tests", because they look like I have to sign a request form (the test) just to get permission to write a line of code I already think I need.

* * *

What kinds of tests are valuable, then?

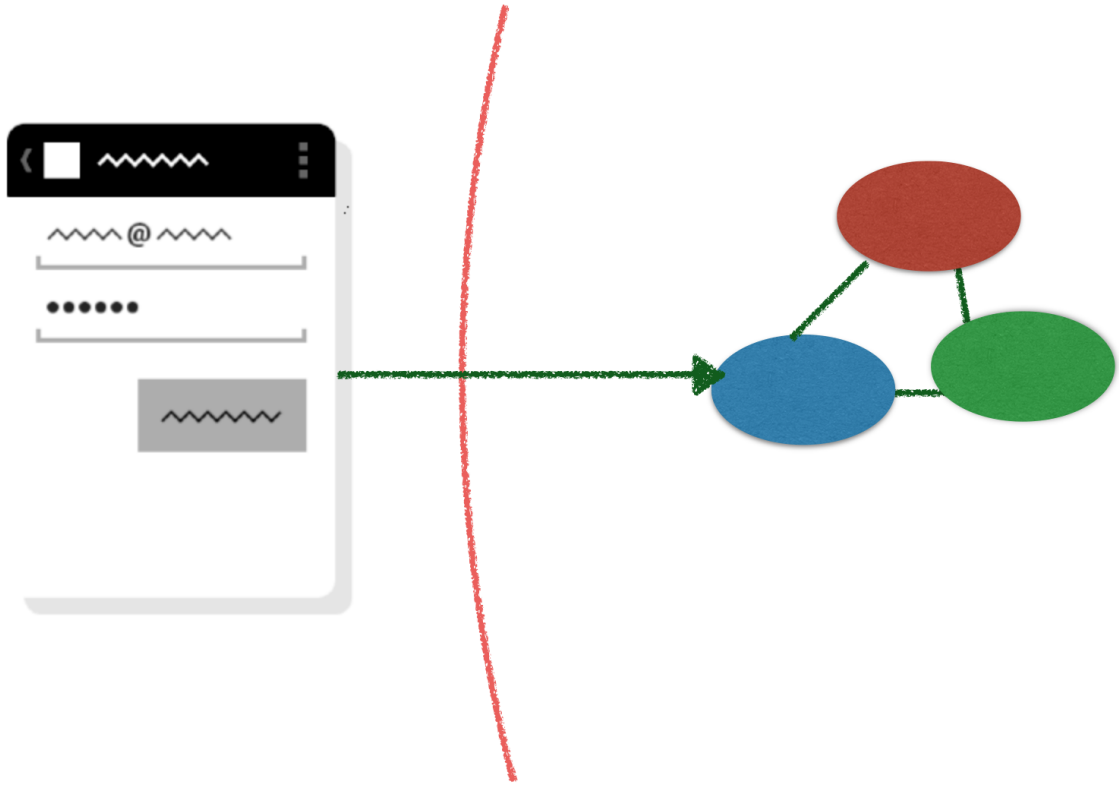
In TDD, like in XP, it all starts from what is valuable *for the customer* of the software. So we start from the requirements, and work backwards to understand what we need to write. A user story is made up of a number of *scenarios*, or *concrete examples*, of what the software should do. So we write one test for each of these scenarios. This is well explained in the many resources on BDD, ATDD and Cucumber; however, I don't recommend that you write your tests with Cucumber, necessarily. You can write these test with JUnit just as well.

When a scenario-level test is too big to pass quickly, then you may ignore it temporarily, and write smaller tests that take you one step closer to getting the original test to pass. This is a pattern that Kent Beck calls "Child test" and is well explained in his Test-Driven Development: By Example book.

Model-view separation

This explanation leaves one question unanswered. How do we test the GUI then?

We apply one principle: **model-view separation**.



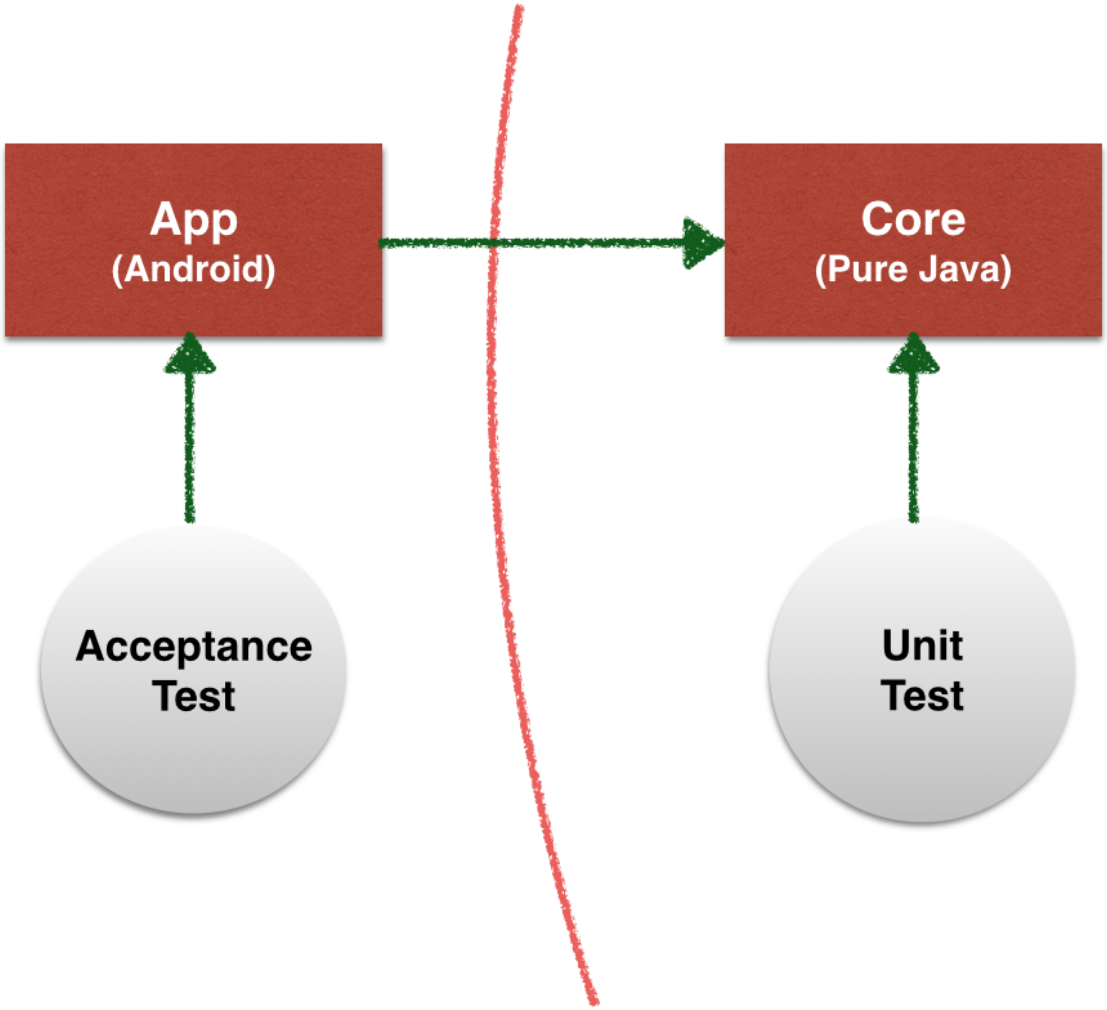
Model-view separation

The GUI does not contain any logic. All the logic is extracted to plain-Java objects that can be developed and tested with no reference to the Android framework. This principle is a cornerstone of Object-Oriented Design, and is well explained in Craig Larman's book Applying UML and Patterns.

When you are developing a new piece of GUI, you probably want to loop quickly between changing something in the GUI code, and seeing how it looks. Automated tests are not very useful here: what you care about is how it looks to your eyes.

When you are releasing the app and want to make sure that no screen of your application is broken, you probably want to check every screen by hand. This is less burdensome that it looks. If the GUI does not contain any IF, then it's easy to check that it works: you just have to look at every screen *once*. You can write a **release checklist** that lists all the things you want to look at before releasing the app; it should be possible to perform the tour of the app in minutes.

When you are doing your everyday coding and refactoring, you certainly don't want to check manually that you haven't broken any view at every commit. For this, you should have **one** end-to-end test for every screen, that tries to perform a basic test of the basic functionality offered by the screen. These test do not guarantee that the screen looks right, but at least they guarantee that the basic functionality works.



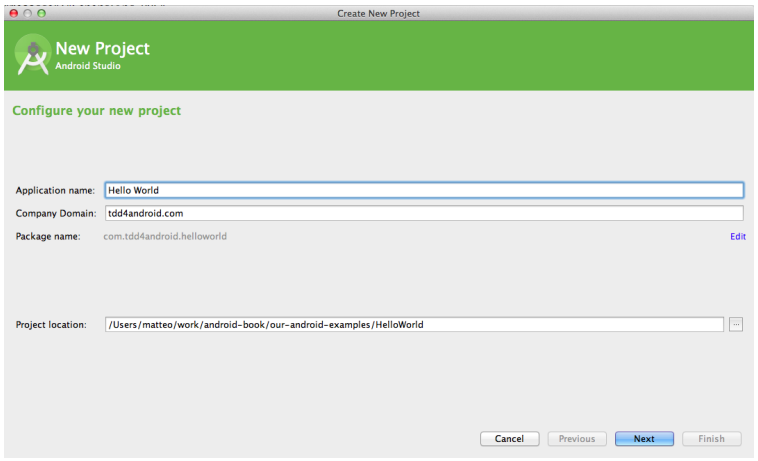
A few tests exercise the GUI, most of the tests don't

5. Hello, World!

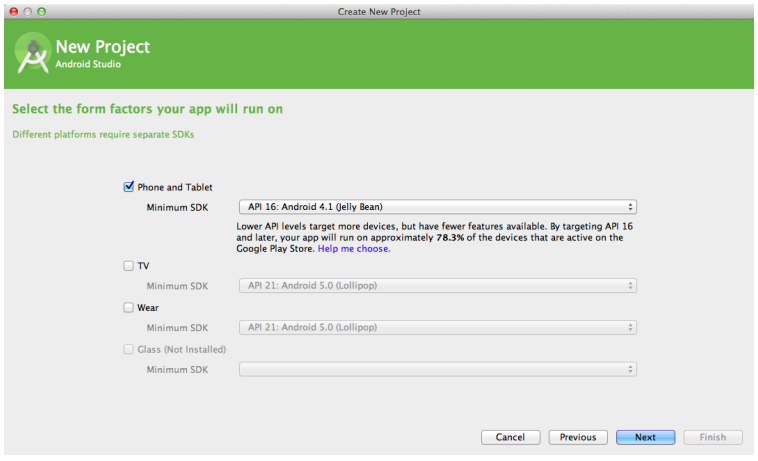
In this chapter we see how to set up a project properly for our purposes. We will execute a small "hello, world" kata. All the other projects in this book start the same way, except that the name of the project will be different each time. You should be able to execute this kata easily in 25 minutes, or one "pomodoro".

First step: new project!

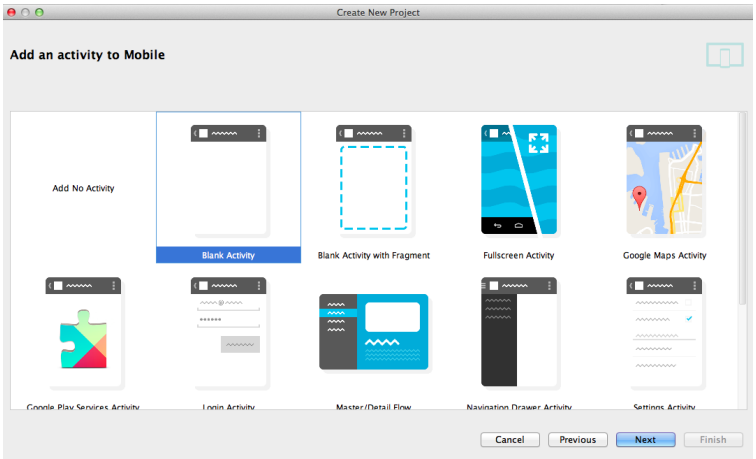
Start Android Studio and generate a new project with default options. Change only the names of the project, the package and the main activity.



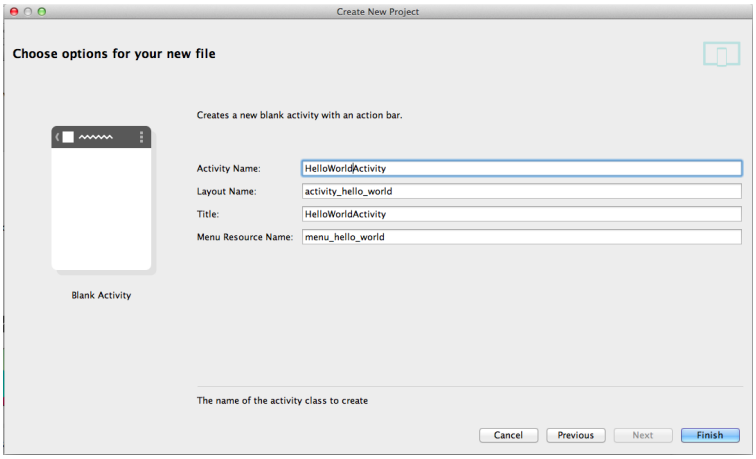
The Create New Project wizard in Android Studio



Select the desired API level

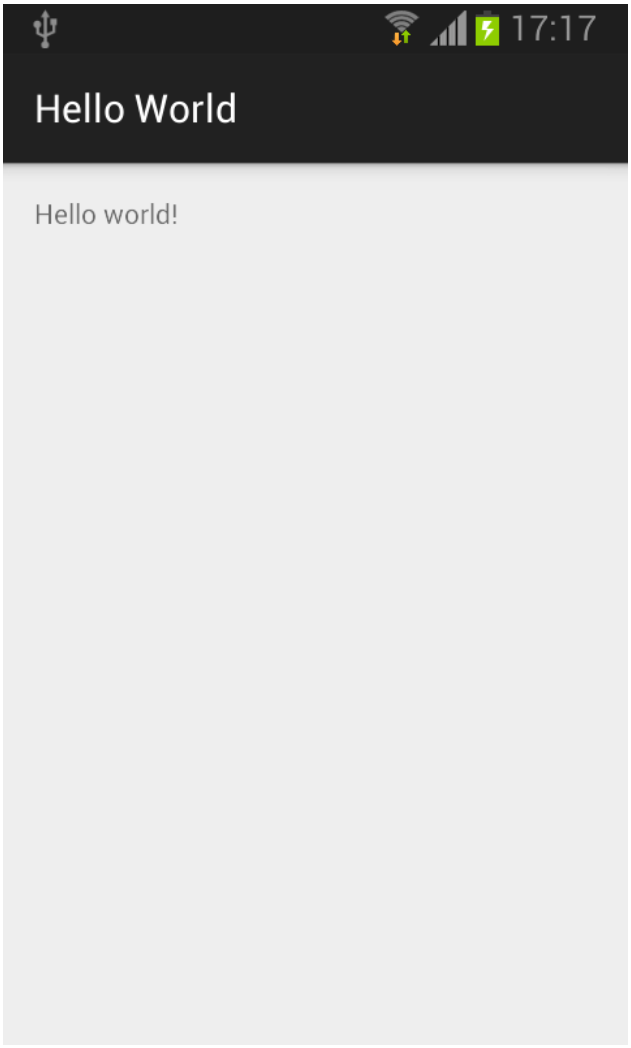


Add a blank activity



Name the activity

Run the project and observe the “hello, world” message on the screen of your device.



The app that the wizard built for us

Change the text

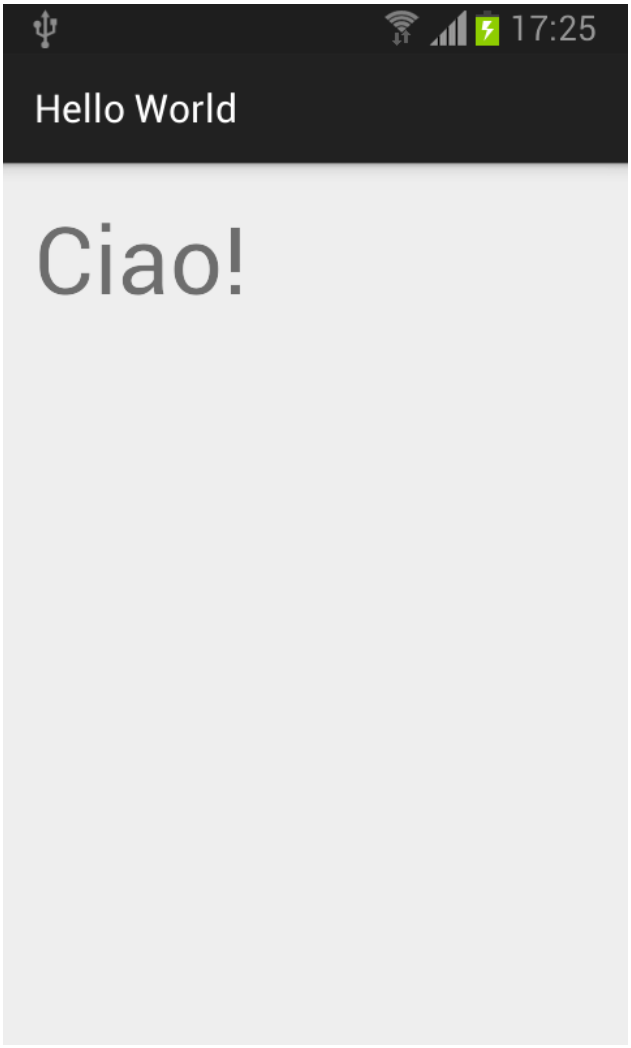
Now we alter the view. We want to be able to change the “Hello, world!” message programmatically. We change the `app/res/layout/activity_hello_world.xml` this way:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
                xmlns:tools="http://schemas.android.com/tools"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:paddingLeft="@dimen/activity_horizontal_margin"
                android:paddingRight="@dimen/activity_horizontal_margin"
                android:paddingTop="@dimen/activity_vertical_margin"
                android:paddingBottom="@dimen/activity_vertical_margin"
                tools:context=".HelloWorldActivity">

    <TextView
        android:id="@+id/message"
        android:textSize="48sp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</RelativeLayout>
```

We removed the canned message string, and we added an id to the TextView, on line 12. We also added a textSize attribute so that the message is shown much bigger, on line 13. Now we can change the message in the `HelloWorldActivity`:

```
public class HelloWorldActivity extends ActionBarActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_hello_world);
    TextView view = (TextView) findViewById(R.id.message);
    view.setText("Ciao!");
  }
}
```

We run the application again, and we see that the message has changed.

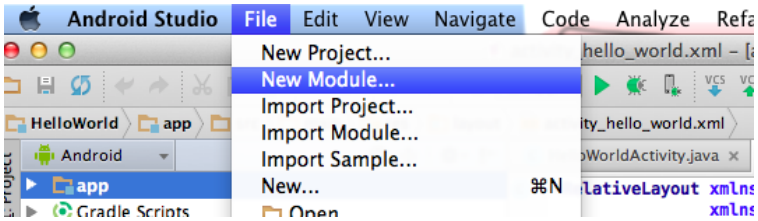


We changed the message programmatically

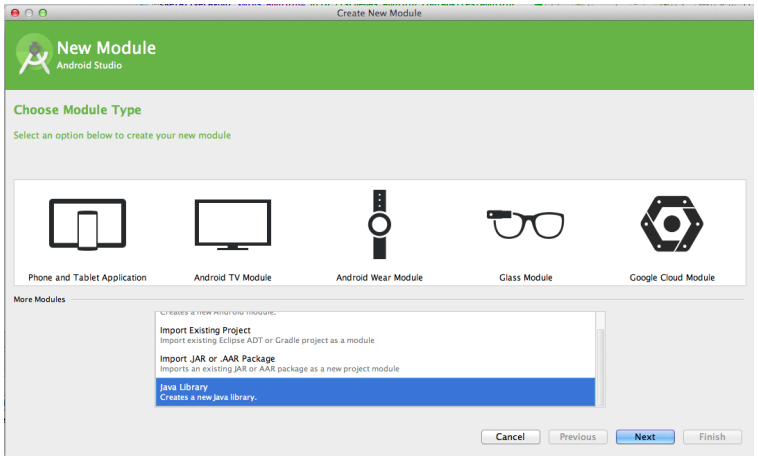
Introduce a new module

We now want to move the logic that produces the error message away from the `app` module, into a new module that will contain pure Java logic, free from any dependency on the Android APIs. By convention, we always call this module `core`.

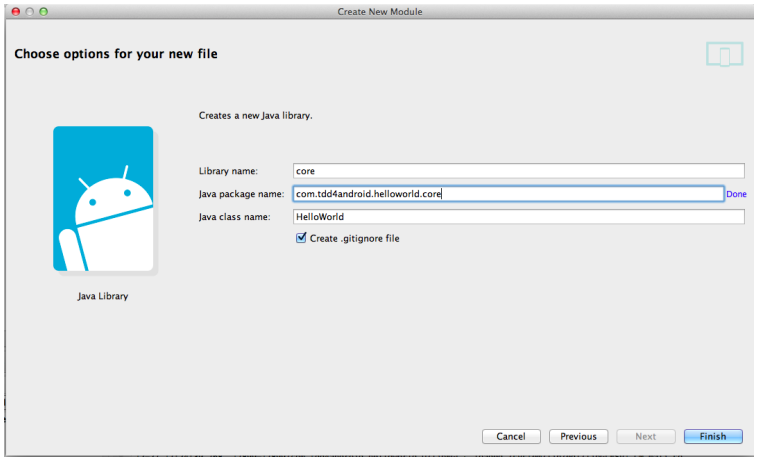
Create a new module with Android Studio. Remember to change the package name. Android Studio requires us to name the first class in this module, so we call it `HelloWorld`.



Create a new module



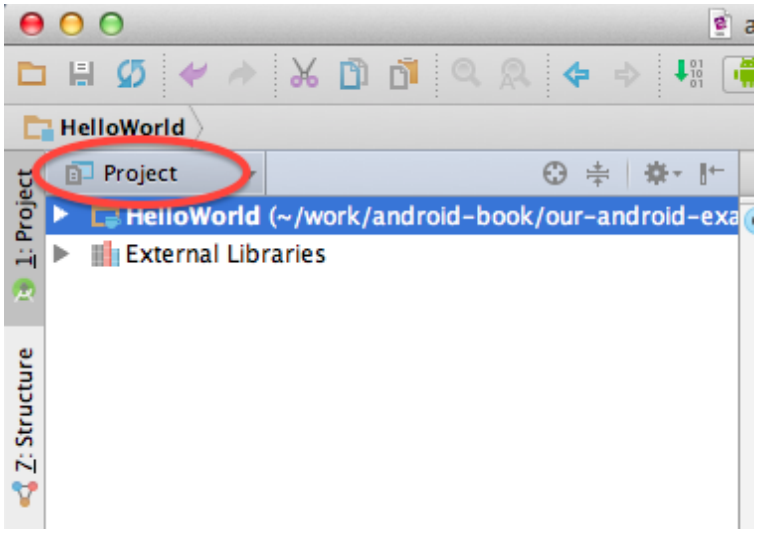
Choose a "Java Library" module type



Name the module and its first class

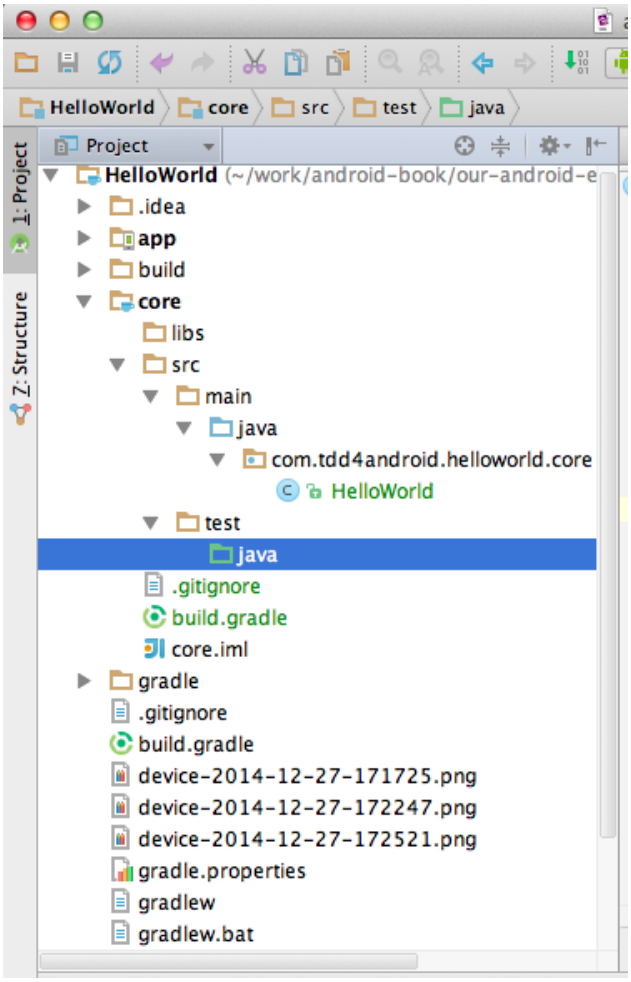
First unit test

We now introduce a first unit test. Create directory `core/src/test/java`; you can do this with either the command line or within Android Studio. If you choose the latter, remember to switch to the Project view first.



Change from "Android" to "Project" view

Then you right-click your way through the project and create directory `test` under directory `src`, and then you create directory `java` under directory `test`. The end result should be as in the following screenshot.



Change from "Android" to "Project" view

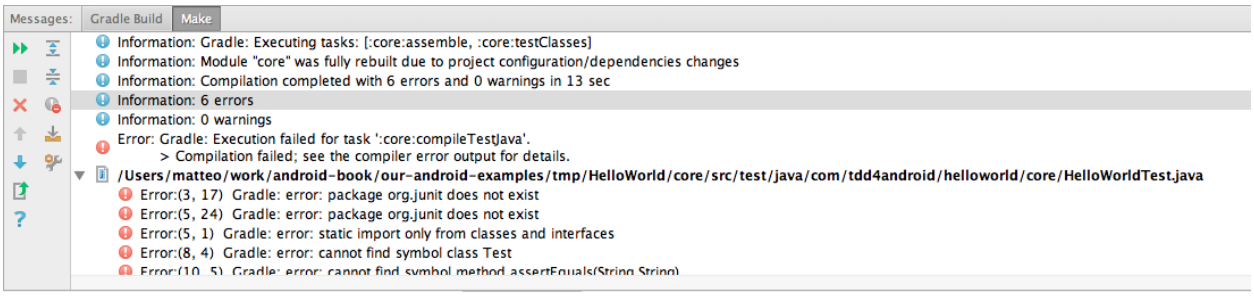
Now you create a new class called `HelloWorldTest` in directory `core/src/test/java` with the following content:

```
package com.tdd4android.helloworld.core;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class HelloWorldTest {
  @Test
  public void testHelloWorld() throws Exception {
    assertEquals("42", new HelloWorld().message());
  }
}
```

The `message` method does not compile, because we didn't create any method in class `HelloWorld`. We fix quickly like this:

```
package com.tdd4android.helloworld.core;

public class HelloWorld {
  public String message() {
    return null;
  }
}
```

We run all the tests in module `core`, and we see a failure: it seems that JUnit is not in the project classpath.



We should add a dependency on JUnit

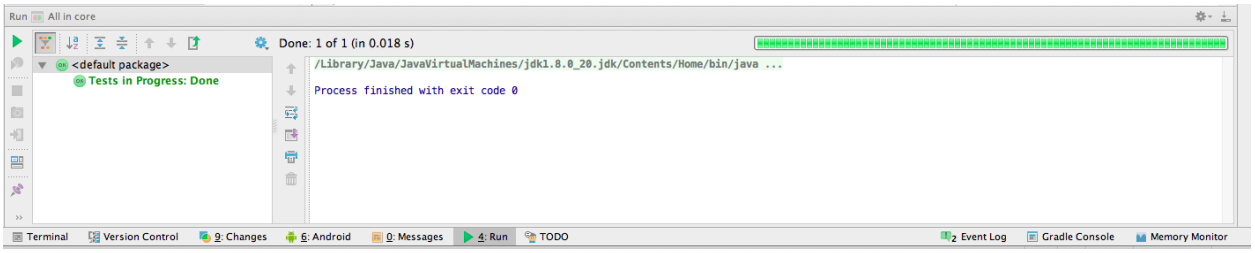
We fix it by changing `core/build.gradle`

```
apply plugin: 'java'

// Fix the version of Java; 1.7 is the highest version supported by Android
sourceCompatibility = 1.7

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])

    // Add dependency on JUnit
    testCompile 'junit:junit-dep:4.11'
}

// Improve test logging
test {
    testLogging {
        events "passed", "skipped", "failed", "standardOut", "standardError"
    }
}
```

We run the tests again, and look! We now fail for the right reason!

Failing for the right reason

We fix the `HelloWorld` class and we get a green bar.



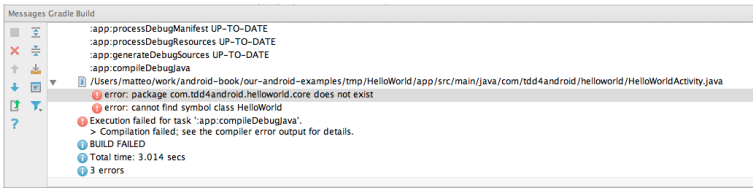
Finally, a green bar

Using the core logic in the app

Now we want to use our new nifty HelloWorld class to produce the message on the Android device's screen. We change `HelloWorldActivity` as follows:

```
package com.tdd4android.helloworld;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import com.tdd4android.helloworld.core.HelloWorld;

public class HelloWorldActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_hello_world);
    TextView view = (TextView) findViewById(R.id.message);
    view.setText(new HelloWorld().message());
  }
}
```

We try to run the application, and it does not compile.

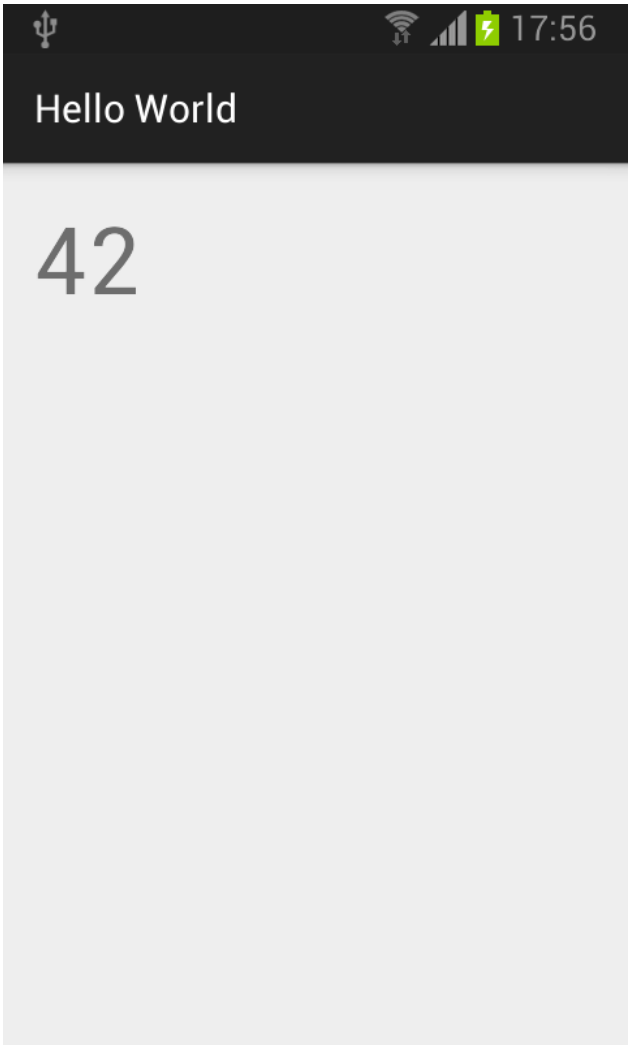


We should add a dependency on module "core"

We need to change `app/build.gradle` to let module `app` depend on module `core`.

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 21
    buildToolsVersion "21.1.2"

    defaultConfig {
        applicationId "com.tdd4android.helloworld"
        minSdkVersion 16
        targetSdkVersion 21
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.\
pro'
        }
    }
}

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

    // We need classes from module "core"
    compile project(':core')
}
```

We run the application again and it works!



Get the answer from the core module

What we've done

We've developed a miniature application, following a miniature version of the Android TDD process that we recommend.

1. We check that the Android part works manually
2. We move the logic to a `core` module that does not depend on Android; we further develop the logic with JUnit.
3. We call the `core` module from the Android module as needed
4. We check manually that the connection from `app` to `core` works.

The whole project at this point is available from `https://github.com/xpmatteo/tdd4android/tree/starting-point`. If you wish to quick-start a new project, you may just clone it from Github and import it in Android Studio.

6. Some tests are useful, some tests are not

Why do we write tests? To me, the value I get from tests is mainly two things:

- To make sure that the software works as I expect
- To get help design the software

both being equally important to me. The overall goal behind all of this is to be able to *deliver software faster*, as Dan North says in his book https://leanpub.com/software-faster.

Now, if you read some Android testing literature, you are being encouraged to write tests like this one:

```
public class GuiTest extends ActivityInstrumentationTestCase2<MainActivity> {
    public GuiTest() {
        super(MainActivity.class);
    }

    public void testMessageGravity() throws Exception {
        TextView myMessage = (TextView) getActivity().findViewById(R.id.myMessage);
        assertEquals(Gravity.CENTER, myMessage.getGravity());
    }
}
```

This is test of the GUI layout. We write it, we run it (which requires launching an Android emulator, or downloading the code to a real device), and we get a red bar:

```
junit.framework.AssertionFailedError: expected:<17> but was:<8388659>
```

Now we go to the `content_main.xml` file and set the proper gravity:

```
<TextView
  android:id="@+id/myMessage"
  android:gravity="center"
/>
```

We run the code again, and look! It passes! What have we accomplished here? Did we get value out of this testing? Let's see:

- Maing sure that the software works as I expect: do we get any extra assurance now? Not really. We just added a declaration in the XML file. What we really care about is that the message *looks right*. All we know is that it has a certain property.

- Getting help in designing the software: not at all. It's not like we're using the test to understand what line of code we should write. We knew perfectly well that we wanted to write that `android:gravity="center"` line of code, before writing the test. Writing the test was done just to grant ourselves *permission* to write the line of code we already knew we wanted.

I call tests like this one "bureaucratic tests", because they look like I have to sign a request form (the test) just to get permission to write a line of code I already think I need.

* * *

What kinds of tests are valuable, then?

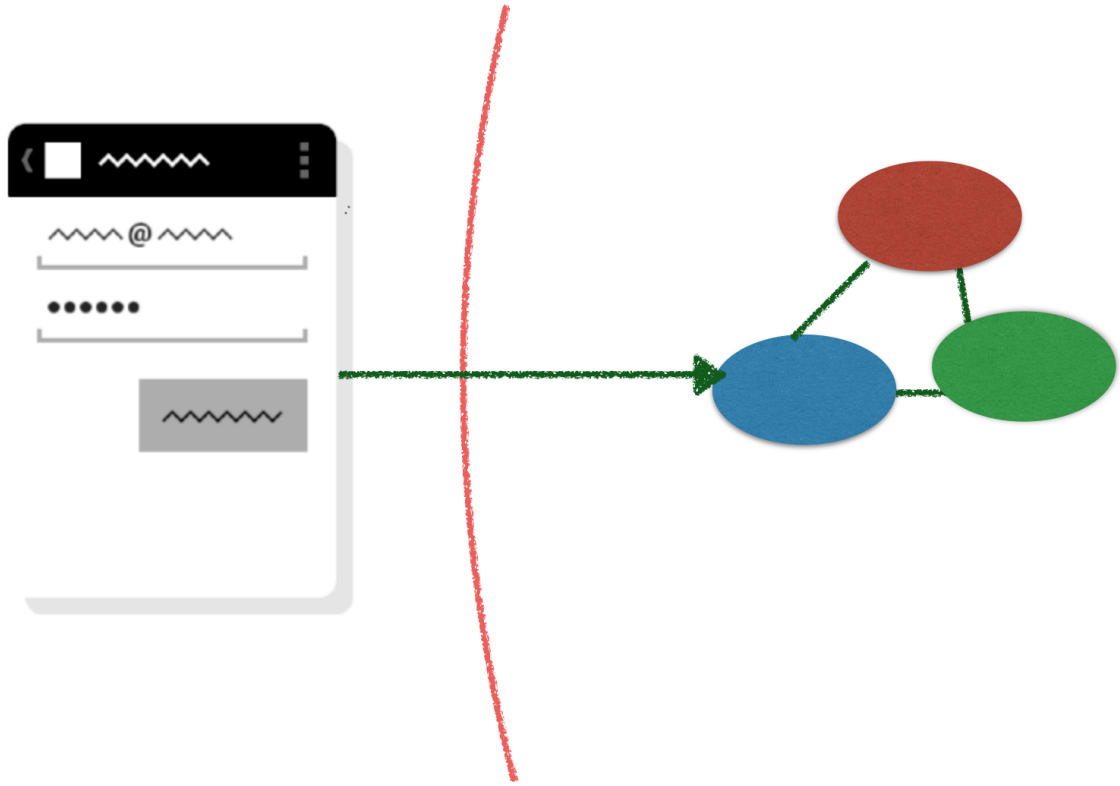
In TDD, like in XP, it all starts from what is valuable *for the customer* of the software. So we start from the requirements, and work backwards to understand what we need to write. A user story is made up of a number of *scenarios*, or *concrete examples*, of what the software should do. So we write one test for each of these scenarios. This is well explained in the many resources on BDD, ATDD and Cucumber; however, I don't recommend that you write your tests with Cucumber, necessarily. You can write these test with JUnit just as well.

When a scenario-level test is too big to pass quickly, then you may ignore it temporarily, and write smaller tests that take you one step closer to getting the original test to pass. This is a pattern that Kent Beck calls "Child test" and is well explained in his Test-Driven Development: By Example book.

Model-view separation and testing the GUI

This explanation leaves one question unanswered. How do we test the GUI then?

We apply one principle: **model-view separation**.



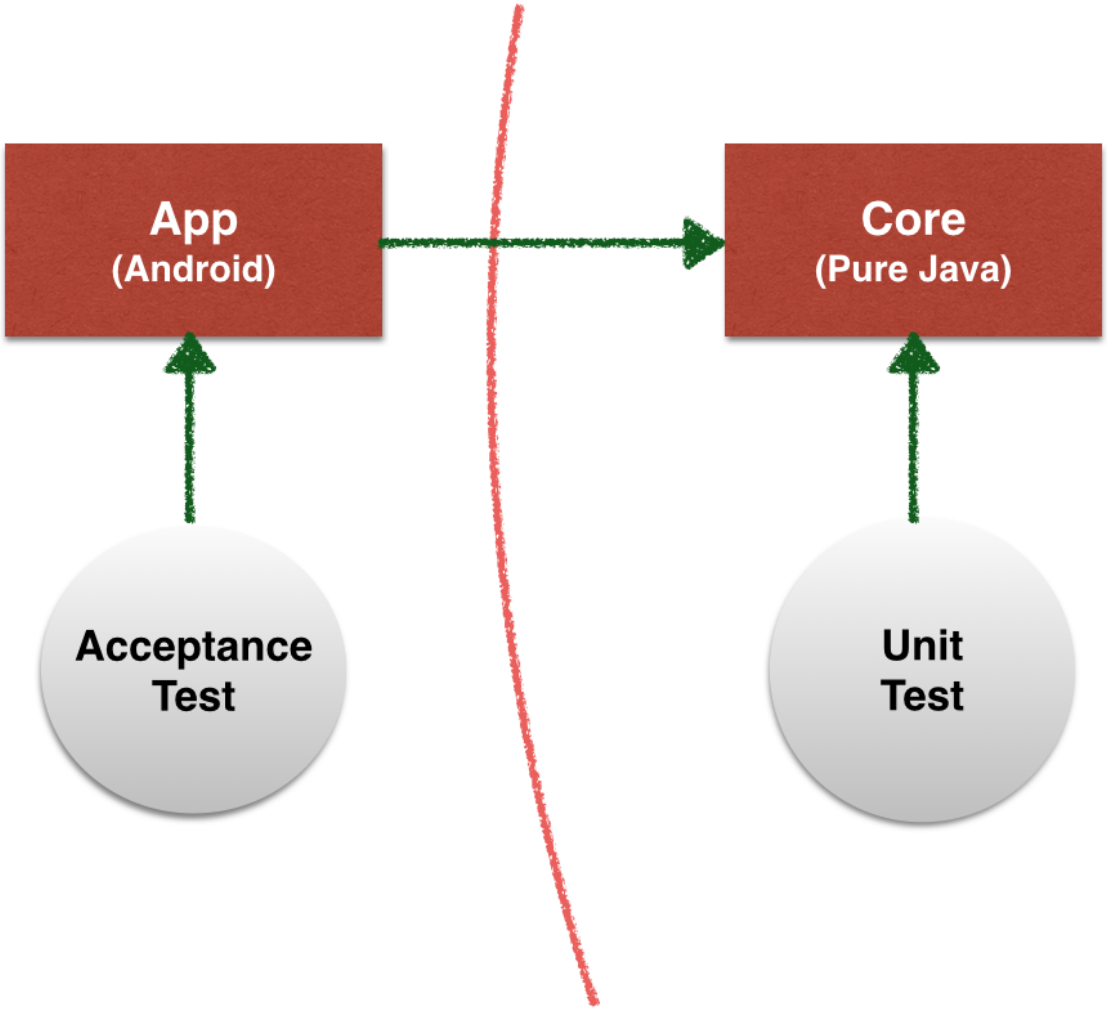
Model-view separation

The GUI does not contain any logic. All the logic is extracted to plain-Java objects that can be developed and tested with no reference to the Android framework. This principle is a cornerstone of Object-Oriented Design, and is well explained in Craig Larman's book Applying UML and Patterns.

When you are developing a new piece of GUI, you probably want to loop quickly between changing something in the GUI code, and seeing how it looks. Automated tests are not very useful here: what you care about is how it looks to your eyes.

When you are releasing the app and want to make sure that no screen of your application is broken, you probably want to check every screen by hand. This is less burdensome that it looks. If the GUI does not contain any IF, then it's easy to check that it works: you just have to look at every screen *once*. You can write a **release checklist** that lists all the things you want to look at before releasing the app; it should be possible to perform the tour of the app in minutes.

When you are doing your everyday coding and refactoring, you certainly don't want to check manually that you haven't broken any view at every commit. For this, you should have **one** end-to-end test for every screen, that tries to perform a basic test of the basic functionality offered by the screen. These test do not guarantee that the screen looks right, but at least they guarantee that the basic functionality works.



A few tests exercise the GUI, most of the tests don't

7. A simple form-based application: Unit Doctor

Problem description

We want to write an application to convert measures from various units to other units. We imagine that it will support an extensive collection of conversions, e.g., from centimeters to yards, from Fahrenheit degrees to Celsius, from HP to KW. You may check out the source code for this example at https://github.com/xpmatteo/unit-doctor.

Examples

The feature we want to implement is "convert a number from one unit to another". To clarify what we want to do to ourselves and our customer, it's a good idea to write down a few examples (a.k.a. scenarios) of how the feature will work.

> **Example: inches to cm**
> Given the user selected "in" to "cm"
> When the user types 2
> Then the result is "2.00 in = 5.08 cm"

> **Example: Fahrenheit to Celsius**
> Given the user selected "F" to "C"
> When the user types 50
> Then the result is "50.00 F = 10.00 C"

> **Example: unsupported units**
> Given the user selected "ABC" to "XYZ"
> Then the result is "I don't know how to convert this"

Note that by writing down the examples we clarified what exactly the customer expects to see: how numbers are formatted, what the result message should look like.

Start with a spike

When you are using APIs you're not familiar with, it's better to do a *spike* before you start doing real TDD.

What is a spike?

A *spike* is an experiment that you do in order to explore how to do a feature. A spike will usually not have tests, will be quick and dirty, will not be complete, will not follow our usual rules for good quality. It's just an exploration. The rules for spikes are

1. New project: start the spike in a new project (not by hacking into your production code)
2. Timebox: set yourself a time limit, for instance two hours.
3. Throw away: after you're done, you *throw away* the spike code. You may keep the spike around as a *junkyard* of bits to copy from; but you never turn the spike into your production project. Start production code with a fresh project.

The goal of our spike is to understand how to receive text from the Android API and how to position form elements in a layout. Trying to implement just the "inches to cm" and the "unsupported units" scenarios should give us enough understanding.

How the unit conversion spike looks like

The activity for the unit conversion spike

```
package name.vaccari.matteo.unitconversionspike;

import android.app.Activity;
import android.os.Bundle;
import android.view.*;
import android.widget.*;


public class MyActivity extends Activity implements View.OnKeyListener {

    private EditText inputNumber;
    private TextView outputNumber;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my);

        getEditText(R.id.inputNumber).setOnKeyListener(this);
        getEditText(R.id.fromUnit).setOnKeyListener(this);
        getEditText(R.id.toUnit).setOnKeyListener(this);
    }

    @Override
    public boolean onKey(View v, int keyCode, KeyEvent event) {
        String input = getEditText(R.id.inputNumber).getText();
        String fromUnit = getEditText(R.id.fromUnit).getText();
        String toUnit = getEditText(R.id.toUnit).getText();
        getEditText(R.id.result).setText(" Hello! " + input + fromUnit + toUnit);
        return false;
    }

    private EditText getEditText(int id) {
        return (EditText) findViewById(id);
    }

    private String getEditTextContent(int id) {
        return getEditText(id).getText().toString();
    }
}
```

The layout for the unit conversion spike

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    tools:context=".MyActivity">

    <EditText
        android:layout_width="200dp"
        android:layout_height="wrap_content"
        android:inputType="numberDecimal"
        android:id="@+id/inputNumber"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/fromUnit"
        android:layout_toRightOf="@+id/inputNumber"
        android:text="in"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="  To  "
        android:id='@+id/to'
        android:layout_toRightOf="@id/fromUnit"
        android:layout_alignBaseline="@id/fromUnit" />

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/toUnit"
        android:layout_toRightOf="@+id/to"
        android:text="cm"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:text="Result"
        android:id="@+id/result"
        android:layout_below="@+id/inputNumber"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginTop="20dp" />
</RelativeLayout>
```

As expected, the spike exercise permit us to learn (among other things):

- How to change the result at every keypress
- How to use a `RelativeLayout`

In general, we'll use in this book the spike technique as an help to discover the correct point where to place the boundary bewteen android dependent and android independent code.

Continue with an end-to-end acceptance test

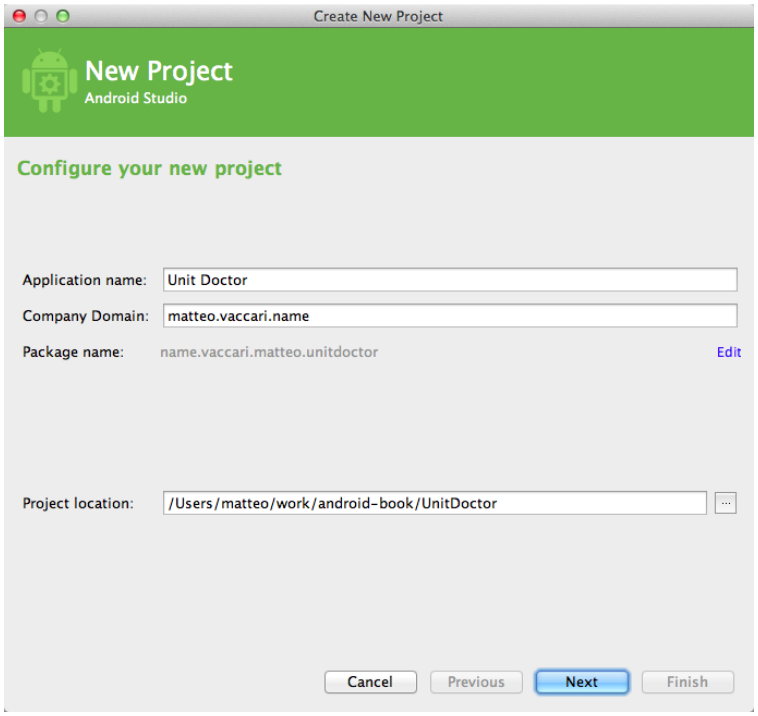
The first step after the optional spike is to write an end-to-end acceptance test. The rules for an acceptance tests are

1. Business talk: write at the same level of abstraction as the examples that were discussed with the customer.
2. Ignore them: when the ATs pass, then the feature is complete. Therefore it will take some time before all the ATs pass. We will ignore all the non-passing ATs until they pass. (Ignore means that we comment them out, or we use some other trick to temporarily remove them from the build.)
3. Pass once, pass forever. When an AT passes, it means that some bit of functionality is present in the system. From this moment onward, the AT must stay green forever. The AT now works as a non-regression test.

Workflow

Step 0: new project

We create a new project (remember, we don't want to "evolve" the spike!). I use Android Studio and I let it set up the new project with its wizard. I invent a fancy name and I call it "Unit Doctor".



The new project wizard

Step 1: write the test *as it should read*

Android Studio set up a source folder named `src/androidTest`. I create a new Java class there.

```
1  package name.vaccari.matteo.unitdoctor;
2
3  import android.test.ActivityInstrumentationTestCase2;
4
5  public class UnitConversionAcceptanceTest
6      extends ActivityInstrumentationTestCase2<MainActivity> {
7
8    public UnitConversionAcceptanceTest() {
9      super(MainActivity.class);
10   }
11
12   public void testInchesToCentimeters() throws Exception {
13     givenTheUserSelectedConversion("in", "cm");
14     whenTheUserEnters("2");
15     thenTheResultIs("2.00 in = 5.08 cm");
16   }
17 }
```

The test is written in the same language as the original example, by imagining that we have implemented the three methods at lines 13-15.

Step 2: implement the helper methods

We assume at first that the UI is similar to what we developed in the spike. So our first implementation of the testing language is quite simple.

```
public void testInchesToCentimeters() throws Exception {
  givenTheUserSelectedConversion("in", "cm");
  whenTheUserEnters("2");
  thenTheResultIs("2.00 in = 5.08 cm");
}

private void givenTheUserSelectedConversion(String fromUnit, String toUnit) {
  getField(R.id.fromUnit).setText(fromUnit);
  getField(R.id.toUnit).setText(toUnit);
}

private void whenTheUserEnters(String inputNumber) {
  getField(R.id.inputNumber).setText(inputNumber);
}

private void thenTheResultIs(String expectedResult) {
  assertEquals(expectedResult, getField(R.id.result).getText());
}

private TextView getField(int id) {
  return (TextView) getActivity().findViewById(id);
}
```

The test does not compile yet, because the identifiers like `R.id.fromUnit` etc. are not yet defined.

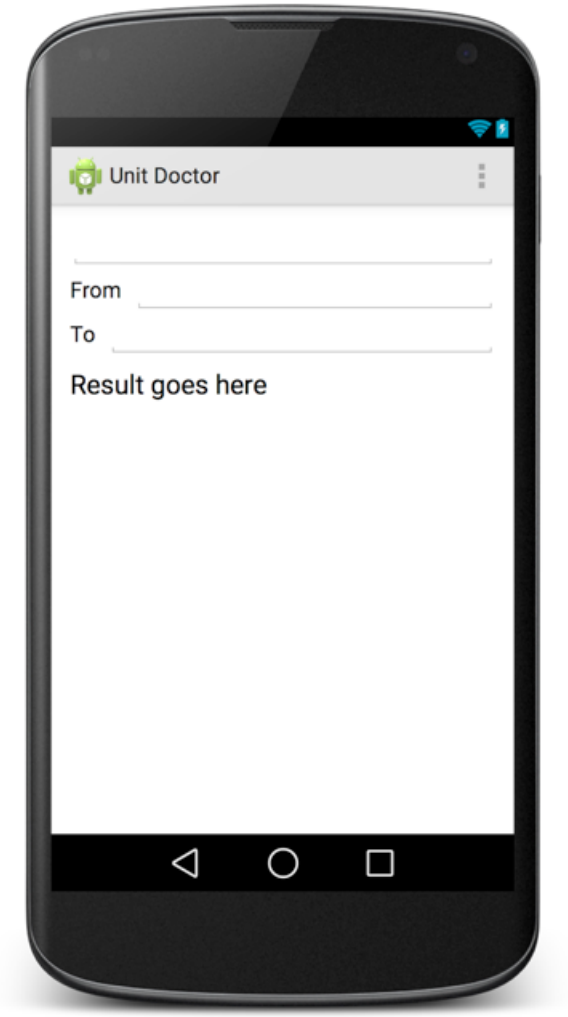
Step 3: implement other scenarios

We implement a few other significant scenarios. This way we check that the testing language that we are developing is expressive enough.

```
public void testFahrenheitToCelsius() throws Exception {
    givenTheUserSelectedConversion("F", "C");
    whenTheUserEnters("50");
    thenTheResultIs("50.00 F = 10.00 C");
}

public void testUnknownUnits() throws Exception {
    givenTheUserSelectedConversion("ABC", "XYZ");
    thenTheResultIs("I don't know how to convert this");
}
```

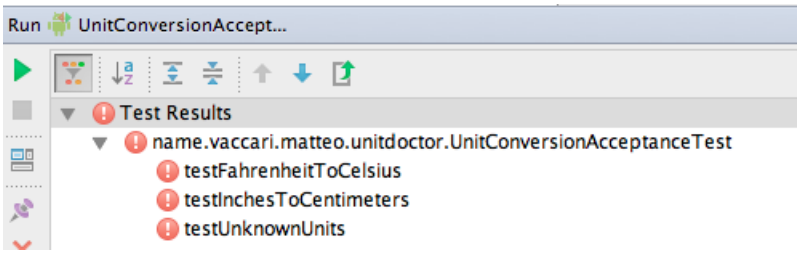
Step 4: implement just enough layout so that we can see the test fail

We turn to editing the layout file so that we can fix all the IDs. This is the time in which the layout designer specialist (but not us) can express himself.



The Android Studio preview of the UI

Now we can run the tests and see them all fail



The first run of the ATs unexpectedly produces errors, not failures

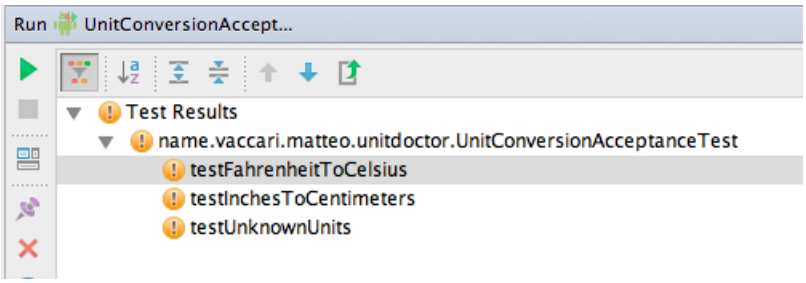
But are they failing for the expected reason?

```
android.view.ViewRootImpl$CalledFromWrongThreadException:
Only the original thread that created a view hierarchy can touch its views.
at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:4607)
...
at name.vaccari.matteo.unitdoctor.UnitConversionAcceptanceTest.givenTheUserSelectedConvers\
ion(UnitConversionAcceptanceTest.java:30)
at name.vaccari.matteo.unitdoctor.UnitConversionAcceptanceTest.testInchesToCentimeters(Uni\
tConversionAcceptanceTest.java:13)
...
```

The error message is "Only the original thread that created a view hierarchy can touch its views." So it fails because we can only touch an element of the UI, such as when we call `setText()` on the text fields, using the main thread of the application. The easiest way to solve the problem, for now, is to annotate the tests with `@UiThreadTest`. We do so, and now we check that the error message is what we expect:

```
junit.framework.AssertionFailedError:
expected:<50.00 F = 10.00 C> but was:<Result goes here>
```

We also observe that Android Studio colors the exclamation point differently (yellow for failures and red for errors), to tell us that the tests produce failures, not errors.



Now the ATs produce failures as expected

Tip: Always check the error message, to make sure that the tests are failing for the right reason.

What is the difference between a *failure* and an *error*? A "failure" is when your test fails because of a broken assertion. An "error" is when the test fails because of an exception. In general, we want our test to produce failures, not errors, because errors mean that our software is doing something unexpected.

Shift to TDD

Now that we have some failing acceptance tests, we leave them be. We now shift to a faster gear by starting to write some *unit* tests.

Shouldn't we make the acceptance tests pass before we continue? Not necessarily. AT's usually take a long time to pass. For the moment, we leave them broken. If we worked in a team then we wouldn't want to break the build, so we would comment them out or "ignore" them by adding an "x" to the beginning of the name of each test method, so that JUnit does not run them.

Unit tests live in their own module

In order to do this we will start by assuming that we have a kernel of the application that does not need to use Android at all. For this reason we open a new module inside the project. (If I were using Eclipse, I would start a new project in the workspace.) We call the new module **UnitDoctorCore**.

Tip: keep android-free code in a separate module or project. Keep android-dependent code in a module that depends on the android-free module.

The project structure so far then is:

- "UnitDoctor" project
  - "app" module, where Android-dependent code lives
  - "UnitDoctorCore" module, where the pure, Android-free logic of the program lives

We make sure that "app" depends on "UnitDoctorCore", so that code in "app" can use objects defined in "UnitDoctorCore". The latter must not depend on "app": the application logic should not depend on the GUI or other infrastructure details.

Our test list

TDD should start with a *test list*: a list of all the *operations* that we should support. Right now we have

- Convert in to cm
- Convert F to C
- Report conversion not supported

from the acceptance tests. It's clear that some additional operations we need are

- Convert input number to double
- Format output message

Now we choose one test from the list. One that seems interesting is "Convert in to cm", so we do that first. We want to write a test for this operation and the question is: "which object will support this operation?"

We will use a technique called *presenter-first*. We assume that we have an object that represents the whole application. We give it the name of the application: `UnitDoctor`. We also assume that we will have a "view" object, that represents the Android GUI. The gist of what we want to test is

> If the view says that the inputs are 1.0, "in" and "cm"
> When we are asked to perform a conversion
> Then we should tell the view that the result is 2.54.

The responsibilities of the "view" object are to return what the user has entered, and to show the results to the user.

This is the first test. If you find the syntax weird, look for an explanation in the JMock appendix.

```
10  @Rule
11  public JUnitRuleMockery context = new JUnitRuleMockery();
12
13  UnitDoctorView view = context.mock(UnitDoctorView.class);
14  UnitDoctor unitDoctor = new UnitDoctor(view);
15
16  @Test
17  public void convertInchesToCm() throws Exception {
18    context.checking(new Expectations() {{
19      allowing(view).inputNumber(); will(returnValue(1.0));
20      allowing(view).fromUnit(); will(returnValue("in"));
21      allowing(view).toUnit(); will(returnValue("cm"));
22      oneOf(view).showResult(2.54);
23    }});
24
25    unitDoctor.convert();
26  }
```

Notes:

- We have defined an interface UnitDoctorView. This interface is being *mocked* here.
- The whole point of "mocking" is to define how the object we are testing, the UnitDoctor, interacts with its collaborator (the UnitDoctorView).
- We don't have yet an implementation for UnitDoctorView. Yet we are able to make progress on the UnitDoctor by mocking the collaborator.
- We don't want UnitDoctor to decide how to format the string to the user; so we just tell the view what is the number to show, and delegate the actual formatting to the view itself.

In order to make this test compile, we have defined the interface

```
public interface UnitDoctorView {
  double inputNumber();
  String fromUnit();
  String toUnit();
  void showResult(double result);
}
```

Making this test pass is easy:

```
public class UnitDoctor {
  private UnitDoctorView view;

  public UnitDoctor(UnitDoctorView view) {
    this.view = view;
  }

  public void convert() {
    double inputNumber = view.inputNumber();
    view.showResult(inputNumber * 2.54);
  }
}
```

The next test forces us to take also the units into account. Looking at our test list we choose "Report conversion not supported".

```
@Test
public void showsConversionNotSupported() throws Exception {
  context.checking(new Expectations() {{
    allowing(view).inputNumber(); will(returnValue(anyDouble()));
    allowing(view).fromUnit(); will(returnValue("XYZ"));
    allowing(view).toUnit(); will(returnValue("ABC"));
    oneOf(view).showConversionNotSupported();
  }});

  unitDoctor.convert();
}

private double anyDouble() {
    return Math.random();
}
```

This forces us to add method `showConversionNotSupported` to the `UnitDoctorView` interface. We make it pass with

```
public void convert() {
  double inputNumber = view.inputNumber();
  if (view.fromUnit().equals("in") && view.toUnit().equals("cm"))
    view.showResult(inputNumber * 2.54);
  else
    view.showConversionNotSupported();
}
```

Continuing down this path we add another test (not shown) to add support for Fahrenheit-to-Celsius:

```
public void convert() {
  double inputNumber = view.inputNumber();
  if (view.fromUnit().equals("in") && view.toUnit().equals("cm"))
    view.showResult(inputNumber * 2.54);
  else if (view.fromUnit().equals("F") && view.toUnit().equals("C"))
    view.showResult((inputNumber - 32) * 5.0/9.0);
  else
    view.showConversionNotSupported();
}
```

This chain of ifs we don't like, but we'll leave it be for the moment. We have implemented the logic for our original examples, so our priority now is to see the application running!

Wouldn't it be better to use Mockito instead of JMock? Well, I find that Mockito is easier to learn, but I prefer JMock. There is more than one reason. The most compelling is that it lets me write tests that are more expressive. With Mockito you must say, for instance:

```
when(model.getNumber()).thenReturn(42);

presenter.render();

verify(view).showNumber(42);
```

With JMock it would be

```
context.checking(new Expectations() {{
  allowing(model).getNumber(); will(returnValue(42));
  oneOf(view).showNumber(42);
}});

presenter.render();
```

I like the last example better, because the description of the interaction of `presenter` with its collaborators is clearly written in one place. In the Mockito example we have that the interaction is described in two places, before and after the execution of `presenter.render()` and I find this is less clear.

Wait.... and the view?

We'd love to see the application running now, but there's a snag... where is the implementation of the `UnitDoctorView`? There are three things to remind us that we have yet to do this:

1. there's no way to see the application working without it
2. there's no way to pass the acceptance tests without it
3. there are still things left to do in our test list:
    - (DONE) Convert in to cm
    - (DONE) Convert F to C
    - (DONE) Report conversion not supported
    - Convert input number to double
    - Format output message

We pick "convert input number to double" from the list and write:

```
public class AndroidUnitDoctorViewTest extends AndroidTestCase {

  public void testReturnInputValues() throws Exception {
    EditText inputNumberField = new EditText(getContext());
    inputNumberField.setText("3.14159");

    AndroidUnitDoctorView view
        = new AndroidUnitDoctorView(inputNumberField);

    assertEquals(3.14159, view.inputNumber());
  }
```

Notes:

- We must interact with the elements of the user interface. Therefore this test needs to be in the "app" module.
- In the "app" module we must use JUnit 3, while in the "Core" module we can use JUnit 4
- In order to create an EditText we need an Android `Context`. The easiest way to get one is to extend `AndroidTestCase`.
- The name of the class is obtained by prefixing a qualifier "Android-" to the name of the interface. This is much better than using the "-Impl" suffix (bleah!) or adding an "I-" prefix to the interface name (also bleah!). So, `AndroidUnitDoctorView` means "the Android implementation of `UnitDoctorView`".
- The interface `UnitDoctorView` lives in the UnitDoctorCore module, while its implementation `AndroidUnitDoctorView` lives in the "app" module. This is correct: the interface talks exclusively in terms of the application *domain language*, so it belongs in the "core" module. Also, interfaces belong to their clients, not to their implementations, so it's OK that they live near the clients.

Making the above test pass is easy:

```
public class AndroidUnitDoctorView implements UnitDoctorView {
  private TextView inputNumberField;

  public AndroidUnitDoctorView(TextView inputNumberField) {
    this.inputNumberField = inputNumberField;
    this.fromUnitField = fromUnitField;
    this.toUnitField = toUnitField;
    this.resultField = resultField;
  }

  @Override
  public double inputNumber() {
    String inputString = inputNumberField.getText().toString();
    return Double.valueOf(inputString);
  }
  // ...
}
```

Next test: “Format output message”

```
public class AndroidUnitDoctorViewTest extends AndroidTestCase {

  EditText inputNumberField;
  TextView fromUnitField;
  TextView toUnitField;
  TextView resultField;
  AndroidUnitDoctorView view;

  @Override
  public void setUp() throws Exception {
    super.setUp();
    inputNumberField = new EditText(getContext());
    fromUnitField = new TextView(getContext());
    toUnitField = new TextView(getContext());
    resultField = new TextView(getContext());
    view = new AndroidUnitDoctorView(inputNumberField, fromUnitField, toUnitField, resultF\
ield);
  }

  public void testSetsResult() {
    inputNumberField.setText("3.14159");
    fromUnitField.setText("A");
    toUnitField.setText("B");

    view.showResult(1.123456789);
```

```
    assertEquals("3.14 A = 1.12 B", resultField.getText());
  }
```

Notes

- We extended the constructor of `AndroidUnitDoctorView` to accept all the UI elements it needs to talk to
- We moved creation of these elements to a shared `setUp` method

Making this pass is still easy:

```
public class AndroidUnitDoctorView implements UnitDoctorView {
  private TextView inputNumberField;
  private TextView fromUnitField;
  private TextView toUnitField;
  private TextView resultField;

  public AndroidUnitDoctorView(TextView inputNumberField, TextView fromUnitField, TextView\
 toUnitField, TextView resultField) {
    this.inputNumberField = inputNumberField;
    this.fromUnitField = fromUnitField;
    this.toUnitField = toUnitField;
    this.resultField = resultField;
  }

  @Override
  public void showResult(double result) {
    String message =
        String.format("%.2f %s = %.2f %s",
            inputNumber(), fromUnit(), result, toUnit());
    resultField.setText(message);
  }

  @Override
  public String fromUnit() {
    return fromUnitField.getText().toString();
  }

  @Override
  public String toUnit() {
    return toUnitField.getText().toString();
  }
  // ...
```

We still have to implement `UnitDoctorView.showConversionNotSupported()`. We write a test (not shown) and make it pass (also not shown, but see [Appendix: Unit Doctor]{#appendix-unit-doctor} for complete code listings.)

Now we are ready to see the app running, right? Are we there yet?

The *main partition*

Not so fast... we still haven't bound the UnitDoctor object and its view to the Android application. Even if we forget to do this, we'll be reminded because:

1. The acceptance tests still don't pass
2. We can run the application, but it does not convert anything yet.

We need a place to instantiate the UnitDoctor and its AndroidUnitDoctorView. This will be our "main function". Our "main function" is the first place where the Android O.S. gives control to our own code. In practice this is the `onCreate()` of `MainActivity`.

What do you mean by "main function"? Aren't we in Android? There is no "main" function here!
Simple Java applications start with a *main* method. The main method is where we build our objects, we combine them together forming a graph of communicating objects, and then we set them running by calling something like `run()` or `execute()`.

Alas, when we work with complex frameworks, such as Java Enterprise Edition or Android, we have no control over the real "main" method. The framework builds our objects for us, robbing us of the chance to customize them with the collaborators that we want. This is expecially severe in Android, where the O.S. creates all the important Android objects such as activities and services.

What's a TDDer to do then?
Not to worry: we are still in control. Just treat the activity's `onCreate()` method as if it was our main. It *is* our main. The trick is to use the activity just for building objects, linking them together appropriately, and letting them run. We keep the logic *out* of the activity, and implement all of the interesting stuff in our own objects, that are created by the activity.

How do I test an activity? How do I inject dependencies in an activity?
Normally we'd like to inject dependencies in an object via its constructor. But this is impossible to do to an activity, for the activity is created by the O.S. behind the scenes; so we can't customize the constructor for an activity. But this is not a problem if you follow the approach in this book, because

1. The activity is tested through the end-to-end acceptance tests.
2. The activity contains only construction and configuration, not logic.

So there is no need to test many cases: if the activity works in the ATs, it will probably work. We will not write unit tests for an activity.

But, but, but, ... what if I must have logic in an activity?
Keep it *out* of the activity, in a separate object. You test-drive that object; then you create that object and use it in the activity.

What is the "main partition"?
It is a set of code files that contain the "main" functions of our application, and the factories and the configurations. It's where all the objects of our application are created and assembled together. All the compile-time dependencies run from the main partition to the core of the application. The main partition is important because that is the place where all the configuration details are set up.

Our "main function" has the following responsibilities:

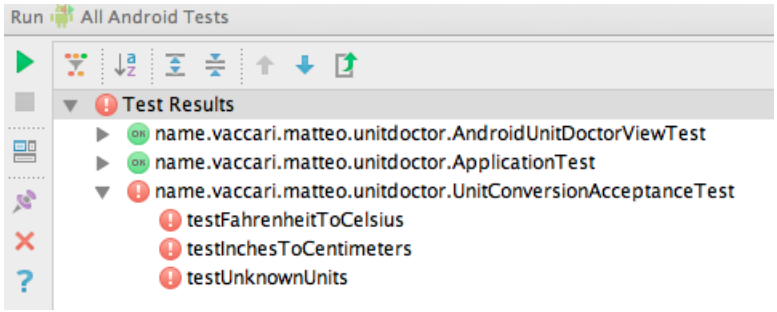
- Instantiate the UnitDoctor and its view
- Call the UnitDoctor whenever the user changes something

```
package name.vaccari.matteo.unitdoctor;

import android.app.Activity;
import android.os.Bundle;
import android.text.*;
import android.view.*;
import android.widget.TextView;

import name.vaccari.matteo.unitdoctor.core.UnitDoctor;


public class MainActivity extends Activity implements TextWatcher {
  private UnitDoctor doctor;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    TextView inputNumberField = (TextView) findViewById(R.id.inputNumber);
    TextView fromUnitField = (TextView) findViewById(R.id.fromUnit);
    TextView toUnitField = (TextView) findViewById(R.id.toUnit);
    TextView resultField = (TextView) findViewById(R.id.result);
    AndroidUnitDoctorView view = new AndroidUnitDoctorView(inputNumberField, fromUnitField\
, toUnitField, resultField);

    doctor = new UnitDoctor(view);
```

```
    inputNumberField.addTextChangedListener(this);
    fromUnitField.addTextChangedListener(this);
    toUnitField.addTextChangedListener(this);
  }

  @Override
  public void beforeTextChanged(CharSequence s, int start, int count, int after) {
  }

  @Override
  public void onTextChanged(CharSequence s, int start, int before, int count) {
    doctor.convert();
  }

  @Override
  public void afterTextChanged(Editable s) {
  }
}
```

We let `MainActivity` implement `TextWatcher` so that the `MainActivity` itself will listen for any change in the text fields.

Now that we have implemented our "main" we run all the Android tests, including the acceptance tests. And... they still fail!!!



Awww! The ATs still fail

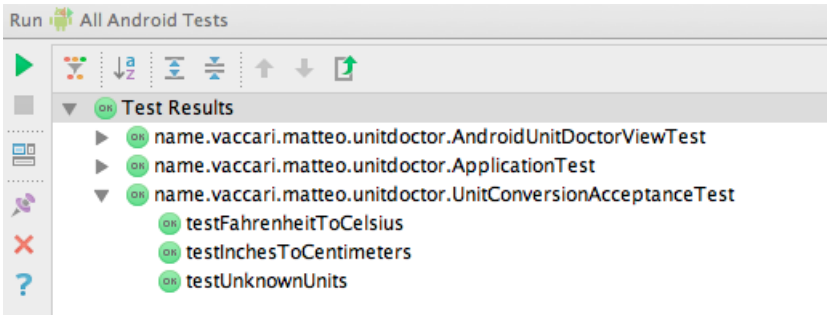
What happened? We have an exception:

```
java.lang.NumberFormatException: Invalid double: ""
at java.lang.StringToReal.invalidReal(StringToReal.java:63)
...
at name.vaccari.matteo.unitdoctor.AndroidUnitDoctorView.inputNumber(AndroidUnitDoctorView.\
java:27)
at name.vaccari.matteo.unitdoctor.core.UnitDoctor.convert(UnitDoctor.java:11)
at name.vaccari.matteo.unitdoctor.MainActivity.onTextChanged(MainActivity.java:39)
```

It seems we forgot to take care of the case when the inputNumber field contains an empty string. Oh well, the fix is simple: first we write a new unit test for `AndroidUnitDoctorView`:

```
public void testDoesNotBreakWhenInputFieldIsEmpty() throws Exception {
  inputNumberField.setText("");
  assertEquals(0.0, view.inputNumber());
}
```

We add the necessary IF to the production code, and then run again all the tests. And ... they all pass!!!



All the Android tests passing, finally!

Now that the ATs are passing, we fire the app on the emulator... and it works.

The app works

Shouldn't we also check for a non-numeric string?
Not really. Since we declared in the xml layout that the inputNumber field will only accept numeric input, this shouldn't happen.

The compile-time project structure

- Project UnitDoctor
  - Module app
    - Source folder “src/main/java”
      - Class `AndroidUnitDoctorView`
      - Class `MainActivity`
    - Source folder “src/androidTest/java”
      - Class `AndroidUnitDoctorViewTest`
      - Class `UnitConversionAcceptanceTest`
    - Source folder “src/main/res”
      - Layout `activity_main.xml`
  - Module UnitDoctorCore
    - Source folder “src/main/java”
      - Class `UnitDoctor`
      - Interface `UnitDoctorView`
    - Source folder “src/test/java”
      - Class `UnitDoctorTest`

What now?

We have started a project with tests. Now our options are open: we can improve the UI or we can improve the core functions of the application. The clean separation that we enforced on view and domain model means that we can add more conversion logic, with full confidence that the new logic will be “picked up” by the user interface.

8. Drawing on the screen and sensing touch

Digging deeper into Android

We've seen how to TDD an application based on ordinary form widgets: text fields and buttons. Now we'd like to go a bit deeper. How do we TDD an application that reacts to the user's touch and draws directly on the screen?

The goal of this chapter is to further understand how to decouple the tests from the Android APIs, even when we *need* to use those APIs. You may find the source code for this chapter in https://github.com/xpmatteo/fairy-fingers.

Problem Description

This problem was invented by Carlo Pescio for his Ribbons book. The user draws colored lines by dragging fingers on the screen. The lines fade quickly to nothing. We call our app "Fairy Fingers".

Start with a spike

The goals of the spike are:

- Understand how to draw on the screen
- Understand how to track the user's finger

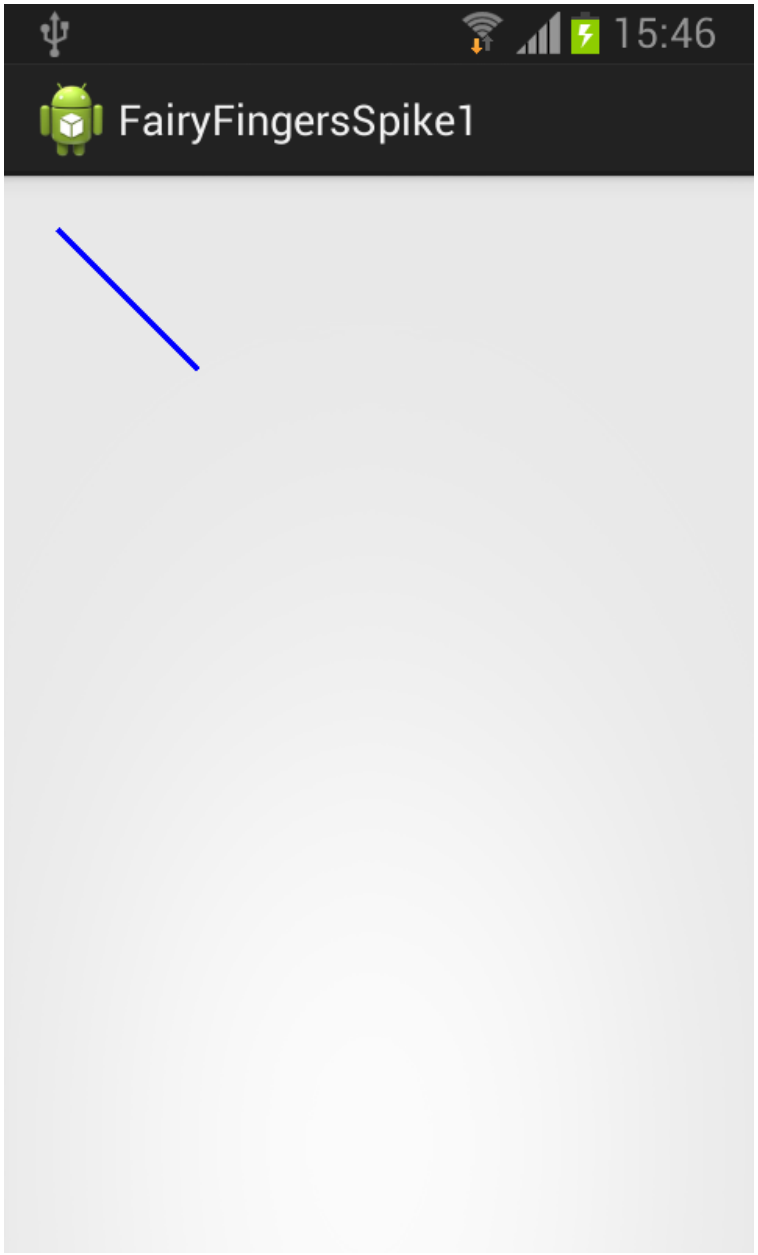
We create an empty project and check that it shows a "hello world" on our device. Then we modify the `res/layout/activity_my.xml` file, removing the standard "hello world" view and replacing it with a custom view (see line 7 in next listing).

```
1  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
2      xmlns:tools="http://schemas.android.com/tools"
3      android:layout_width="match_parent"
4      android:layout_height="match_parent"
5      tools:context=".MyActivity">
6
7      <com.tdd4android.fairyfingers.spike.MyView
8          android:layout_width="wrap_content"
9          android:layout_height="wrap_content" />
10 </RelativeLayout>
```

The IDE complains that the view does not exist, so we go ahead and create it. We need to override the constructors. I don't know which ones are really needed, so I override them all. Then we need to override the `OnDraw` method. We want to see if we can paint on the screen at all, so we draw a single line.

```
package com.tdd4android.fairyfingers.spike;

import android.content.Context;
import android.graphics.*;
import android.util.AttributeSet;
import android.view.View;

public class MyView extends View {
  public MyView(Context context) {
    super(context);
  }
  public MyView(Context context, AttributeSet attrs) {
    super(context, attrs);
  }
  public MyView(Context context, AttributeSet attrs, int defStyleAttr) {
    super(context, attrs, defStyleAttr);
  }

  @Override
  protected void onDraw(Canvas canvas) {
    Paint paint = new Paint();
    paint.setColor(Color.BLUE);
    paint.setStrokeWidth(3);
    canvas.drawLine(10, 10, 100, 100, paint);
  }
}
```

We run it, and it works!

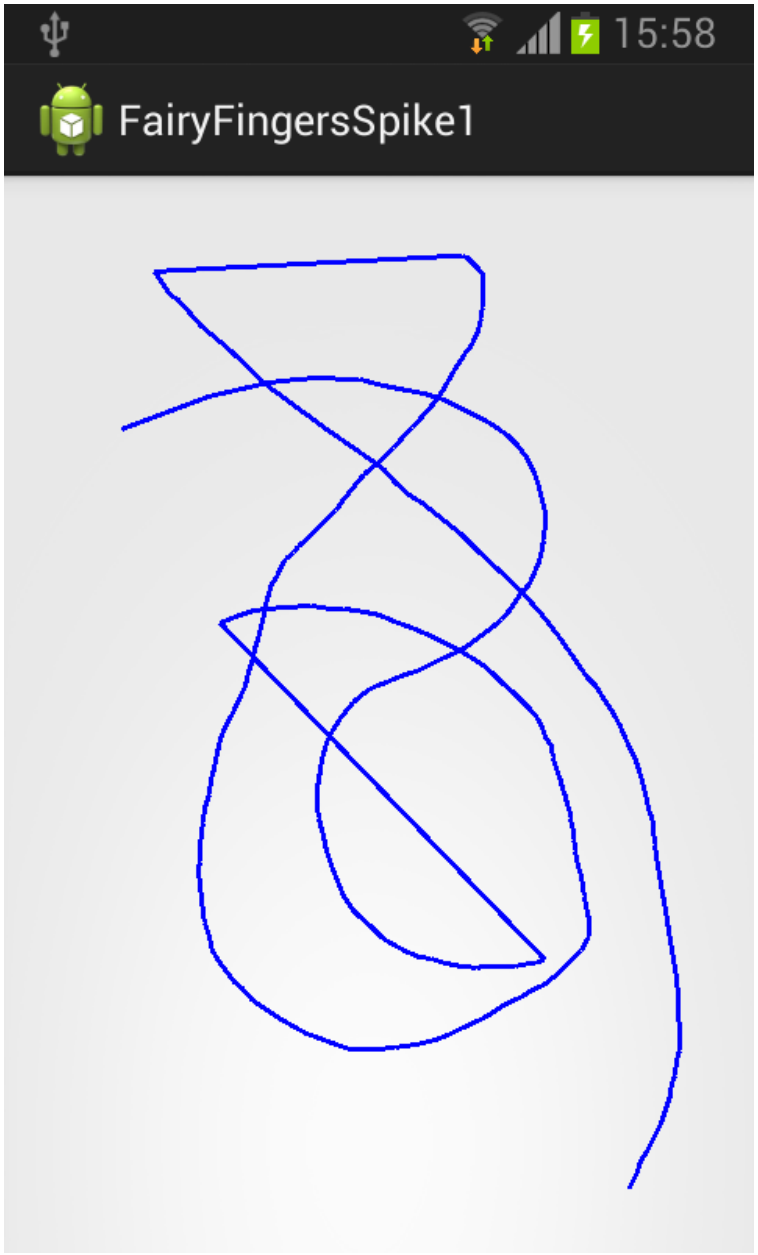


We can draw a line!

Now we want to track the user touching the screen. Android calls the `onTouchEvent` method whenever the user touches or drags her finger on the screen. It gives us a `MotionEvent` object that contains the screen coordinates of the user's finger. We must store those coordinates in a list and then draw them in the `onDraw` method. We must also remember to call `invalidate` after every touch event. This notifies the OS that the screen should be updated; Android will answer by calling `onDraw` later.

```
public class MyView extends View {
  private List<Point> points = new ArrayList<Point>();

  // ...

  @Override
  protected void onDraw(Canvas canvas) {
    Paint paint = new Paint();
    paint.setColor(Color.BLUE);
    paint.setStrokeWidth(3);
    for (int i=1; i<points.size(); i++) {
      Point from = points.get(i-1);
      Point to = points.get(i);
      canvas.drawLine(from.x, from.y, to.x, to.y, paint);
    }
  }

  @Override
  public boolean onTouchEvent(MotionEvent event) {
    points.add(new Point((int) event.getX(), (int) event.getY()));
    invalidate();
    return true;
  }
}
```

We drag a finger on the screen and we see the following.



We can track the user's finger

At the moment, we draw a single line. Touching again the screen makes the line longer. The next thing we'd like to do is to draw a new line every time we touch the screen.

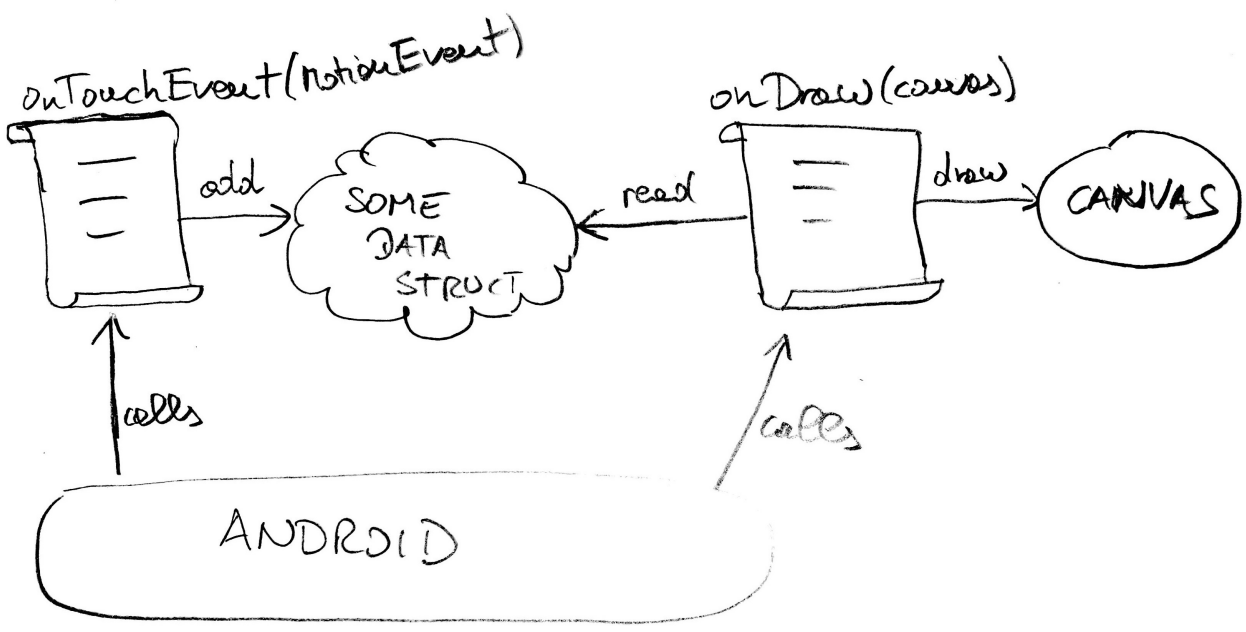
The key here is how we use the `MotionEvent` data structure. The relevant fields return the event coordinates with `getX` and `getY`, and the type of action, that is returned by `getActionMasked`. The actions we are interested in at the moment are `ACTION_DOWN`, `ACTION_MOVE` and `ACTION_UP`, corresponding to touching, dragging and lifting the finger. We can exploit this information to clear the old line whenever the finger goes down on the screen.

```
public boolean onTouchEvent(MotionEvent event) {
  int action = event.getActionMasked();
  if (action == MotionEvent.ACTION_DOWN) {
    points.clear();
    points.add(new Point((int) event.getX(), (int) event.getY()));
  } else if (action == MotionEvent.ACTION_MOVE) {
    points.add(new Point((int) event.getX(), (int) event.getY()));
  }
  invalidate();
  return true;
}
```

So far, so good. We could explore the `MotionEvent` further in order to understand multitouch, but we can leave that for later. We learned enough already for writing a first version of Fairy Fingers that only supports single touch.

What have we learned? There are two points where we interact with the device.

1. When the user touches the screen, we receive data through the `onTouchEvent(MotionEvent e)` call. We should accumulate these data in some kind of data structure.
2. When the screen is being drawn, Android calls `onDraw(Canvas c)`. We should use our previously created data structure to know what we should draw on the canvas.



How FairyFingers interacts with the device

Acceptance tests

We start with the following acceptance tests:

- Single touch. Dragging the finger should produce a colored trail
- Fade. The trails should fade to nothing.
- Many trails. We draw a trail, then we draw another.
- Many colours. Every time we draw a trail, we should get a different colour
- Multi-touch. Dragging two fingers should produce two trails.
- Multi-touch dashes. We draw a continuous trail with one finger, and a dashed trail with another finger at the same time. We should see a pattern like

```
--   --   --
------------
```

These acceptance tests are meant to be executed manually. Some can and will be automated. The ones that deal with multi-touch cannot be automated with present-generation tools (Monkeyrunner).

Note that we started using the word "trail" instead of "line". We'd like to have a special name for the "things" that we draw with in this app, and the word "line" is both too generic and too specific. The Android `canvas` object has a `drawLine` method; we'd like to distinguish our own "lines" from what Android calls a "line". Therefore we will call them "trails".

Setup the project

We create a new project for Fairy Fingers. We start much like we did in the spike; we create a custom view. Then we add a new `Core` module that will contain the pure Java code, as usual.

Our entry point will be in methods `onDraw()` and `onTouchEvent()` of the `FairyFingersView`. We will delegate most of the work to a `FairyFingersCore` object. We imagine that we will have something like the following pseudo-code:

```
private FairyFingersCore core = new FairyFingersCore();

@Override
protected void onDraw(Canvas canvas) {
  for every trail in core {
    draw the trail on the canvas
  }
}

@Override
public boolean onTouchEvent(MotionEvent event) {
  core.onTouchEvent(event.getActionMasked(), event.getX(), event.getY());
  invalidate();
  return true;
}
```

Start the TDD

The first step for TDD is to write a test list. We start by writing a todo list from the acceptance tests list:

- create a two-points trail
- create a many-points trail
- create two trails
- fade a trail
- randomize colours
- draw all the trails

Where do we start? We choose "create a two points trail" because we'd like to discover how we will solve this. Now we can write the first test.

```
package com.tdd4android.fairyfingers.core;

import org.junit.Test;
import static org.junit.Assert.*;

public class FairyFingersCoreTest {
  // Constants copied from android.view.MotionEvent
  public static final int ACTION_DOWN             = 0;
  public static final int ACTION_UP               = 1;
  public static final int ACTION_MOVE             = 2;

  @Test
  public void twoPointsTrail() throws Exception {
    FairyFingersCore core = new FairyFingersCore();

    core.onMotionEvent(ACTION_DOWN, 10, 20);
    core.onMotionEvent(ACTION_MOVE, 30, 40);
    core.onMotionEvent(ACTION_UP, 50, 60);

    assertEquals(1, core.trailsCount());
  }
}
```

While we write this test, we think of a two simpler ones:

```
@Test
public void noTrails() throws Exception {
  FairyFingersCore core = new FairyFingersCore();
  assertEquals(0, core.trailsCount());
}
```

and

```
@Test
public void unfinishedTrail() throws Exception {
  FairyFingersCore core = new FairyFingersCore();

  core.onMotionEvent(ACTION_DOWN, 10, 20);
  core.onMotionEvent(ACTION_MOVE, 30, 40);

  assertEquals(1, core.trailsCount());
}
```

The last one is needed because we expect the trail to be visible even while it's not finished yet.

The first implementation of `FairyFingersCore` makes these three tests pass, but is not very useful yet.

```
package com.tdd4android.fairyfingers.core;

public class FairyFingersCore {
  private int trailsCount;

  public int trailsCount() {
    return trailsCount;
  }

  public void onMotionEvent(int action, float x, float y) {
    if (action == ACTION_DOWN)
      trailsCount++;
  }
}
```

The problem is in the assertions. The assertion on the `trailsCount()` by itself is not very useful. It does not prove that the important data about the trail have been memorized. How can we improve the tests in a way that forces us to flesh out `FairyFingersCore` better?

(Pause for a minute. What would YOU do?)

* * *

The FairyFingersCore should build a Trail object, and we'd like this Trail to contain exactly the coordinates that were supplied by the tests. One way to do this is with getters:

```
assertEquals(10, core.getTrail(0).getPoints(0).getX());
assertEquals(20, core.getTrail(0).getPoints(0).getY());
assertEquals(30, core.getTrail(0).getPoints(1).getX());
assertEquals(40, core.getTrail(0).getPoints(1).getY());
```

But this test code is extremely boring to write! Being bored is an important signal. It's the test pushing back: it doesn't want to be written like this. One concrete problem is that there are too many "dots" in the assertion. We dig too much further into the objects. One other problem is that we are assuming that we will need all those getters; it's not clear yet that these getters will be used in production code.

Trick: use "toString". The `toString` of the `Trail` will certainly be needed for debugging and logging. How about:

```
assertEquals("(10,20)->(30,40)", core.getTrail(0).toString());
```

In our test, we are only assuming that the `FairyFingersCore` will return an object that somehow contains the expected points in a certain order. We don't even assert what kind of object it is. It looks promising! The tests are rewritten in the following style:

```
private FairyFingersCore core = new FairyFingersCore();

@Test
public void justFingerDown() throws Exception {
  core.onMotionEvent(ACTION_DOWN, 10, 20);

  assertEquals(1, core.trailsCount());
  assertEquals("(10.0,20.0)", core.getTrail(0).toString());
}

@Test
public void unfinishedTrail() throws Exception {
  core.onMotionEvent(ACTION_DOWN, 100, 200);
  core.onMotionEvent(ACTION_MOVE, 300, 400);

  assertEquals(1, core.trailsCount());
  assertEquals("(100.0,200.0)->(300.0,400.0)", core.getTrail(0).toString());
}

@Test
public void aFinishedTrail() throws Exception {
  core.onMotionEvent(ACTION_DOWN, 1.1, 2.2);
  core.onMotionEvent(ACTION_MOVE, 3.33, 4.44);
  core.onMotionEvent(ACTION_UP, 5.555, 6.666);

  assertEquals(1, core.trailsCount());
```

```
  assertEquals("(1.1,2.2)->(3.33,4.44)->(5.555,6.666)", core.getTrail(0).toString());
}
```

We moved the FairyFingersCore object in a field, in order to remove the duplication of creating it in every test. We could use a `@Before` method to initialize it, but doing it this way is shorter. We rely on the fact that JUnit creates a new `FairyFingersCoreTest` object for every test method it calls. Therefore, every test has a fresh `FairyFingersCore` object.

Also note that we always use different numbers in the tests. If we wrote `core.onMotionEvent(ACTION_DOWN, 10, 10)` we would be open to the risk of swapping x and y in our production code.

We avoid using the same test data in multiple tests. Using always different data prevents us to leave hardcoded test data in production code!

One last observation: Android uses the `float` data type for coordinates. Our code should do the same.

9. More Android

Persisting data

Saving application data.

TSTTCPW is saving to preferences

The next simplest thing is text files

Cloud!

App life-cycle

How to suspend and resume the application safely

10. References

Steve Freeman and Nat Pryce, *Growing Object Oriented Software, Guided By Tests*, http://www.growing-object-oriented-software.com/

Kent Beck, *Test Driven Development: By Example*, Addison-Wesley, 2001

Kent Beck, *Test Driven Development with Kent Beck*, https://pragprog.com/screencasts/v-kbtdd/test-driven-development

J.B. Rainsberger, *The World's Best Intro to TDD*, http://www.jbrains.ca/permalink/the-worlds-best-intro-to-tdd-demo-video

J.B. Rainsberger, *Responsible Design For Android*, https://leanpub.com/ResponsibleDesignAndroid-Part1

The UnitDoctor repository: https://github.com/xpmatteo/unit-doctor

The FairyFingers repository: https://github.com/xpmatteo/fairy-fingers