Writing truly testable code

Anton Rutkevich
Juno

Why tests?

- Codebase grows
  -> **chances** of making a mistake **go up**
- Software gets released
  -> **cost** of a mistake **goes up**

-> Fear of introducing modifications!

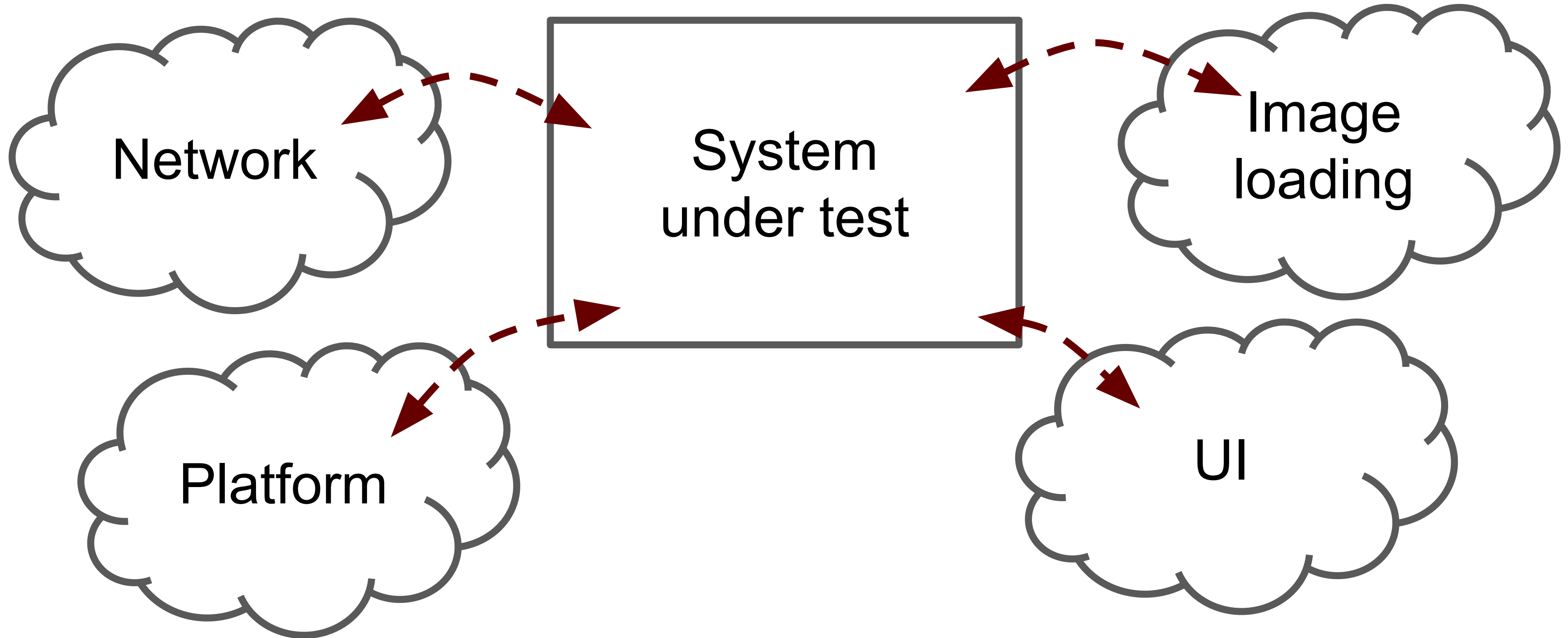
The two major issues

- Managing system complexity
  -> Rich Hickey **"Simple Made Easy"**
- Testing the system
  -> this talk

Our first test

Our first test attempt

What could possibly go wrong?



Agenda

- Truly testable is ...
- 3 rules of truly testable code
- Testability in practice
- How to start?

Truly testable is…

What is a test?

PASS

VALIDATE

Truly testable is…

all
inputs and outputs
are under control

3 rules of truly testable code

Rule #1

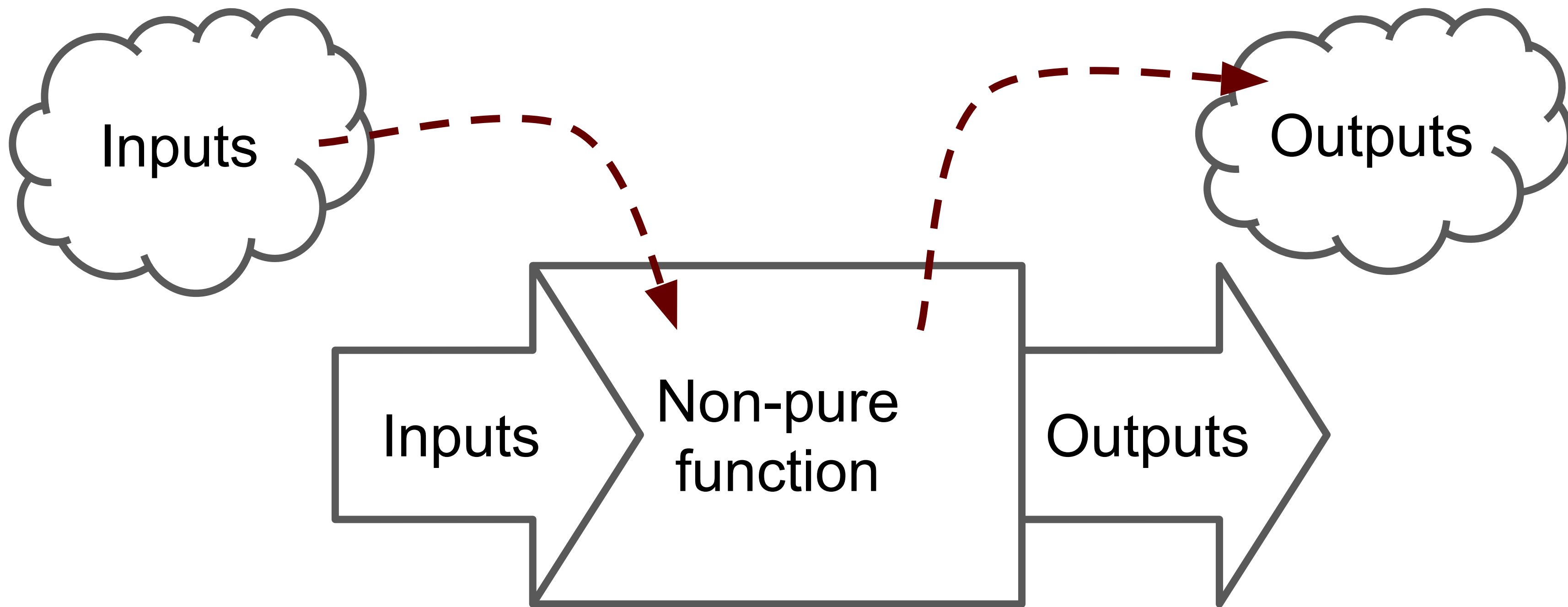
Testing a function

```
f(Arg[1], … , Arg[N]) -> (R[1], … , R[L])
```

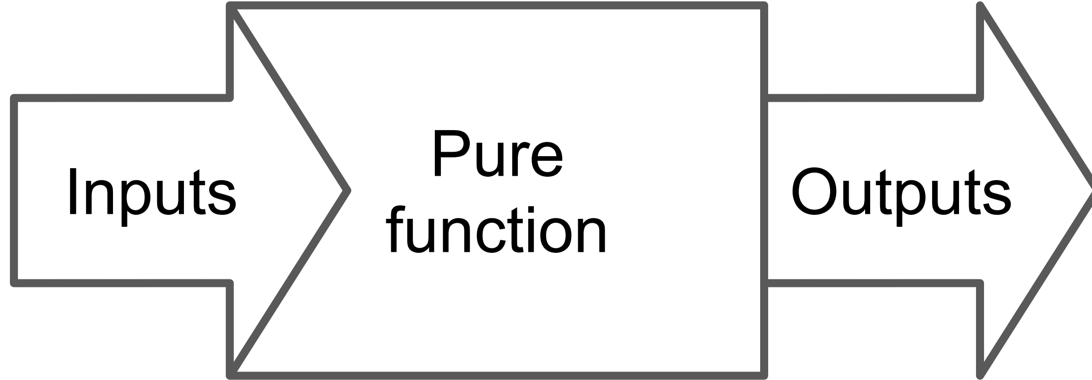
A non-pure function

```kotlin
fun nextItemDescription(prefix: String): String {
    GLOBAL_VARIABLE++
    return "$prefix: $GLOBAL_VARIABLE"
}
```

A non-pure function

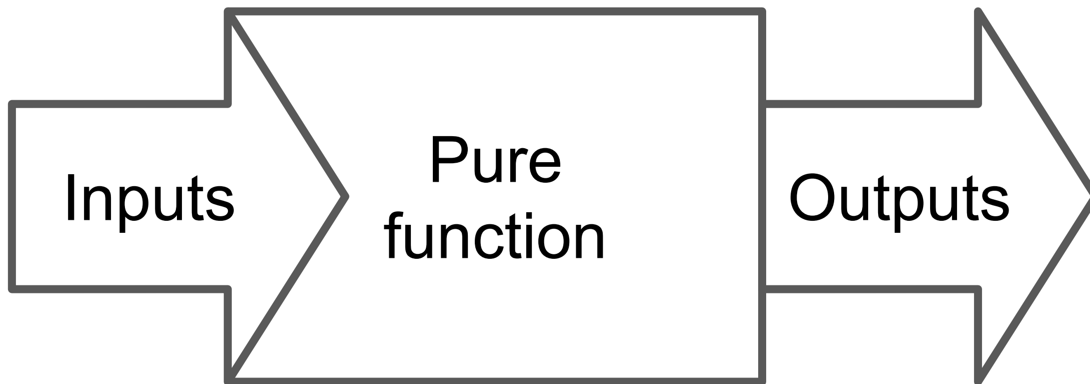


A pure function



A pure function

```
fun itemDescription(prefix: String, itemIndex: Int): String {
   return "$prefix: $itemIndex"
}
```



A function is easy to test if

```
f(Arg[1], … , Arg[N]) -> (R[1], … , R[L])
```

`Arg[i]` and **`R[i]`** are explicit: passed through parameters and returned as results

Rule #1

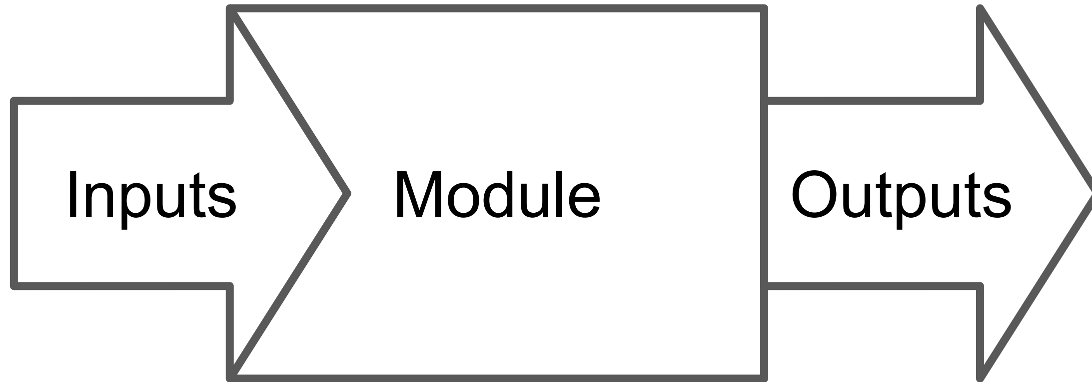
Pass
arguments & results
explicitly

Rule #2

Testing a module

```
M(In[1], … , In[N]) -> (Out[1], … , Out[L])
```

Testing a module



Module inputs

```
class Module(
        val title: String, // input

) {




}
```

Module inputs

```
class Module(
      val title: String, // input

) {
   fun doSomething() {    // input



      // ...
   }
}
```

Module inputs

```
class Module(
        val title: String, // input
        val dependency: Explicit   // dependency
) {
    fun doSomething() {    // input
        val explicit = dependency.getCurrentState()  // input


        // ...
    }
}
```

Module inputs

```
class Module(
        val title: String, // input
        val dependency: Explicit   // dependency
) {
    fun doSomething() {    // input
        val explicit = dependency.getCurrentState()  // input
        val implicit = Implicit.getCurrentState()    // input

        // ...
    }
}
```

Module inputs

```
class Module(
        val title: String, // input
        val dependency: Explicit   // dependency
) {
    fun doSomething() {    // input
        val explicit = dependency.getCurrentState()  // input
        val implicit = Implicit.getCurrentState()    // input

        // ...
    }
}
```

Module outputs

```
class Module(

) {
    var state = "Some state"

    fun doSomething() {
        state = "New state"                    // output




        // ...
    }
}
```

Module outputs

```
class Module(
        val dependency: Explicit    // dependency
) {
    var state = "Some state"

    fun doSomething() {
        state = "New state"                          // output

        dependency.setCurrentState("New state") // output

        // ...
    }
}
```

Module outputs

```
class Module(
      val dependency: Explicit   // dependency
) {
   var state = "Some state"

   fun doSomething() {
      state = "New state"                  // output

      dependency.setCurrentState("New state") // output
      Implicit.setCurrentState("New state")   // output
      // ...
   }
}
```

Module outputs

```
class Module(
        val dependency: Explicit   // dependency
) {
    var state = "Some state"

    fun doSomething() {
        state = "New state"                      // output

        dependency.setCurrentState("New state") // output
        Implicit.setCurrentState("New state")   // output
        // ...
    }
}
```

Module inputs

```
In[1], … , In[N]
```

- **Interactions** with module API and dependencies API
- **Values** passed through the interactions
- **Order** of the interactions
- **Timings** between the interactions

Module outputs

```
Out[1], … , Out[N]
```

- **Interactions** with module API and dependencies API
- **Values** passed through the interactions
- **Order** of the interactions
- **Timings** between the interactions
- Modifications of the module **state**

Testing a module

```
M(In[1], … , In[N]) -> (Out[1], … , Out[L])
   given, when               then
```

Test = call of the function + validation of the outputs

A module is easy to test if

```
M(In[1], … , In[N]) -> (Out[1], … , Out[L])
```

`In[i]` and `Out[i]` are passed through
module API or explicit dependencies API.

Rule #2

Pass dependencies explicitly

Rule #3

Explicit arguments & dependencies
==
full control ?

Explicit dependency test

```
class Module(explicit: Explicit) {
    val tripled = 3 * explicit.getValue()
}
```

Explicit dependency test

```
class Module(explicit: Explicit) {
    val tripled = 3 * explicit.getValue()
}

@Test
fun testValueGetsTripled() {




}
```

Explicit dependency test

```
class Module(explicit: Explicit) {
    val tripled = 3 * explicit.getValue()
}

@Test
fun testValueGetsTripled() {
    // prepare Explicit dependency

    val result = Module( ??? ).tripled


}
```

Explicit dependency test

```
class Module(explicit: Explicit) {
    val tripled = 3 * explicit.getValue()
}

@Test
fun testValueGetsTripled() {
    // prepare Explicit dependency

    val result = Module( ??? ).tripled
    val expected = 15

}
```

Explicit dependency test

```
class Module(explicit: Explicit) {
    val tripled = 3 * explicit.getValue()
}

@Test
fun testValueGetsTripled() {
    // prepare Explicit dependency

    val result = Module( ??? ).tripled
    val expected = 15
    assertThat(result).isEqualTo(expected)
}
```

Explicit dependency test

```
class Module(explicit: Explicit) {
    val tripled = 3 * explicit.getValue()
}

@Test
fun testValueGetsTripled() {
    // prepare Explicit dependency

    val result = Module( ??? ).tripled
    val expected = 15
    assertThat(result).isEqualTo(expected)
}
```

‘Explicit’ as a Singleton

```
// ‘object’ stands for Singleton in Kotlin
object Explicit {
    fun getValue(): Int = ...
}
```

‘Explicit’ as a Singleton

```
// ‘object’ stands for Singleton in Kotlin
object Explicit {
    fun getValue(): Int = ...
}

@Test
fun testValueGetsTripled() {
    val result = Module( ??? ).tripled
    val expected = 15
    assertThat(result).isEqualTo(expected)
}
```

‘Explicit’ as a final class

```
// Classes are final by default in Kotlin
class Explicit {
   fun getValue(): Int = ...
}
```

‘Explicit’ as a final class

```
// Classes are final by default in Kotlin
class Explicit {
   fun getValue(): Int = ...
}

@Test
fun testValueGetsTripled() {
    val result = Module( ??? ).tripled
    val expected = 15
    assertThat(result).isEqualTo(expected)
}
```

‘Explicit’ as an Interface

```
interface Explicit {
    fun getValue(): Int

    class Impl: Explicit {
        override fun getValue(): Int = ...
    }
}
```

‘Explicit’ as an Interface

```
@Test
fun testValueGetsTripled() {
    // prepare Explicit dependency




    val result = Module( ??? ).tripled
    val expected = 15
    assertThat(result).isEqualTo(expected)
}
```

‘Explicit’ as an Interface

```kotlin
@Test
fun testValueGetsTripled() {
    val mockedExplicit = object : Explicit {
        override fun getValue(): Int = 5
    }

    val result = Module(mockedExplicit).tripled
    val expected = 15
    assertThat(result).isEqualTo(expected)
}
```

‘Explicit’ as an Interface

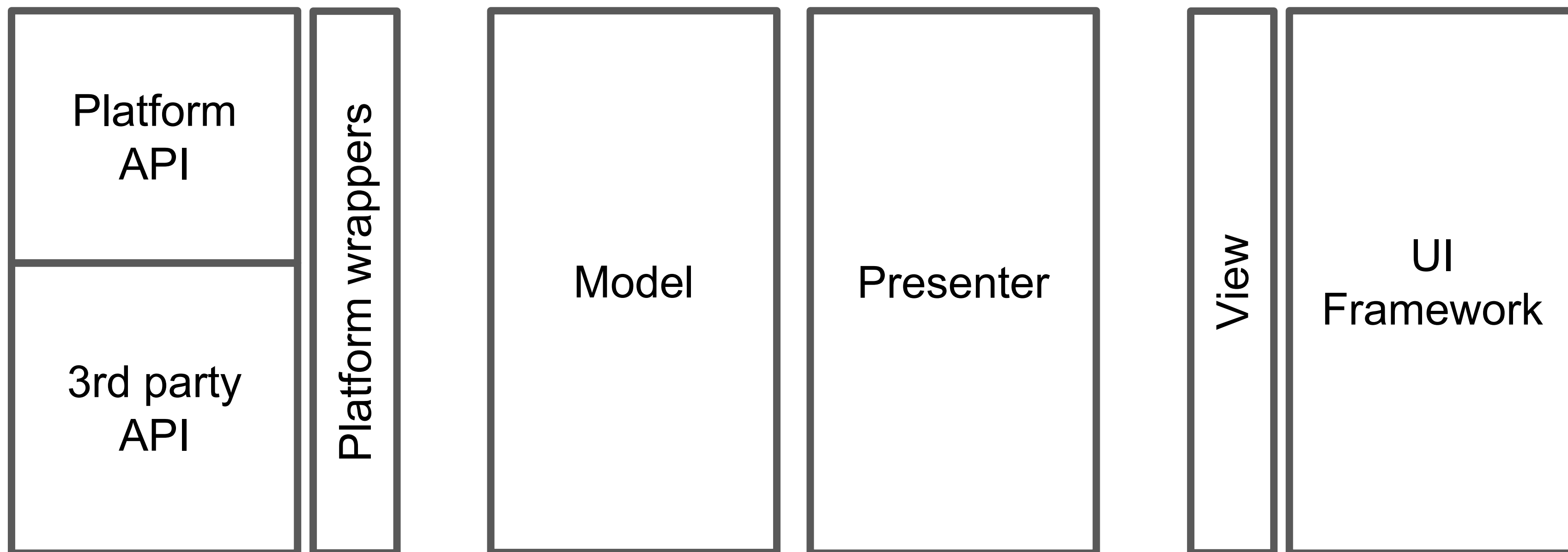
```
@Test
fun testValueGetsTripled() {
    val mockedExplicit = object : Explicit {
        override fun getValue(): Int = 5
    }

    val result = Module(mockedExplicit).tripled
    val expected = 15
    assertThat(result).isEqualTo(expected)
}
```

Rule #3

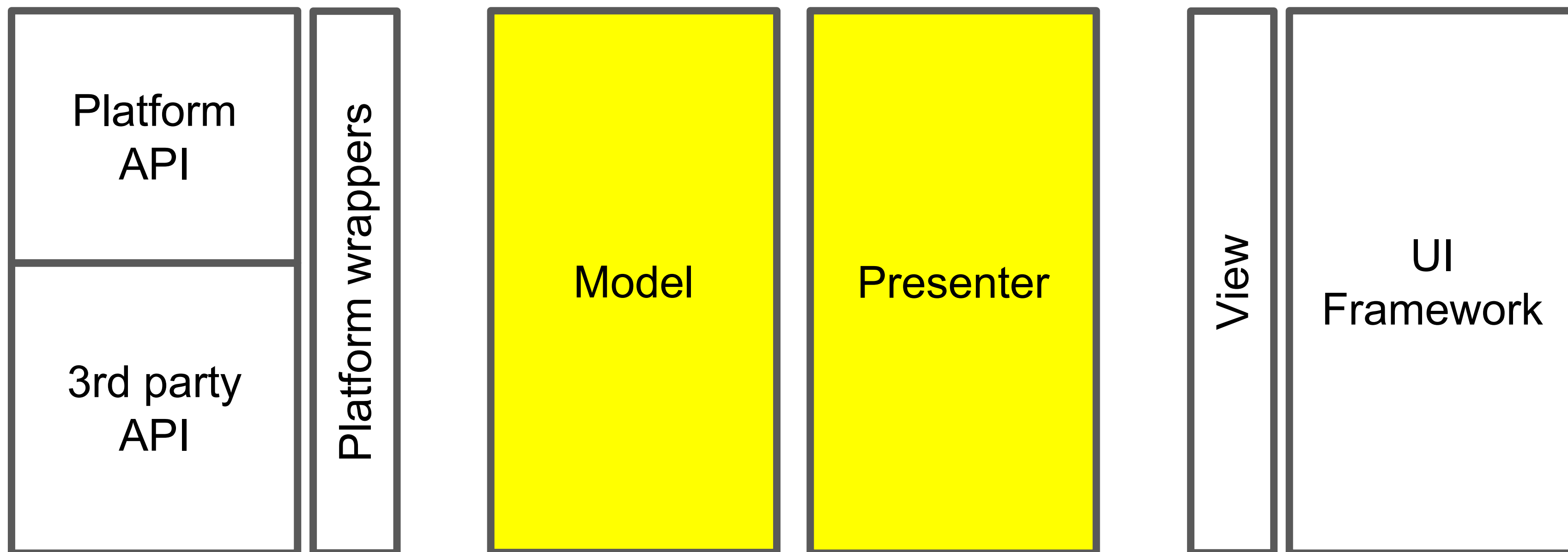
Make sure you can mock dependencies

Testability in practice

MVP. Overview



MVP. Overview



Practical aspects

1. Understand **explicits**
2. Locate **implicits** and convert them to **explicits**
3. Mock dependencies in tests
4. Abstract away the platform
5. Consider extracting implementation details

Testability in practice.

Understanding explicits

Explicit inputs

```
class ModuleInputs(
       input: String,




) {




}
```

Explicit inputs

```
class ModuleInputs(
        input: String,
        inputLambda: () -> String,


) {



}
```

Explicit inputs

```
class ModuleInputs(
        input: String,
        inputLambda: () -> String,
        inputObservable: Observable<String>,

) {



}
```

Explicit inputs

```
class ModuleInputs(
        input: String,
        inputLambda: () -> String,
        inputObservable: Observable<String>,

) {


   fun passInput(input: String) { }
}
```

Explicit inputs

```
class ModuleInputs(
        input: String,
        inputLambda: () -> String,
        inputObservable: Observable<String>,
        dependency: Explicit
) {
    private val someField = dependency.getInput()

    fun passInput(input: String) { }
}
```

Explicit inputs

```
class ModuleInputs(
        input: String,
        inputLambda: () -> String,
        inputObservable: Observable<String>,
        dependency: Explicit
) {
    private val someField = dependency.getInput()

    fun passInput(input: String) { }
}
```

Explicit outputs

```
class ModuleOutputs(


) {

    fun getOutput(): String = "Output"




}
```

Explicit outputs

```
class ModuleOutputs(
        outputLambda: (String) -> Unit,

) {

    fun getOutput(): String = "Output"

    init {
        outputLambda("Output")

    }
}
```

Explicit outputs

```
class ModuleOutputs(
        outputLambda: (String) -> Unit,

) {
    val outputObservable = Observable.just("Output")
    fun getOutput(): String = "Output"

    init {
        outputLambda("Output")

    }
}
```

Explicit outputs

```
class ModuleOutputs(
        outputLambda: (String) -> Unit,
        dependency: Explicit
) {
    val outputObservable = Observable.just("Output")
    fun getOutput(): String = "Output"

    init {
        outputLambda("Output")
        dependency.passOutput("Output")
    }
}
```

Explicit outputs

```
class ModuleOutputs(
        outputLambda: (String) -> Unit,
        dependency: Explicit
) {
    val outputObservable = Observable.just("Output")
    fun getOutput(): String = "Output"

    init {
        outputLambda("Output")
        dependency.passOutput("Output")
    }
}
```

Testability in practice.

Top 5 Implicits

#5: Statics and Singletons

```
class Module {
   private val state = Implicit.getCurrentState()
}
```

#5: Statics and Singletons

```
class Module(dependency: Explicit) {
    private val state = dependency.getCurrentState()
}
```

#5: Statics and Singletons

```
class Module(dependency: Explicit) {
   private val state = dependency.getCurrentState()
}
```

#4: Random generators

```
class Module {
   private val fileName = "some-file${Random().nextInt()}"
}
```

#4: Random generators

```
class Module(rng: Rng) {
   private val fileName = "some-file${rng.nextInt()}"
}
```

#4: Random generators

```
class Module(rng: Rng) {
   private val fileName = "some-file${rng.nextInt()}"
}
```

#3: File system & other storage

```
class Module {
   fun initStorage(path: String) {
          File(path).createNewFile()







   }
}
```

#3: File system & other storage

```
class Module {
    fun initStorage(path: String): FileCreationError? {
        return if (File(path).createNewFile()) {
            null
        } else {
            FileCreationError.Exists
        }




    }
}
```

#3: File system & other storage

```
class Module {
   fun initStorage(path: String): FileCreationError? = try {
       if (File(path).createNewFile()) {
           null
       } else {
           FileCreationError.Exists
       }
   } catch (e: SecurityException) {
       FileCreationError.Security(e)
   } catch (e: Exception) {
       FileCreationError.Other(e)
   }
}
```

#3: File system & other storage

```kotlin
class Module {
   fun initStorage(path: String): FileCreationError? = try {
       if (File(path).createNewFile()) {
           null
       } else {
           FileCreationError.Exists
       }
   } catch (e: SecurityException) {
       FileCreationError.Security(e)
   } catch (e: Exception) {
       FileCreationError.Other(e)
   }
}
```

#3: File system & other storage

```
class Module(private val fileCreator: FileCreator) {
    fun initStorage(path: String): FileCreationError? = try {
        if (fileCreator.createNewFile(path)) {
            null
        } else {
            FileCreationError.Exists
        }
    } catch (e: SecurityException) {
        FileCreationError.Security(e)
    } catch (e: Exception) {
        FileCreationError.Other(e)
    }
}
```

#3: File system & other storage

```kotlin
class Module(private val fileCreator: FileCreator) {
    fun initStorage(path: String): FileCreationError? = try {
        if (fileCreator.createNewFile(path)) {
            null
        } else {
            FileCreationError.Exists
        }
    } catch (e: SecurityException) {
        FileCreationError.Security(e)
    } catch (e: Exception) {
        FileCreationError.Other(e)
    }
}
```

#2: Time

```
class Module {
    private val nowTime = System.currentTimeMillis()
    private val nowDate = Date()
    // and all other time/date APIs
}
```

#2: Time

```
class Module(time: TimeProvider) {
    private val nowTime = time.nowMillis()
    private val nowDate = time.nowDate()
    // and all other time/date APIs
}
```

#2: Time

```
class Module(time: TimeProvider) {
    private val nowTime = time.nowMillis()
    private val nowDate = time.nowDate()
    // and all other time/date APIs
}
```

#1: Formatting & Locales

```
class MyTimePresenter(timestamp: Long) {
   val formattedTimestamp = SimpleDateFormat("yyyy-MM-dd HH:mm")
           .format(timestamp)
}
```

#1: Formatting & Locales

```
class MyTimePresenter(timestamp: Long) {
   val formattedTimestamp = SimpleDateFormat("yyyy-MM-dd HH:mm")
           .format(timestamp)
}

fun test() {




}
```

#1: Formatting & Locales

```
class MyTimePresenter(timestamp: Long) {
    val formattedTimestamp = SimpleDateFormat("yyyy-MM-dd HH:mm")
            .format(timestamp)
}

fun test() {
    val mobiusConfStart = 1492758000L



}
```

#1: Formatting & Locales

```
class MyTimePresenter(timestamp: Long) {
    val formattedTimestamp = SimpleDateFormat("yyyy-MM-dd HH:mm")
            .format(timestamp)
}

fun test() {
    val mobiusConfStart = 1492758000L
    val expected = ""
    val actual = MyTimePresenter(timestamp).formattedTimestamp
    assertThat(actual).isEqualTo(expected)
}
```

#1: Formatting & Locales

```
class MyTimePresenter(timestamp: Long) {
   val formattedTimestamp = SimpleDateFormat("yyyy-MM-dd HH:mm")
           .format(timestamp)
}

fun test() {
   val mobiusConfStart = 1492758000L
   val expected = "2017-04-21 10:00"
   val actual = MyTimePresenter(timestamp).formattedTimestamp
   assertThat(actual).isEqualTo(expected)
}
```

#1: Formatting & Locales

```
class MyTimePresenter(timestamp: Long) {
   val formattedTimestamp = SimpleDateFormat("yyyy-MM-dd HH:mm")
           .format(timestamp)
}

fun test() {
   val mobiusConfStart = 1492758000L
   val expected = "2017-04-21 10:00"
   val actual = MyTimePresenter(timestamp).formattedTimestamp
   assertThat(actual).isEqualTo(expected)
}
>> `actual on dev machine` = "2017-04-21 10:00"
```

#1: Formatting & Locales

```
class MyTimePresenter(timestamp: Long) {
   val formattedTimestamp = SimpleDateFormat("yyyy-MM-dd HH:mm")
           .format(timestamp)
}

fun test() {
   val mobiusConfStart = 1492758000L
   val expected = "2017-04-21 10:00"
   val actual = MyTimePresenter(timestamp).formattedTimestamp
   assertThat(actual).isEqualTo(expected)
}
>> `actual on dev machine` = "2017-04-21 10:00"
>> `actual on CI`          = "2017-04-21 07:00"
```

#1: Formatting & Locales

```
class MyTimePresenter(timestamp: Long) {
   val formattedTimestamp = SimpleDateFormat("yyyy-MM-dd HH:mm")
           .format(timestamp)
}

fun test() {
   val mobiusConfStart = 1492758000L
   val expected = "2017-04-21 10:00"
   val actual = MyTimePresenter(timestamp).formattedTimestamp
   assertThat(actual).isEqualTo(expected)
}
>> `actual on dev machine` = "2017-04-21 10:00" // UTC +3
>> `actual on CI`          = "2017-04-21 07:00" // UTC
```

#1: Formatting & Locales

```
class MyTimePresenter(timestamp: Long) {
   val formattedTimestamp = SimpleDateFormat("yyyy-MM-dd HH:mm")
           .format(timestamp)
}

fun test() {
   val mobiusConfStart = 1492758000L
   val expected = "2017-04-21 10:00"
   val actual = MyTimePresenter(timestamp).formattedTimestamp
   assertThat(actual).isEqualTo(expected)
}
>> `actual on dev machine` = "2017-04-21 10:00" // UTC +3
>> `actual on CI`          = "2017-04-21 07:00" // UTC
```

#1: Formatting & Locales

Same for

1. `NumberFormat`
2. `Currency`
3. `Locale`
4. `TimeZone`
5. ...

Testability in practice.

Mocking

Interfaces still work everywhere

```
interface MyService {
   fun doSomething()
   class Impl(): MyService {
       override fun doSomething() { /* ... */ }
   }
}
```

Interfaces still work everywhere

```
interface MyService {
   fun doSomething()
   class Impl(): MyService {
       override fun doSomething() { /* ... */ }
   }
}

class TestService: MyService {
   override fun doSomething() { /* ... */ }
}
```

Interfaces still work everywhere

```
interface MyService {
   fun doSomething()
   class Impl(): MyService {
       override fun doSomething() { /* ... */ }
   }
}

class TestService: MyService {
   override fun doSomething() { /* ... */ }
}

val mockService = mock<MyService>()
```

Interfaces still work everywhere

```
interface MyService {
   fun doSomething()
   class Impl(): MyService {
       override fun doSomething() { /* ... */ }
   }
}

class TestService: MyService {
   override fun doSomething() { /* ... */ }
}

val mockService = mock<MyService>()
```

Extracting singletons

```
object Implicit {
   fun getCurrentState(): String = "State"
}

class SomeModule {
   init {
       val state = Implicit.getCurrentState()
   }
}
```

Extracting singletons

```
interface StateProvider {
   fun getCurrentState(): String
}

object Implicit {
   fun getCurrentState(): String = "State"
}

class SomeModule {
   init {
       val state = Implicit.getCurrentState()
   }
}
```

Extracting singletons

```
interface StateProvider {
   fun getCurrentState(): String
}

object Implicit: StateProvider {
   override fun getCurrentState(): String = "State"
}

class SomeModule {
   init {
       val state = Implicit.getCurrentState()
   }
}
```

Extracting singletons

```
interface StateProvider {
   fun getCurrentState(): String
}

object Implicit: StateProvider {
   override fun getCurrentState(): String = "State"
}

class SomeModule(stateProvider: StateProvider) {
   init {
       val state = stateProvider.getCurrentState()
   }
}
```

Extracting singletons

```kotlin
interface StateProvider {
   fun getCurrentState(): String
}

object Implicit: StateProvider {
   override fun getCurrentState(): String = "State"
}

class SomeModule(stateProvider: StateProvider) {
   init {
       val state = stateProvider.getCurrentState()
   }
}
```

Other alternatives

Mockito 2 for mocking **final**

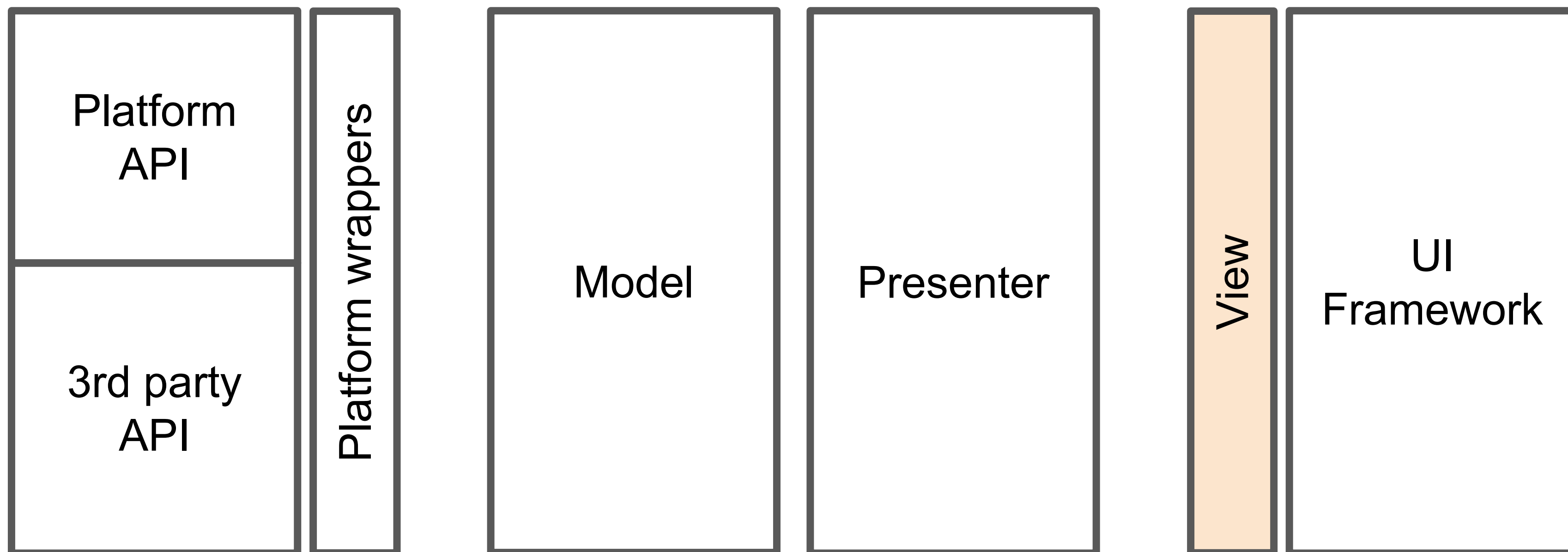
PowerMock for mocking **final**, **static**, **singletons**, …

- Indicate design issues; can be avoided
- Might blow up at some point

Testability in practice.

Abstracting away the platform

MVP. View layer



Presenter isolation

View layer approaches

- Activity implements View
- View as a separate class

Activity implements View

```
class SplashPresenter(view: SplashView) {
   init {
       view.showLoading()
   }
}
```

Activity implements View

```
interface SplashView {
   fun showLoading()
}
```

Activity implements View

```
interface SplashView {
   fun showLoading()
}

class SplashActivity: Activity {
   override fun onCreate() {

   }



}
```

Activity implements View

```
interface SplashView {
    fun showLoading()
}

class SplashActivity: Activity, SplashView {
    override fun onCreate() {

    }
    override fun showLoading() {
        findViewById(R.id.progress).show()
    }
}
```

Activity implements View

```
interface SplashView {
   fun showLoading()
}

class SplashActivity: Activity, SplashView {
   override fun onCreate() {
       SplashPresenter(view = this)
   }
   override fun showLoading() {
       findViewById(R.id.progress).show()
   }
}
```

Activity implements View

```kotlin
interface SplashView {
   fun showLoading()
}

class SplashActivity: Activity, SplashView {
   override fun onCreate() {
      SplashPresenter(view = this)
   }
   override fun showLoading() {
      findViewById(R.id.progress).show()
   }
}
```

View as a separate class

```
// Same as before
class SplashPresenter(view: SplashView) {
   init {
       view.showLoading()
   }
}
```

View as a separate class

```
interface SplashView {
   fun showLoading()







}
```

View as a separate class

```kotlin
interface SplashView {
    fun showLoading()


    class Impl : SplashView {
        override fun showLoading() {

        }
    }
}
```

View as a separate class

```
interface SplashView {
   fun showLoading()


   class Impl(private val viewRoot: View) : SplashView {
       override fun showLoading() {
           viewRoot.findViewById(R.id.progress).show()
       }
   }
}
```

View as a separate class

```
interface SplashView {
    fun showLoading()

    // Platform lives inside of Impl
    class Impl(private val viewRoot: View) : SplashView {
        override fun showLoading() {
            viewRoot.findViewById(R.id.progress).show()
        }
    }
}
```

View as a separate class

```
class SplashActivity: Activity {

    override fun onCreate() {








    }
}
```

View as a separate class

```
class SplashActivity: Activity {

    override fun onCreate() {
        // Platform View class
        val rootView: View = ...




    }
}
```

View as a separate class

```
class SplashActivity: Activity {

    override fun onCreate() {
        // Platform View class
        val rootView: View = ...

        SplashPresenter(
                view = SplashView.Impl(rootView)
        )
    }
}
```

View as a separate class

```
class SplashActivity: Activity {

    override fun onCreate() {
        // Platform View class
        val rootView: View = ...

        SplashPresenter(
                view = SplashView.Impl(rootView)
        )
    }
}
```

Presenter is isolated from the platform

```
@Test
fun testLoadingIsShown() {
    val mockedView = mock<SplashView>()




}
```

Presenter is isolated from the platform

```
@Test
fun testLoadingIsShown() {
    val mockedView = mock<SplashView>()

    SplashPresenter(mockedView)


}
```

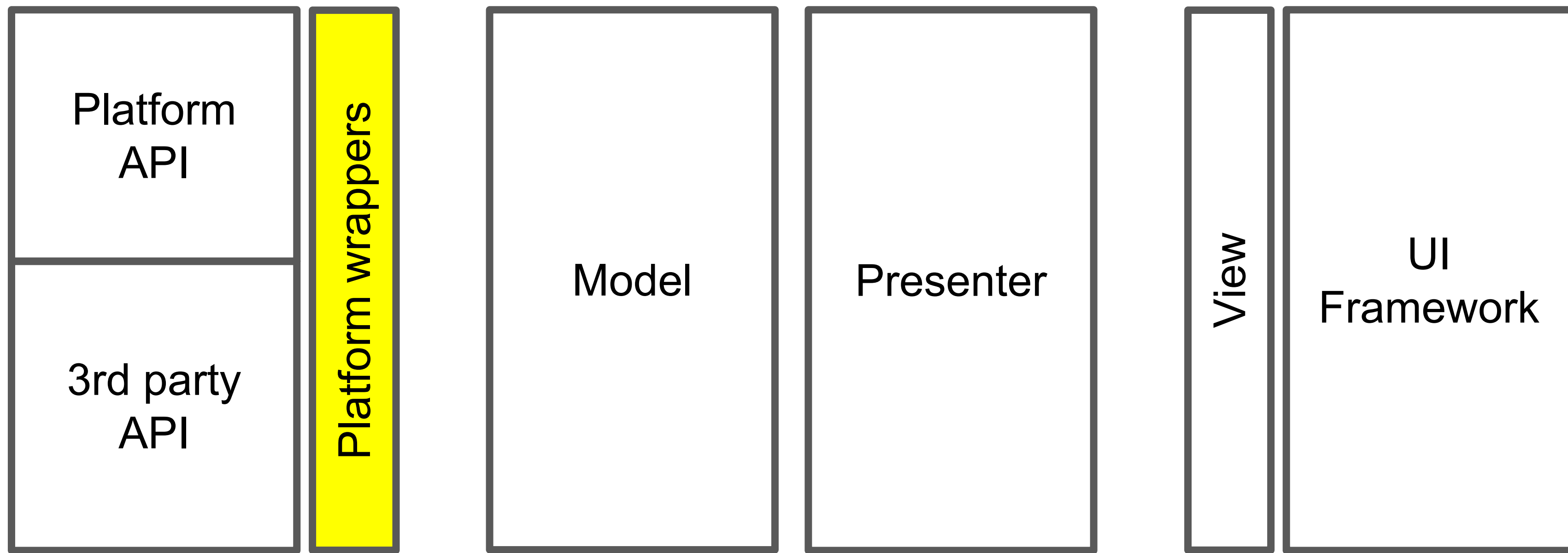
Presenter is isolated from the platform

```
@Test
fun testLoadingIsShown() {
    val mockedView = mock<SplashView>()

    SplashPresenter(mockedView)

    verify(mockedView).showLoading()
}
```

Presenter is isolated from the platform

```
@Test
fun testLoadingIsShown() {
    val mockedView = mock<SplashView>()

    SplashPresenter(mockedView)

    verify(mockedView).showLoading()
}
```

MVP. Platform wrappers



Model isolation

You can not modify

- Platform APIs
- 3rd party APIs

They can exist in different forms

- Static methods
- Singletons
- Final classes
- Non-final classes

They can exist in different forms

- Static methods
- Singletons
- Final classes
- Non-final classes

They can exist in different forms

- Static methods
- Singletons
- Final classes
- Non-final classes

They can exist in different forms

- Static methods
- Singletons
- Final classes
- Non-final classes

They can exist in different forms

- Static methods
- Singletons
- Final classes
- Non-final classes

Creating a wrapper

```
class Module {
    init {
        ThirdParty.doSomething()
    }
}
```

Creating a wrapper

```
interface Wrapper {
    fun doSomething()

    class Impl: Wrapper {
        override fun doSomething() {
            ThirdParty.doSomething()
        }
    }
}
```

Applying the wrapper

```
class Module {
    init {
        ThirdParty.doSomething()
    }
}
```

Applying the wrapper

```
class Module(wrapper: Wrapper) {
    init {
        wrapper.doSomething()
    }
}
```

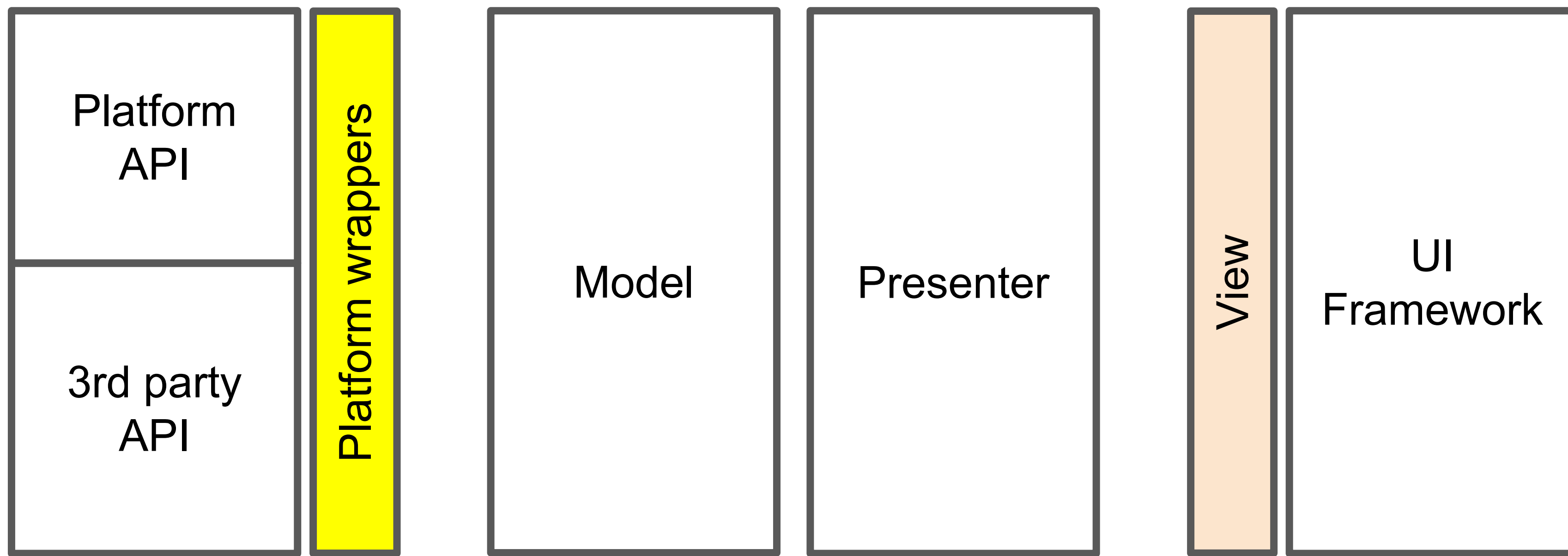
Applying the wrapper

```
class Module(wrapper: Wrapper) {
    init {
        wrapper.doSomething()
    }
}
```

Wrapper extra benefits

- Fixes poor Platform API design
- Single Responsibility instead of God objects
- Easy to apply 3rd party API changes

MVP. Platform wrappers vs View



Accessing resources

Accessing string ids

```
class SplashPresenter(view: SplashView, resources: Resources) {
   init {
       view.setTitle(resources.getString(R.string.welcome))
       view.showLoading()
   }
}
```

Accessing string ids

```
class SplashPresenter(view: SplashView, resources: Resources) {
    init {
        view.setTitle(resources.getString(R.string.welcome))
        view.showLoading()
    }
}

interface Resources {
    fun getString(id: Int): String
}
```

Accessing string ids

```
class SplashPresenter(view: SplashView, resources: Resources) {
    init {
        view.setTitle(resources.getString(R.string.welcome))
        view.showLoading()
    }
}
```

Accessing string ids

```
public final class R {
   public static final class string {
       public static final int welcome=0x7f050000;
   }
}
```

Implementation details

Implementation details

```
class SomeModule(input: String) {
   val state = calculateInitialState(input)
   // Pure
   private fun calculateInitialState(input: String): String =
           "Some complex computation for $input"
}
```

Implementation details

```
class SomeModule(input: String) {
    val state = calculateInitialState(input)
    // Pure
    private fun calculateInitialState(input: String): String =
            "Some complex computation for $input"
}
```

Implementation details

```
class SomeModule(input: String) {
    val state = calculateInitialState(input)
    // Pure
    private fun calculateInitialState(input: String): String =
            "Some complex computation for $input"
}

class AnotherModule(input: String) {
    val state = calculateInitialState(input)
    // Pure
    private fun calculateInitialState(input: String): String =
            "Some complex computation for $input"
}
```

Implementation details

```
class SomeModule(input: String) {
    val state = calculateInitialState(input)
    // Pure
    private fun calculateInitialState(input: String): String =
            "Some complex computation for $input"
}

object StateCalculator {
    fun calculateInitialState(input: String): String =
            "Some complex computation for $input"
}
```

Implementation details

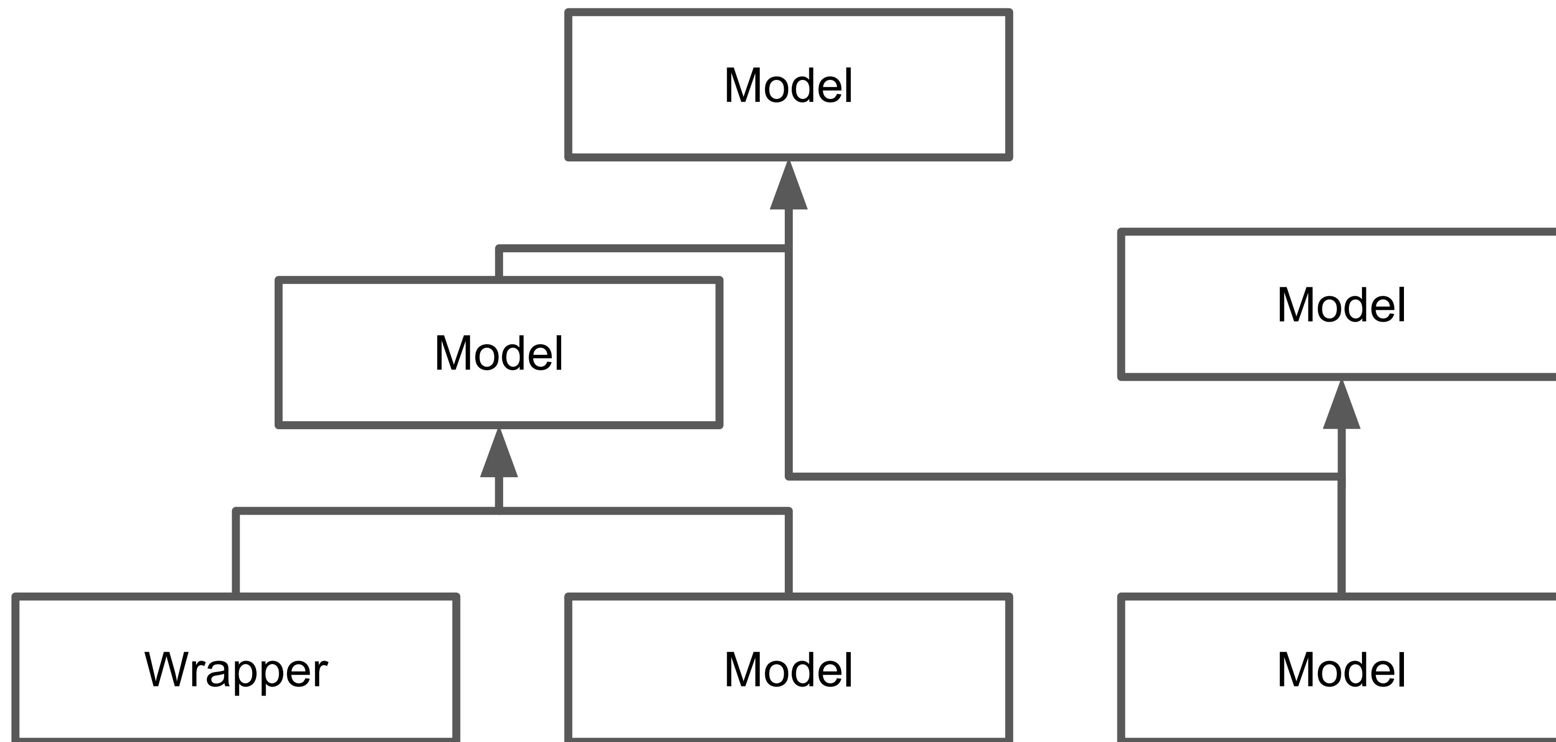
```
class SomeModule(input: String) {
   val state = StateCalculator.calculateInitialState(input)
}

class AnotherModule(input: String) {
   val state = StateCalculator.calculateInitialState(input)
}

object StateCalculator {
   fun calculateInitialState(input: String): String =
           "Some complex computation for $input"
}
```

Implementation details

```
class SomeModule(input: String, stateCalculator: StateCalculator){
   val state = stateCalculator.calculateInitialState(input)
}

interface StateCalculator {
   fun calculateInitialState(input: String): String
   class Impl: StateCalculator {
       override fun calculateInitialState(input: String): String =
               "Some complex computation for $input"
   }
}
```

How to start?

Start with Models



Start with Models

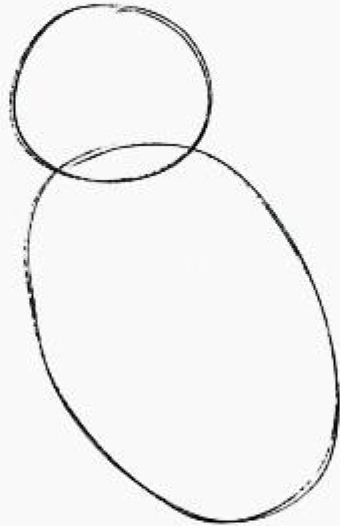


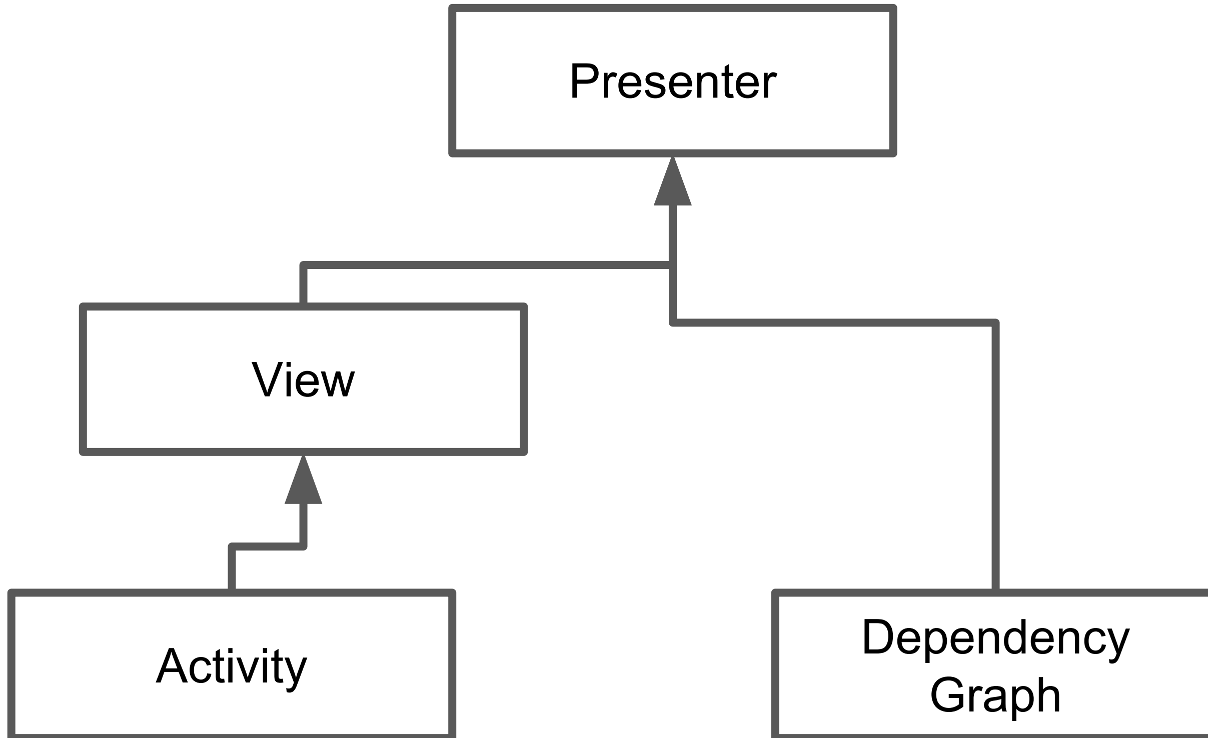
Fig 1. Draw two circles

Fig 2. Draw the rest of the damn Owl

As a result

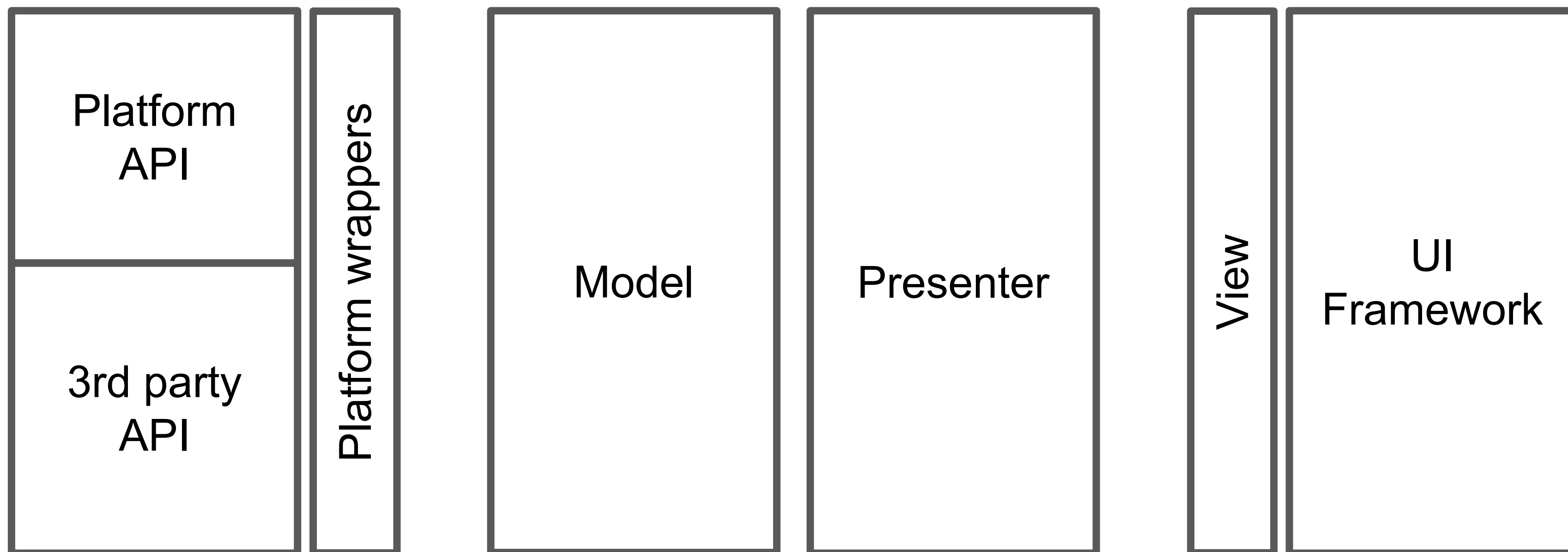
- **Implicit** dependencies (singletons) become **explicit** (passed through DI)
- Initialization flow gets explicit
- Models become easy to test

And then

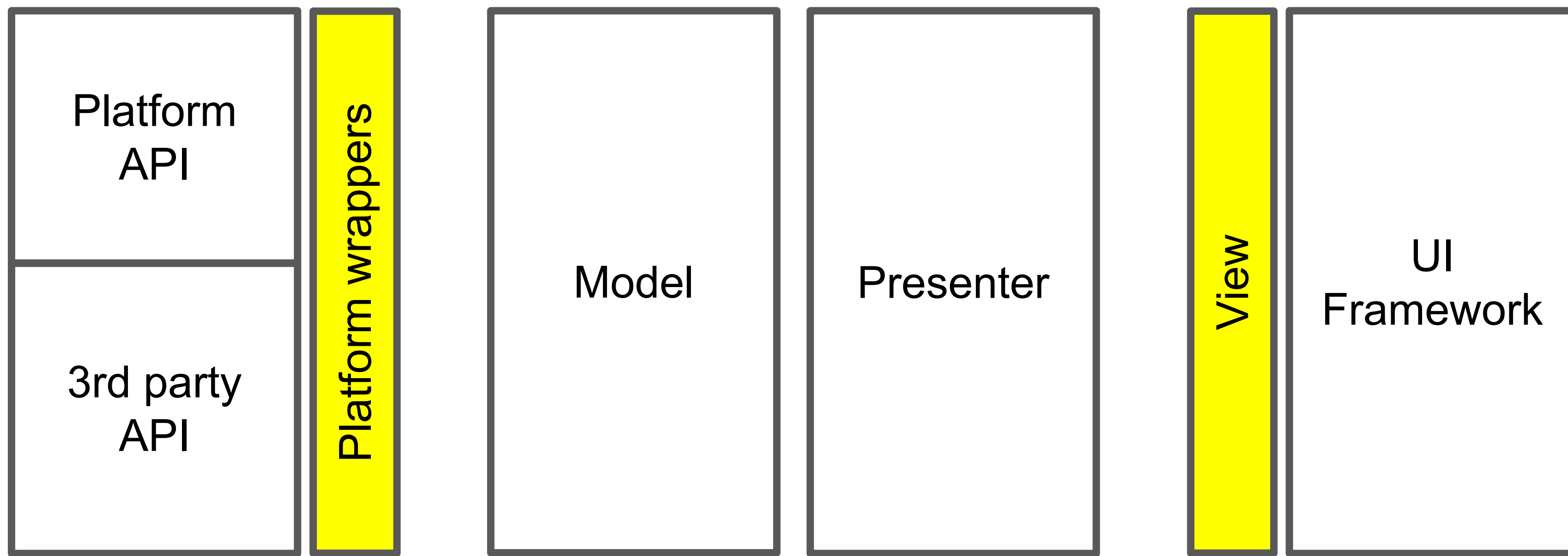


What’s next?

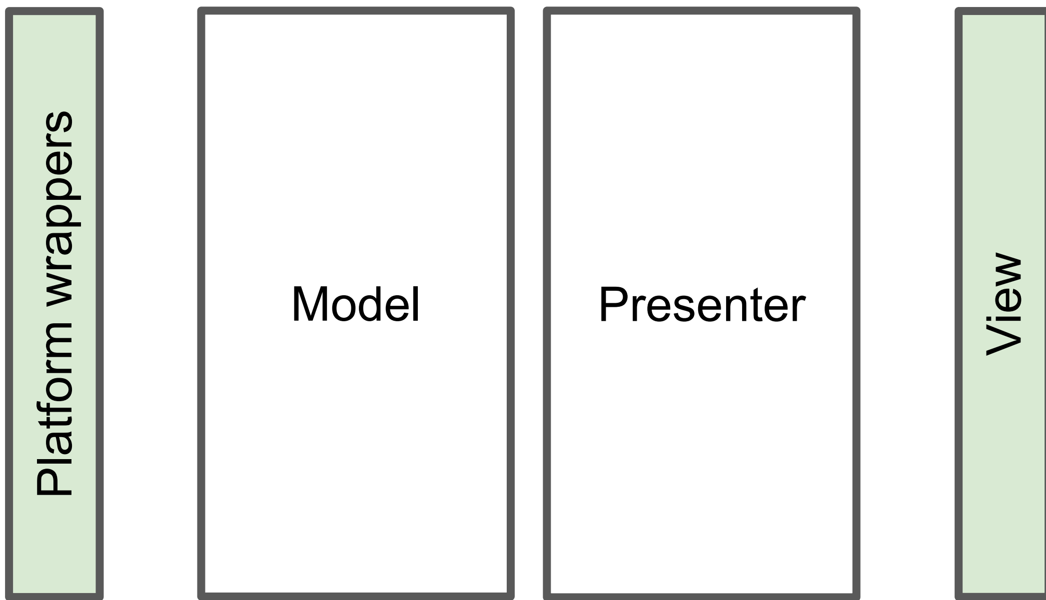
MVP. Integration tests



MVP. Integration tests



MVP. Integration tests



To summarize

3 rules

- Pass arguments and return results **explicitly**
- Pass dependencies **explicitly**
- Make sure **you can mock** dependencies

Learning the art of programming, like most other disciplines, consists of first learning the rules and then learning when to break them.

Joshua Bloch, Effective Java

Questions?

Twitter, GitHub, Facebook
AntonRutkevich

e-mail
anton@rutkevich.com