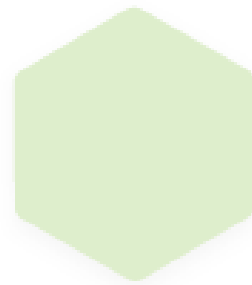


# Reuse features in Android applications

An introduction to component modularity

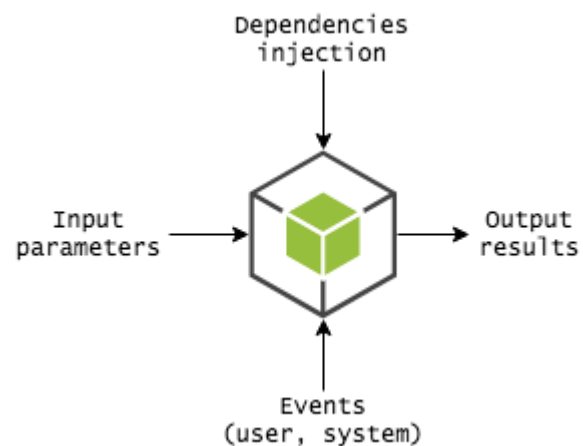
Romain Rochegude



# Introduction

# What to reuse?

- divide a program into separated sub-programs (features)
- independent, reusable and isolated
- unit of code to compile (i.e., Android Studio *module*)
- almost like [React Component](#)



## Benefits

- ease incremental builds and deliveries
- module is unit-testable
- modules can be added, modified or removed without any impact on one another
- modules can be reused

## Background: Android key concepts

### Activity (since API level 1)

one of the fundamental building blocks

<https://developer.android.com/guide/components/activities/index.html>

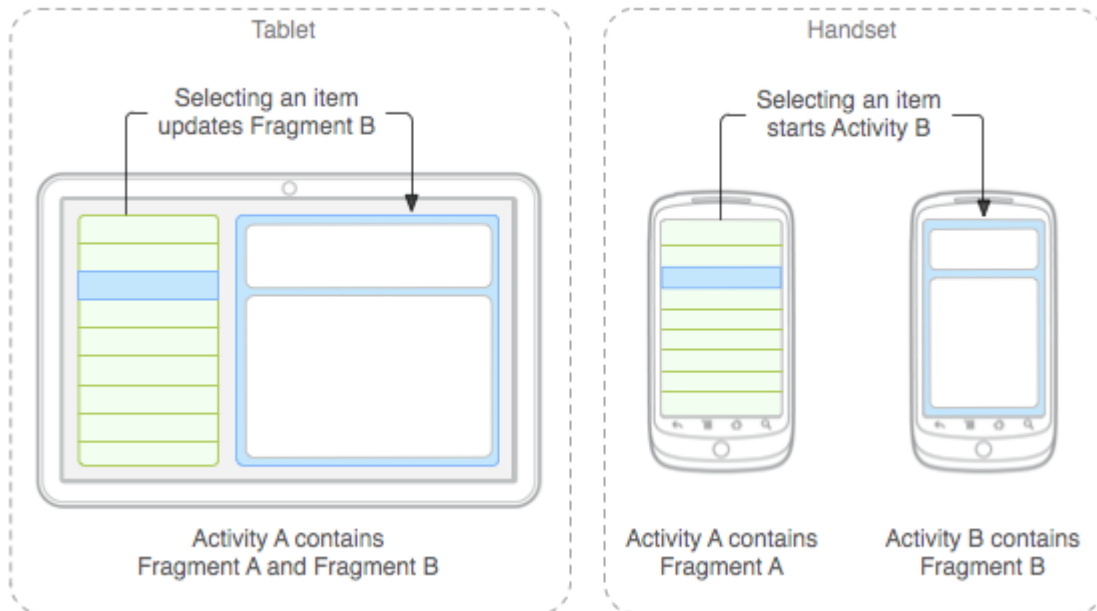
- behind every screen stands a single `Activity`

# Background: Android key concepts

## Fragment (since API level 11)

portion of user interface in an `Activity`

<https://android-developers.googleblog.com/2011/02/android-30-fragments-api.html>



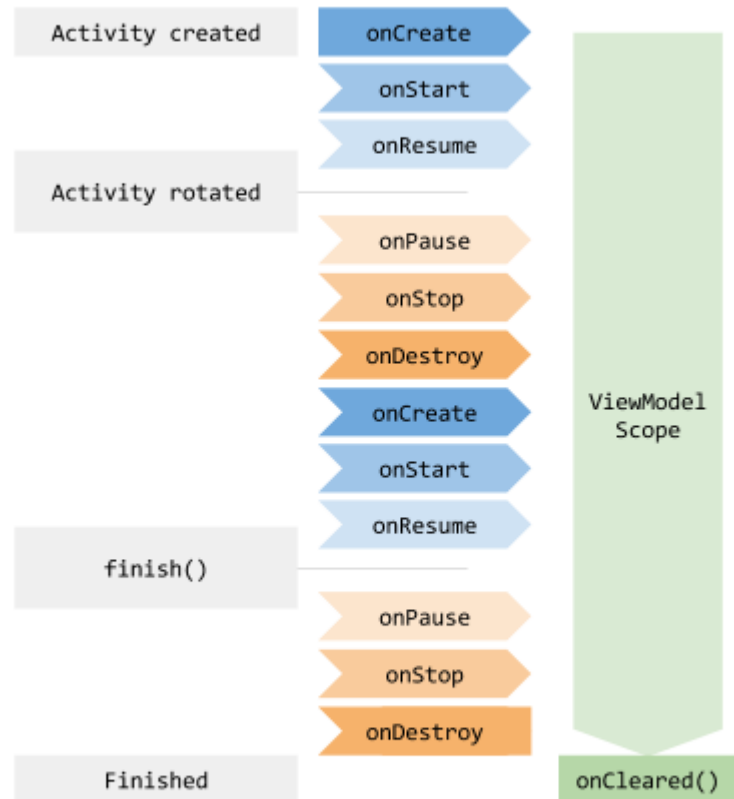
## Background: Android key concepts

### ViewModel (since ACC)

The `ViewModel` class

- is designed to store and manage UI-related data in a lifecycle conscious way.
- allows data to survive configuration changes such as screen rotations.

<https://developer.android.com/topic/libraries/architecture/viewmodel.html>



```
fun onCreate(savedInstanceState: Bundle) {  
    // Create a ViewModel the first time the system calls an activity's onCreate  
    // Re-created activities receive the same MyViewModel instance  
    val viewModel = ViewModelProviders.of(this).get(MyViewModel::class.java)  
}
```



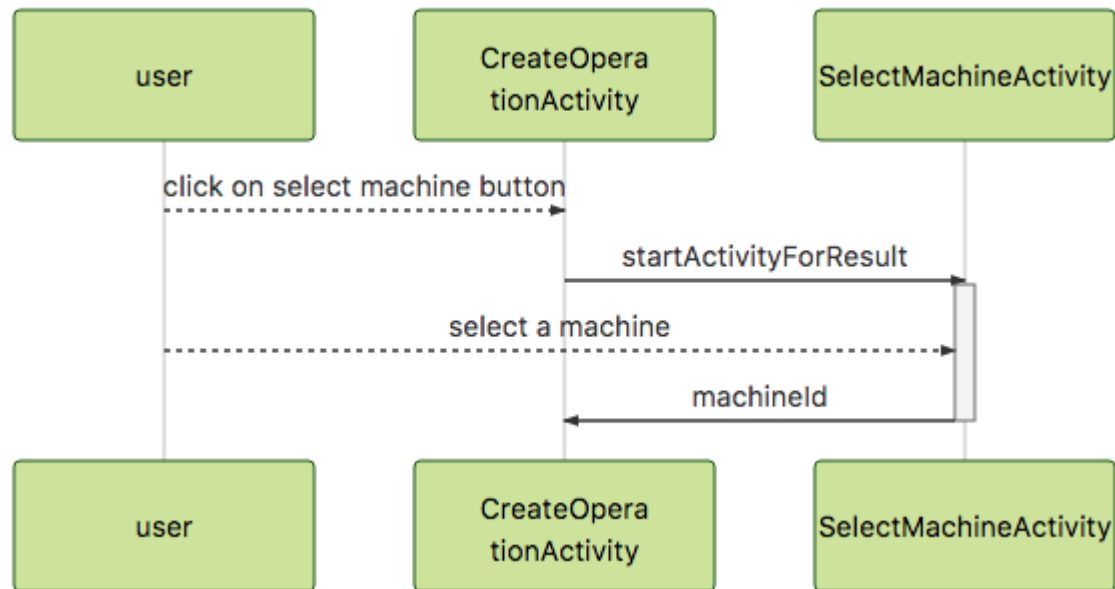
# 1. Native solutions

# Activity + result code

## Getting a Result from an Activity

### Example

- First activity: fill a form to create a new operation
- Second activity: select a capable machine



With the help of [Anko](#)

### CreateOperationActivity.kt

```
findViewById<Button>(R.id.button_select_machine).setOnClickListener {  
    startActivityForResult<SelectMachineActivity>(  
        SELECT_MACHINE,  
        SelectMachineActivity.CAPABILITY to typeOfOperation  
    )  
}
```

## SelectMachineActivity.kt

```
findViewById<Button>(R.id.select_machine_a).setOnClickListener {  
    val intent = Intent()  
    intent.putExtra(MACHINE_ID, machineA.id)  
    setResult(  
        Activity.RESULT_OK,  
        intent  
    )  
    finish()  
}  
  
companion object Params {  
    val CAPABILITY = "SelectMachineActivity:capability"  
    val MACHINE_ID = "SelectMachineActivity:machineId"  
}
```

## CreateOperationActivity.kt

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {  
    if (requestCode == SELECT_MACHINE) {  
        if (resultCode == Activity.RESULT_OK) {  
            selectedMachineId = data?.getLongExtra(SelectMachineActivity.MACHINE_ID, -1)  
        }  
    } else {  
        super.onActivityResult(requestCode, resultCode, data)  
    }  
}
```

## Activity + result code: assessments

- Pros:
  - stable
  - many libraries written this way
- Cons:
  - not composable (1 activity per screen)
  - break the code flow (but [rx to the rescue](#))

## Fragment + callbacks

### Communicating with Other Fragments

The embedded `Fragment` defines a callback interface

```
class SelectMachineFragment : Fragment() {  
    interface OnFragmentInteractionListener {  
        fun onSelectedMachine(selectedMachineId: Long)  
    }  
}
```

The **Activity** must implement this callback

```
class CreateOperationActivity :  
    AppCompatActivity(),  
    SelectMachineFragment.OnFragmentInteractionListener {  
  
    override fun onSelectedMachine(selectedMachineId: Long) {  
        this.selectedMachineId = selectedMachineId  
    }  
}
```



The `Fragment` handles a reference to its callback

```
class SelectMachineFragment : Fragment() {  
    private var listener: OnFragmentInteractionListener? = null  
  
    override fun onAttach(context: Context) {  
        super.onAttach(context)  
        if (context is OnFragmentInteractionListener) {  
            listener = context  
        } else {  
            throw RuntimeException(context.toString() +  
                " must implement OnFragmentInteractionListener")  
        }  
    }  
  
    override fun onDetach() {  
        super.onDetach()  
        listener = null  
    }  
}
```

The **Fragment** uses the callback interface to deliver the event to the parent activity

```
override fun onCreateView(inflater: LayoutInflater,
                           container: ViewGroup?,
                           savedInstanceState: Bundle?): View? {
    val view = inflater.inflate(
        R.layout.fragment_select_machine,
        container,
        false
    )
    view.findViewById<Button>(R.id.select_machine_a).setOnClickListener {
        listener?.onSelectedMachine(1L)
    }
    return view
}
```

The **Activity** can deliver a message to another **Fragment**

```
class AnotherFragment : Fragment() {  
    fun updateUi(selectedMachineId: Long) {  
        TODO("update UI with selectedMachineId")  
    }  
}
```

```
class CreateOperationActivity :  
    AppCompatActivity(),  
    SelectMachineFragment.OnFragmentInteractionListener {  
  
    override fun onSelectedMachine(selectedMachineId: Long) {  
  
        val anotherFragment = supportFragmentManager.findFragmentById(  
            R.id.another_fragment_container_id  
        ) as AnotherFragment  
  
        if (anotherFragment == null) {  
            anotherFragment.updateUi(selectedMachineId)  
        } else {  
            TODO("create and display AnotherFragment with selectedMachineId")  
        }  
    }  
}
```

## Fragment + callbacks: assessments

- Pros:
  - composable
  - now compatible with the ACC [ViewModel](#)
- Cons:
  - boilerplate code
  - no compile-time checking

## Native solutions: assessments

- Pros:
  - native solutions are possible
  - tried and tested
- Cons:
  - troublesome to setup
  - difficult to compose
  - no navigation concerns

## 2. Use of a finite state machine (FSM)

## Background: FSM key concepts

- sequential logic circuits
- finite number of states
- one state at a time (the current state)
- change from one state to another by triggering an event (a transition)



## Event-driven programming

- the module fires an event,
- the hosting application receives this event and acts accordingly
- the flow is determined by events
  - user actions, network requests, sensors, timer, other threads, etc.

## Why EasyFlow

- simple to set up
- possible definition of a global context
- states definition through the `StateEnum` interface
- events definition through the `EventEnum` interface
- fluent API
- callbacks to perform specific jobs when entering or leaving a state

# Setup with Android components

- `Fragment` to define a state of the application (i.e., a use case) and output event(s)
- `Activity` to manage states and how to navigate (i.e., the flow of events to change application state)

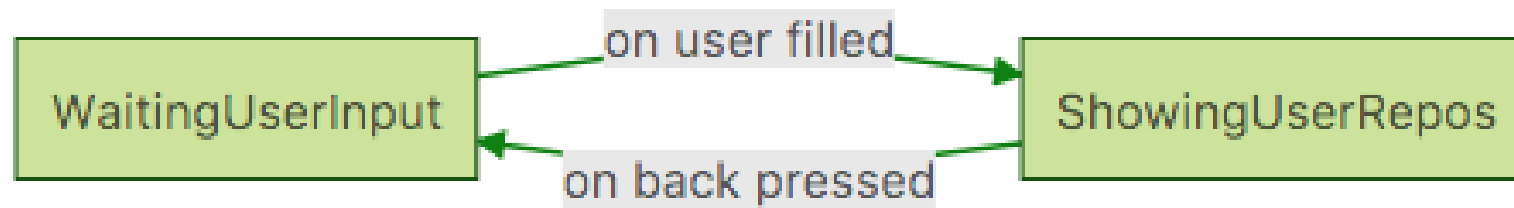
## Constraint 1: orientation changes

- Use of the ACC `ViewModel`
  - Define and share a specific `ViewModel` between `Fragment`s

## Constraint 2: dependency injection

- The hard case of `Dagger 2`
  - Pros: code generation, hosted by Google
  - Cons: many concepts to know and huge amount of code to write
- A nice way with `Koin`

## Example



## Common things

- The global context of the FSM

```
class FsmContext : StatefulContext() {  
    val args = Bundle()  
}
```

## Common things

- The shared `ViewModel`

```
class FsmViewModel : ViewModel() {  
    val fsmModel: MutableLiveData<FsmModel> = MutableLiveData()  
  
    init {  
        fsmModel.value = FsmModel()  
    }  
  
    val flowContext: FsmContext  
        get() = fsmModel.value?.flowContext!!  
  
    fun trigger(event: EventEnum) {  
        flowContext.safeTrigger(event)  
    }  
}
```

## Common things

- The FSM module

```
val fsmModule = applicationContext {  
    viewModel {  
        FsmViewModel()  
    }  
}  
  
object BackPressed : FsmEvent
```

Focus on "user input"

## The *Model*

```
class UserInputModel {  
    val user: ObservableField<String> = ObservableField()  
}
```

## The *ViewModel*

```
class UserInputViewModel: ViewModel() {  
    val model: UserInputModel = UserInputModel()  
    val onSelectEvent = SingleLiveEvent<String>()  
  
    fun onSelectButtonClicked() {  
        onSelectEvent.postValue(model.user.get())  
    }  
}
```



```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable
            name="model"
            type="fr.guddy.kandroidmodular.userinput.mvvm.UserInputModel" />
        <variable
            name="viewModel"
            type="fr.guddy.kandroidmodular.userinput.mvvm.UserInputViewModel" />
    </data>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <EditText
            android:id="@+id/editTextUser"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@={model.user}" />

        <android.support.v7.widget.AppCompatButton
            android:id="@+id/buttonSelect"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:onClick="@{() -> viewModel.onSelectButtonClicked()}"
            android:text="@string/user_input_button" />
    </LinearLayout>
</layout>
```

```

class UserInputFragment : Fragment() {
    /*...*/

    override fun onCreateView(/*...*/): View? {
        binding = DataBindingUtil.inflate(/*...*/)
        return binding.root
    }

    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        viewModel = getViewModel()
        fsmViewModel = getViewModelFromActivity()
        binding.viewModel = viewModel
        binding.model = viewModel.model
        viewModel.onSelectEvent.observe(this) { user -> onSelect(user) }
    }

    private fun onSelect(user: String) {
        if (TextUtils.isEmpty(user)) {
            binding.editTextUser.error = getString(R.string.empty_user)
        } else {
            fsmViewModel.flowContext.userInputResult = UserInputResult(user)
            fsmViewModel.trigger(UserFilled)
        }
    }
}

```

## Koin setup for DI

- Define the module:

```
val userInputModule = applicationContext {  
    viewModel { UserInputViewModel() }  
}
```

- Start DI:

```
val allModules = listOf(  
    /*...*/  
    userInputModule  
)  
  
class MyApplication : Application() {  
    override fun onCreate() {  
        super.onCreate()  
  
        startKoin(this, allModules)  
    }  
}
```

## FSM configuration

- The result data

```
@PaperParcel
data class UserInputResult(val user: String) : PaperParcelable {
    companion object {
        @JvmField
        val CREATOR = PaperParcelUserInputResult.CREATOR
    }
}
```

With the help of [paperparcel](#)

- The module setup

```
object WaitingUserInput : FsmState

object UserFilled : FsmEvent

var FsmContext.userInputResult: UserInputResult
    get() = args.getParcelable("UserInputResult")
    set(value) {
        args.putParcelable("UserInputResult", value)
    }

fun FsmContext.clearUserInputResult() {
    args.remove(_resultKey)
}
```

## Integration in the hosting application

```
class MainActivity : AppCompatActivity() {  
  
    private lateinit var fsmViewModel: FsmViewModel  
    private lateinit var flow: EasyFlow<FsmContext>  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        fsmViewModel = getViewModel()  
        buildFsm()  
    }  
}
```

```

// MainActivity.kt
private fun buildFsm() {
    flow = from<FsmContext>(WaitingUserInput).transit(
        on(UserFilled).to(ShowingUserRepos).transit(
            on(BackPressed).to(WaitingUserInput)
        )
    )
    // callbacks
    flow.whenEnter(WaitingUserInput) { showUserInputFragment() }
    flow.whenEnter(ShowingUserRepos) { context ->
        showUserReposFragment(context.userInputResult.user)
    }
    flow.whenLeave(ShowingUserRepos) { context ->
        context.clearUserInputResult()
    }
    // start with current state
    flow.start(fsmViewModel.flowContext)
}

private fun showUserInputFragment() { /*...*/ }

private fun showUserReposFragment(user: String) { /*...*/ }

```

```
// MainActivity.kt
```

```
override fun onBackPressed() {  
    if (supportFragmentManager.backStackEntryCount > 0) {  
        fsmViewModel.trigger(BackPressed)  
        supportFragmentManager.popBackStack()  
    } else {  
        super.onBackPressed()  
    }  
}
```



# Conclusion

# Benefits

- relevant MVVM architecture
- power of the Kotlin language
- an elegant way to define the application flow
- no explicit coupling between screens
- increase testability
  - test at module level (easy to stub injected dependencies thanks to Koin)
  - test at application level
- adjustable to technical stack

## Main used Kotlin concepts

- Extensions (functions, properties)
- Object declarations
- Delegated Properties
- Data classes
- Default and named arguments

## To go further

- <https://roroche.github.io/AndroidModularSample/>
- <https://github.com/RoRoche/kAndroidModular>

# What's next?

## Practical

- Syntax enhancement thanks to Kotlin
- Group redundant concerns in Java/Android libraries
- Expose features through a repository

## Ideal

- Front-end with drag&drop feature to build application flow?
- Kotlin: build iOS application and share common modules?
- React-native: write and share common modules (mobile and desktop)?

# Thanks

- [Macroscope](#) for many relevant articles
  - [Applications as State Machines](#)
  - [Introducing SwiftyStateMachine](#)
- [Nicolas Chassagneux](#) for many enriching discussions