Sexy Architecting
VIPER: MVP on steroids

Dmytro Zaitsev
Mobile Team Lead
MVP/MVC/MVVM is NOT an Architecture!

It’s only responsible for the presentation layer delivery mechanism
Real-world Android app

- Hard to understand
- Hard to maintain
- Hard and often impossible to test
- The business logic is mixed in Activity/Fragment
- High coupled components
- MVC -> Massive View Controllers
Clean Architecture

- Independent of Frameworks
- Testable
- Independent of UI
- Independent of Database
- Independent of any external agency
The Web is an I/O Device!

-Robert Martin
What is VIPER?

- A way of architecting applications which takes heavy inspiration from the Clean Architecture
- Divides an app’s logical structure into distinct layers of responsibility
- Makes it easier to isolate dependencies
- Makes it easier to test the interactions at the boundaries between layers
- Eliminates Massive View Controllers
Main parts of VIPER

- View
- Interactor
- Presenter
- Entity
- Router
View

Displays what it is told to by the Presenter and relays user input back to the Presenter
View

- Is passive
- Never asks the Presenter for data
- Handles user interaction and input
- Simply delegates user’s actions to the Presenter
- Waits for the Presenter to give it content to display
- Awaits for a response telling it what should be displayed next
- Determines how the content is displayed
Example of View

internal interface CheeseViewCallbacks {
    fun onNewCheese(cheese: Collection<CheeseViewModel>)
    fun showError()
    fun hideProgress()
    fun showProgress()
}
Example of View

class CheeseView : ConstraintLayout, CheeseViewCallbacks {
    @Inject internal lateinit var presenter: CheesePresenter
    private lateinit var progressDialog : ProgressDialog
    private lateinit var adapter : CheeseAdapter

    /** ... */

    override fun onNewCheese(cheese: Collection<CheeseViewModel>) {
        adapter.setModels(cheese)
        adapter.notifyDataSetChanged()
    }

    override fun showError() {
        Toast.makeText(context, R.string.error, Toast.LENGTH_SHORT).show()
    }

    override fun hideProgress() = progressDialog.dismiss()

    override fun showProgress() = progressDialog.show()
}
Presenter

Contains view logic for preparing content for display (as received from the Interactor) and for reacting to user inputs (by requesting new data from the Interactor)
Presenter

- Works like a bridge between the main parts of a VIPER module
- Knows about the content it maintains and when it should be displayed
- Receives input events coming from the View
- Sends requests to an Interactor
- Receives the data structures coming from the Interactor
- Applies view logic over this data to prepare the content
- Tells the View what to display
- Knows when to navigate to another screen, and which screen to navigate to
class CheesePresenter(private val getCheeseInteractor: GetCheeseInteractor) {
    var view: CheeseViewCallbacks? = null
    var router: MainRouter? = null
    /** ... */
    fun fetchCheese(count: Int) {
        view?.showProgress()
        getCheeseInteractor.execute({ cheese -> // onNext
            view?.onNewCheese(cheese)
            view?.hideProgress()
        },
        { throwable -> // onError
            view?.showError()
            view?.hideProgress()
        },
        count)
    }
    fun onClicked(model: CheeseViewModel) = router?.navigateToDetails(model)
}
Interactor

Contains the business logic as specified by a use case
**Interactor**

- Encapsulates application specific business rules
- Represents use cases
- Regular Java object
- No Android framework dependency
Example of Interactor

class GetCheeseInteractor @Inject constructor(
    private val subscribeOn : Scheduler,
    private val observeOn : Scheduler,
    private val cheeseStorage: CheeseStorage) {
    private val subscriptions = CompositeSubscription()

    fun execute(subscriber: Subscriber<Collection<Cheese>>, count: Int) {
        subscriptions.add(cheeseStorage.getCheese(count)
            .subscribeOn(subscribeOn)
            .observeOn(observeOn)
            .subscribe(subscriber))
    }
}
Entity

Contains basic model objects used by the Interactor
Entity

- Model objects manipulated by an Interactor
- POJOs
- Encapsulates different types of data
data class Cheese(
    val id : Long,
    val name : String,
    val price : Long,
    val description : String,
    val type : String,
    val texture : String,
    val fatContent : String,
    val animalMilk : String,
    val regionOfOrigin: String
)
Router

Contains navigation logic for describing which screens are shown in which order
Router

- Responsible for the navigation logic between modules
- Responsible for passing data between screens
- Receives input commands from the Presenter
Example of Router

```kotlin
internal interface MainRouter {
    fun navigateToDetails(model: CheeseViewModel)
    fun navigateToPreferences()
    fun navigateToRegistration()
}
```
Example of Router

class MainActivity : AppCompatActivity(), MainRouter {
    override fun navigateToDetails(model: CheeseViewModel) {
        startActivity(Intent(this, DetailsActivity::class.java).apply {
            with(this) {
                putExtra(DetailsActivity.NAME, model.name)
                putExtra(DetailsActivity.CHECKED, model.is Checked)
            }
        })
    }

    override fun navigateToPreferences() {
        startActivity(Intent(this, SettingsActivity::class.java))
    }

    override fun navigateToRegistration() {
        supportFragmentManager.beginTransaction().apply {
            replace(R.id.content, LoginFragment())
            commit()
        }
    }
}
Why should you use VIPER?

- It’s easier to track issues via crash reports
- The source code will be cleaner, more compact and reusable
- Adding new features is easier
- There are less conflicts with the rest of the development team
- It’s easier to write automated tests
When should you NOT use VIPER?

- It’s an overkill for small projects
- Causes an overhead when starting new projects
- MVP/MVC/MVVM-VIPER mix can cause headaches
- Lots of code all over the project
Testing

- Presentation layer
  - Espresso, Roboelectric

- Domain layer
  - JUnit, TestNG, Mockito, PowerMock

- Data layer
  - Roboelectric, JUnit, TestNG, Mockito, PowerMock
References

- https://github.com/RxViper/RxViper
- http://mutualmobile.github.io/blog/2013/12/04/viper-introduction/
- https://www.objc.io/issues/13-architecture/viper/
- https://www.ckl.io/blog/ios-project-architecture-using-viper/
Thank you!

Questions?

Dmytro Zaitsev
@DmitriyZaitsev