Mejoras en la arquitectura de Android – v1.0

--/05/2023 by Rui Vieira

Temas:

1. Motivación
2. Pautas recomendadas
3. Diseño y Requisitos de proyecto
4. Análisis del estado actual
    1. Mejoras a aplicar
        1. *Data injection objects straight in the UI*
        2. Clases no recomendadas
        3. Dominio
            1. Acoplamiento fuerte
            2. *Unnecessary usage of UseCase abstraction*
            3. Como debería ser
        4. Faltan ejemplos de pruebas
    2. Jetpack Compose
    3. Librería *mlb-itxcomponentsandroid*
5. Conclusión

Motivación:

- *Quality! (robust and solid app)*
  - *Write testable code.*

- *With quality, I can do:*
  - *Faster development*
  - *Faster testing*
  - *Faster shipping*

- *Target model:*
  - *High maintainability*
  - *Low amount of time*
  - *Low risk*

Modelo apropiado?

Modelo apropiado:

- *Easy to read, maintain, scale and test.*
  - *Apply best practices! “However, you should treat them as guidelines and adapt them to your requirements as needed.”*

https://developer.android.com/topic/architecture

Modelo apropiado:

- *Easy to read, maintain, scale and test.*
  - *Apply best practices! "However, you should treat them as guidelines and adapt them to your requirements as needed."*
    - *Common architectural principles ?*

https://developer.android.com/topic/architecture

Modelo apropiado:

- *Easy to read, maintain, scale and test.*
  - *Apply best practices! "However, you should treat them as guidelines and adapt them to your requirements as needed."*

    - *Common architectural principles:*
      - *Separation of concerns*
      - *Drive UI from data models*
      - *Single source of truth*
      - *Unidirectional Data Flow*

https://developer.android.com/topic/architecture

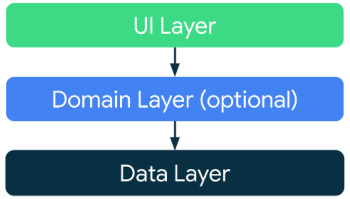
Modelo apropiado:

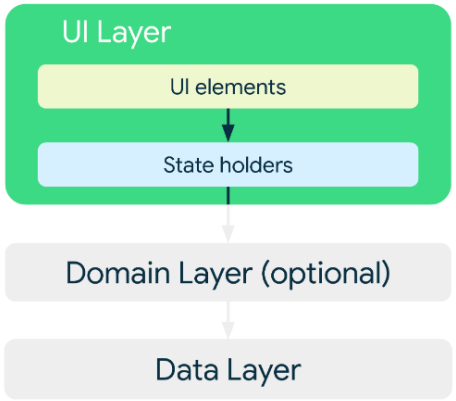
- *Easy to read, maintain, scale and test.*
  - *Apply best practices! "However, you should treat them as guidelines and adapt them to your requirements as needed."*
    - *Recommended app architecture ?*

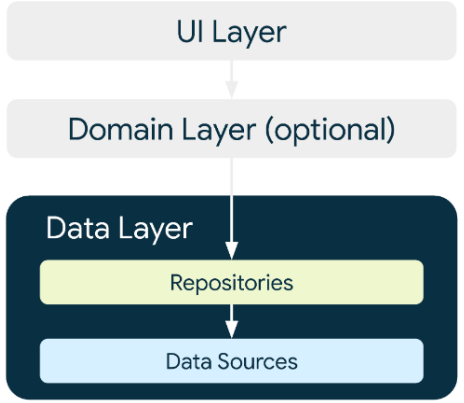
https://developer.android.com/topic/architecture

Modelo apropiado:

- *Easy to read, maintain, scale and test.*
  - *Apply best practices! "However, you should treat them as guidelines and adapt them to your requirements as needed."*
    - *Recommended app architecture:*

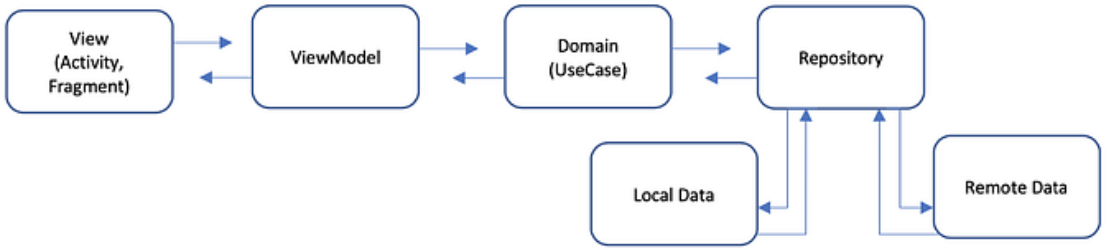






https://developer.android.com/topic/architecture

Diseño de proyecto:



- Clean architecture
- Abstraction layers

Análisis del estado actual?

Mejoras a aplicar:

1) Data injection objects straight in the UI

- *Lots of boilerplate non UI code located in Activities and Fragments*
- *Fails on the Separation of Concerns principle.*

```
@Suppress( ...names: "LargeClass")
class HomeFragment : BaseFragment<FragmentHomeBinding, HomeViewModel>() {

    override val mTag: String = "HomeFragment"
    override fun getViewBinding(): FragmentHomeBinding = FragmentHomeBinding.inflate(layoutInflater)
    override val mViewModel: HomeViewModel by viewModel()
    private val mainViewModel: MainViewModel by activityViewModels()
    private val mNavigator: NavigationManager by inject()
    private val homeSharedData: HomeSharedData?
        get() = mViewModel.homeSharedData

    override fun setupViews() {
        mViewModel.checkActiveWaves()
        mViewModel.disableCameraBarcodeModeConnection()
        setupListeners()
        setupBtnWave()
    }

    override fun setupViewModel() {
        mViewModel.getBottomSheetParams()
        mainViewModel.getStoreInfo()
    }
```

```
    private fun getSelectedSintMode() =
        SintMode.from(homeSharedData?.homePickingInfo?.sintModeValues?.selectedOption)

    private fun getUnitsToPick(): ItemsCount =
        homeSharedData?.homePickingInfo?.let { it: HomePickingInfo
            when(getSelectedSintMode()) {
                SintMode.SINT -> it.sintUnitsCount ^let
                SintMode.SINT_PLUS -> it.sintPlusUnitsCount ^let
                SintMode.SINT_PLUS_STORE -> it.sintPlusStoreUnitsCount ^let
            }
        } ?: ItemsCount( total: 0,  filtered: 0,  state: false)

    private fun getIncompleteOrdersUnits() =
        homeSharedData?.homePickingInfo?.let { it: HomePickingInfo
            val selectedSintMode = getSelectedSintMode()
            if (selectedSintMode != SintMode.SINT_PLUS) it.incompleteOrdersCount?.items ?: 0 ^let
            else 0 ^let
        } ?: 0
```


https://axinic.central.inditex.grp/bitbucket/scm/psintand/main.git

Mejoras a aplicar:

2) Non recommended classes

- *Use LiveData only if Java code is still present in the project.*
- *Use StateFlow instead.*
- *Check also the diferences:*

  StateFlow – SharedFlow – Channel

```
@HiltViewModel
class FirstViewModel @Inject constructor (
    private val getFirstDataExampleUseCase: GetFirstDataExampleUseCase
): ViewModel(), LifecycleObserver {

    private val error by lazy { MutableLiveData<String>() }
    val showErrorEvent: LiveData<Event<String>> = Transformations.map(error) { Event(it) }

    /** loadingIndicator display the loading state in screen **/
    private val _loadingIndicator by lazy { MutableLiveData<Boolean>() }
    val loadingIndicator = _loadingIndicator

    private val _examplesList by lazy { MutableLiveData<List<ExampleVO>>() }
    val examplesList = _examplesList
```

Mejoras a aplicar:

3.1) Strong coupling (run()/prepareFlow()/DomainError)

- *Difficult to comprehend (FindUsages)*
- *Not simple and not always needed (SharedPreferences)*

```
abstract class FlowUseCase<T, R>(protected open val dispatcherProvider: DispatcherProvider) {

    protected open fun dispatcher() : CoroutineContext = dispatcherProvider.io()

    @CheckResult
    fun prepare(param: T) = prepareFlow(param).flowOn(dispatcher())

    protected abstract fun prepareFlow(param: T): Flow<R>
}
```

```
class LoadFilterByIdUseCase(
    private val filterRepository: FilterRepository,
    dispatcherProvider: DispatcherProvider
) : FlowUseCase<String, Either<Filter, DomainError>>(dispatcherProvider) {

    override fun prepareFlow(param: String): Flow<Either<Filter, DomainError>> = flow {
        emit(filterRepository.getFilter(param))
    }
}
```

```
abstract class UseCaseNoParams<Result> {
    protected abstract suspend fun run(): Flow<Result>
    fun invoke(
        scope: CoroutineScope = GlobalScope,
        dispatcher: CoroutineDispatcher = Dispatchers.Default,
        onResult: (Result) -> Unit = {}
    ) {
        val job = scope.async(dispatcher) { run() }
        scope.launch(Dispatchers.Main) { this: CoroutineScope
            job.await().collect { result ->
                onResult(result)
            }
        }
    }
}

abstract class UseCaseNoResult<Params> {
    protected abstract suspend fun run(params: Params)
    fun invoke(
        scope: CoroutineScope = GlobalScope,
        dispatcher: CoroutineDispatcher = Dispatchers.Default,
        params: Params,
    ) {
        scope.launch(dispatcher) { run(params) }
    }
}

abstract class UseCase<Params, Result> {
    protected abstract suspend fun run(params: Params): Flow<Result>
    fun invoke(
        scope: CoroutineScope = GlobalScope,
        params: Params,
        dispatcher: CoroutineDispatcher = Dispatchers.Default,
        onResult: (Result) -> Unit = {}
    ) {
        val job = scope.async(dispatcher) { run(params) }
        scope.launch(Dispatchers.Main) { this: CoroutineScope
            job.await().collect { result ->
                onResult(result)
            }
        }
    }
}
```


https://axinic.central.inditex.grp/bitbucket/scm/psintand/main.git

https://github.com/inditex/mob-rfidsuppliercontrol

Mejoras a aplicar:

3.1) Strong coupling (run()/prepareFlow()/DomainError)

- *Difficult to comprehend (FindUsages)*
- *Not simple and not always needed (SharedPreferences)*

```
abstract class FlowUseCase<T, R>(protected open val dispatcherProvider: DispatcherProvider) {

    protected open fun dispatcher() : CoroutineContext = dispatcherProvider.io()

    @CheckResult
    fun prepare(param: T) = prepareFlow(param).flowOn(dispatcher())

    protected abstract fun prepareFlow(param: T): Flow<R>
}
```

```
class LoadFilterByIdUseCase(
    private val filterRepository: FilterRepository,
    dispatcherProvider: DispatcherProvider
) : FlowUseCase<String, Either<Filter, DomainError>>(dispatcherProvider) {

    override fun prepareFlow(param: String): Flow<Either<Filter, DomainError>> = flow {
        emit(filterRepository.getFilter(param))
    }
}
```

Strict Generic UseCase** VS **Context based UseCase

```
abstract class UseCaseNoParams<Result> {
    protected abstract suspend fun run(): Flow<Result>
    fun invoke(
        scope: CoroutineScope = GlobalScope,
        dispatcher: CoroutineDispatcher = Dispatchers.Default,
        onResult: (Result) -> Unit = {}
    ) {
        val job = scope.async(dispatcher) { run() }
        scope.launch(Dispatchers.Main) { this: CoroutineScope
            job.await().collect { result ->
                onResult(result)
            }
        }
    }
}

abstract class UseCaseNoResult<Params> {
    protected abstract suspend fun run(params: Params)
    fun invoke(
        scope: CoroutineScope = GlobalScope,
        dispatcher: CoroutineDispatcher = Dispatchers.Default,
        params: Params,
    ) {
        scope.launch(dispatcher) { run(params) }
    }
}

abstract class UseCase<Params, Result> {
    protected abstract suspend fun run(params: Params): Flow<Result>
    fun invoke(
        scope: CoroutineScope = GlobalScope,
        params: Params,
        dispatcher: CoroutineDispatcher = Dispatchers.Default,
        onResult: (Result) -> Unit = {}
    ) {
        val job = scope.async(dispatcher) { run(params) }
        scope.launch(Dispatchers.Main) { this: CoroutineScope
            job.await().collect { result ->
                onResult(result)
            }
        }
    }
}
```


https://axinic.central.inditex.grp/bitbucket/scm/psintand/main.git

https://github.com/inditex/mob-rfidsuppliercontrol

Mejoras a aplicar:

3.1) Strong coupling (run()/prepareFlow()/DomainError)

- *Difficult to comprehend (FindUsages)*
- *Not simple and not always needed (SharedPreferences)*

```
abstract class FlowUseCase<T, R>(protected open val dispatcherProvider: DispatcherProvider) {

    protected open fun dispatcher() : CoroutineContext = dispatcherProvider.io()

    @CheckResult
    fun prepare(param: T) = prepareFlow(param).flowOn(dispatcher())

    protected abstract fun prepareFlow(param: T): Flow<R>
}
```

```
class LoadFilterByIdUseCase(
    private val filterRepository: FilterRepository,
    dispatcherProvider: DispatcherProvider
) : FlowUseCase<String, Either<Filter, DomainError>>(dispatcherProvider) {

    override fun prepareFlow(param: String): Flow<Either<Filter, DomainError>> = flow {
        emit(filterRepository.getFilter(param))
    }
}
```

DO NOT MIX ALL-IN-ONE:

Generic DomainErrors

VS

Layered Errors

(APIErrors, RepositoryErrors, UseCaseErrors)

```
sealed class DomainError {
    object StoreNotFoundError : DomainError()
    object MissingSettingError : DomainError()
    object LocalStorageError : DomainError()
    object TimeoutError : DomainError()
    object ServerError : DomainError()
    object ConnectivityError : DomainError()
    object EmptyBodyError : DomainError()
    object UnauthorizedError : DomainError()
    object ConflictError : DomainError()
    object AppNotFoundError : DomainError()
    object SledNotConnected: DomainError()
    object OnTagNotFound: DomainError()
    object OnInvalidTag: DomainError()
    object MDMUrlNotFound : DomainError()
    object CtBaseUrlNotFound : DomainError()

    data class OnInvalidBarcode(val invalidBarcode: String): DomainError()
    data class GenericExceptionError(val message: String?) : DomainError()
    data class GenericServerError(val code: String?, val cause: String?, val description: String?)
        : DomainError()
}
```

To improve:

```
data class ExampleApiException (val errorMessage: String? = null) : Exception() {

    companion object {
        const val EMPTY_RESULT = "EMPTY_RESULT"
        const val UNKNOWN = "UNKNOWN"
    }
}
```

Mejoras a aplicar:

3.2) Unnecessary usage of UseCase abstraction

- *Use only when the viewmodel gets too complex*

```
@HiltViewModel
class NotesViewModel @Inject constructor(
    private val noteUseCases: NoteUseCases
) : ViewModel() {

    private var getNotesJob: Job? = null

    init {
        getNotes(NoteOrder.Date(OrderType.Descending))
    }
```

```
class HomeViewModel(
    private val loadHomePickingInfoUseCase: LoadHomePickingInfoUseCase,
    private val loadAllHomePickingInfoUseCase: LoadHomeInfoUseCase,
    private val loadBottomSheetMenuUseCase: LoadBottomSheetMenuUseCase,
    private val loadSelectedFiltersUseCase: LoadSelectedFiltersUseCase,
    private val deleteSelectedFilterUseCase: DeleteSelectedFilterUseCase,
    private val createWaveUseCase: CreateWaveUseCase,
    private val loadActiveWavesUseCase: LoadActiveWavesUseCase,
    private val toggleFilterStateUseCase: ToggleFilterStateUseCase,
    private val disableRfidBarcodeModeUseCase: DisableRfidBarcodeModeUseCase,
    private val loadWaveUnitsUseCase: LoadWaveUnitsUseCase,
    override val sharedDataInterface: SharedDataInterface
) : BaseViewModel(), SharedDataHolder {
```


https://axinic.central.inditex.grp/bitbucket/scm/psintand/main.git

https://github.com/philipplackner/CleanArchitectureNoteApp/tree/testing

Mejoras a aplicar:

3.2) Unnecessary usage of UseCase abstraction

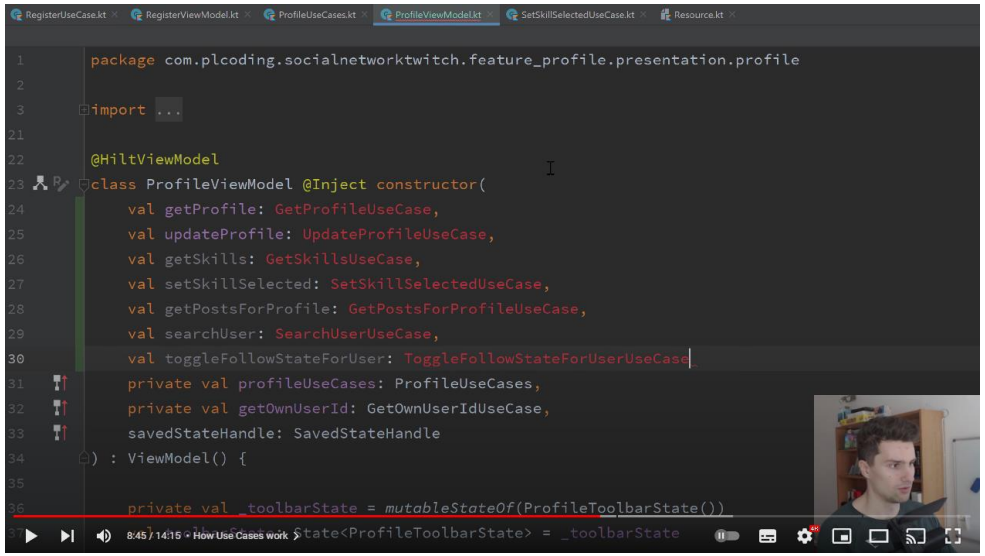
- *Use only when the viewmodel gets too complex*
- *If used, make it contextual based and not API/Repo based*

```kotlin
class HomeViewModel(
    private val loadHomePickingInfoUseCase: LoadHomePickingInfoUseCase,
    private val loadAllHomePickingInfoUseCase: LoadHomeInfoUseCase,
    private val loadBottomSheetMenuUseCase: LoadBottomSheetMenuUseCase,
    private val loadSelectedFiltersUseCase: LoadSelectedFiltersUseCase,
    private val deleteSelectedFilterUseCase: DeleteSelectedFilterUseCase,
    private val createWaveUseCase: CreateWaveUseCase,
    private val loadActiveWavesUseCase: LoadActiveWavesUseCase,
    private val toggleFilterStateUseCase: ToggleFilterStateUseCase,
    private val disableRfidBarcodeModeUseCase: DisableRfidBarcodeModeUseCase,
    private val loadWaveUnitsUseCase: LoadWaveUnitsUseCase,
    override val sharedDataInterface: SharedDataInterface
) : BaseViewModel(), SharedDataHolder {
```

```kotlin
@HiltViewModel
class NotesViewModel @Inject constructor(
    private val noteUseCases: NoteUseCases
) : ViewModel() {

    private var getNotesJob: Job? = null

    init {
        getNotes(NoteOrder.Date(OrderType.Descending))
    }
```



https://axinic.central.inditex.grp/bitbucket/scm/psintand/main.git

https://www.youtube.com/watch?v=ufq3LJ-5P6s

Mejoras a aplicar:

3.3) How it should be in Domain

- *No BaseUseCases*
- *Couple by context (all-in-one UseCases file)*

```
data class NoteUseCases(
    val getNotes: GetNotes,
    val deleteNote: DeleteNote,
    val addNote: AddNote,
    val getNote: GetNote
)
```

- *UseCase exemple with no "UseCase" suffix*
- *Repository injected*
- *Result obtained with no effort*

```
class GetNotes(
    private val repository: NoteRepository
) {
    operator fun invoke(
        noteOrder: NoteOrder = NoteOrder.Date(OrderType.Descending)
    ): Flow<List<Note>> {
        return repository.getNotes().map { notes ->
            //...
            emptyList()
        }
    }
}
```

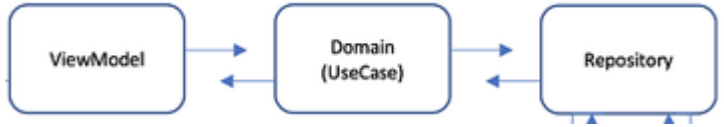
- *Usages are readable and trackable*
- *Number of constructor dependencies decrease*

```
@HiltViewModel
class NotesViewModel @Inject constructor(
    private val noteUseCases: NoteUseCases
) : ViewModel() {

    private var getNotesJob: Job? = null

    init {
        getNotes(NoteOrder.Date(OrderType.Descending))
    }

    private fun getNotes(noteOrder: NoteOrder) {
        getNotesJob?.cancel()
        getNotesJob = noteUseCases.getNotes(noteOrder)
            .onEach { notes ->
                _state.value = state.value.copy(
                    notes = notes,
                    noteOrder = noteOrder
                )
            }
            .launchIn(viewModelScope)
    }
```



- *UseCases injection example*

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {
    @Provides
    @Singleton
    fun provideNoteUseCases(repository: NoteRepository): NoteUseCases {
        return NoteUseCases(
            getNotes = GetNotes(repository),
            deleteNote = DeleteNote(repository),
            addNote = AddNote(repository),
            getNote = GetNote(repository)
        )
    }
    @Provides
    @Singleton
    fun provideNoteRepository(db: NoteDatabase): NoteRepository {
        return NoteRepositoryImpl(db.noteDao)
    }

    @Provides
    @Singleton
    fun provideNoteDatabase(app: Application): NoteDatabase {
        return Room.databaseBuilder(
            app,
            NoteDatabase::class.java,
            NoteDatabase.DATABASE_NAME
        ).build()
    }
}
```

https://github.com/philipplackner/CleanArchitectureNoteApp/tree/testing

Mejoras a aplicar:

3.3) How it should be in Domain

- *No BaseUseCases*
- *Couple by context (all-in-one UseCases file)*

```
data class NoteUseCases(
    val getNotes: GetNotes,
    val deleteNote: DeleteNote,
    val addNote: AddNote,
    val getNote: GetNote
)
```

- *UseCase exemple with no "UseCase" suffix*
- *Repository injected*
- *Result obtained with no effort*

```
class GetNotes(
    private val repository: NoteRepository
) {
    operator fun invoke(
        noteOrder: NoteOrder = NoteOrder.Date(OrderType.Descending)
    ): Flow<List<Note>> {
        return repository.getNotes().map { notes ->
            //...
            emptyList()
        }
    }
}
```

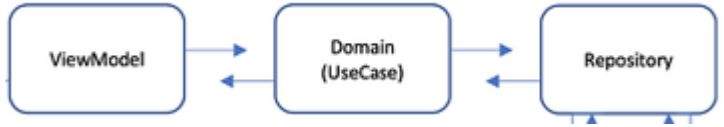
- *Usages are readable and trackable*
- *Number of constructor dependencies decrease*

```
@HiltViewModel
class NotesViewModel @Inject constructor(
    private val noteUseCases: NoteUseCases
) : ViewModel() {

    private var getNotesJob: Job? = null

    init {
        getNotes(NoteOrder.Date(OrderType.Descending))
    }

    private fun getNotes(noteOrder: NoteOrder) {
        getNotesJob?.cancel()
        getNotesJob = noteUseCases.getNotes(noteOrder)
            .onEach { notes ->
                _state.value = state.value.copy(
                    notes = notes,
                    noteOrder = noteOrder
                )
            }
            .launchIn(viewModelScope)
    }
```



- *UseCases injection example*

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {
    @Provides
    @Singleton
    fun provideNoteUseCases(repository: NoteRepository): NoteUseCases {
        return NoteUseCases(
            getNotes = GetNotes(repository),
            deleteNote = DeleteNote(repository),
            addNote = AddNote(repository),
            getNote = GetNote(repository)
        )
    }
    @Provides
    @Singleton
    fun provideNoteRepository(db: NoteDatabase): NoteRepository {
        return NoteRepositoryImpl(db.noteDao)
    }

    @Provides
    @Singleton
    fun provideNoteDatabase(app: Application): NoteDatabase {
        return Room.databaseBuilder(
            app,
            NoteDatabase::class.java,
            NoteDatabase.DATABASE_NAME
        ).build()
    }
}
```

https://github.com/philipplackner/CleanArchitectureNoteApp/tree/testing

Mejoras a aplicar:

3.3) How it should be in Domain

- *No BaseUseCases*
- *Couple by context (all-in-one UseCases file)*

```
data class NoteUseCases(
    val getNotes: GetNotes,
    val deleteNote: DeleteNote,
    val addNote: AddNote,
    val getNote: GetNote
)
```

- *UseCase exemple with no "UseCase" suffix*
- *Repository injected*
- *Result obtained with no effort*

```
class GetNotes(
    private val repository: NoteRepository
) {
    operator fun invoke(
        noteOrder: NoteOrder = NoteOrder.Date(OrderType.Descending)
    ): Flow<List<Note>> {
        return repository.getNotes().map { notes ->
            //...
            emptyList()
        }
    }
}
```

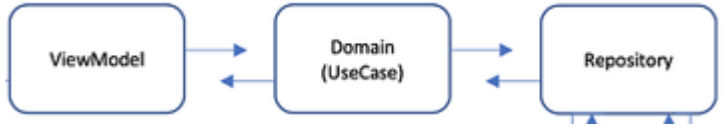
- *Usages are readable and trackable*
- *Number of constructor dependencies decrease*

```
@HiltViewModel
class NotesViewModel @Inject constructor(
    private val noteUseCases: NoteUseCases
) : ViewModel() {

    private var getNotesJob: Job? = null

    init {
        getNotes(NoteOrder.Date(OrderType.Descending))
    }

    private fun getNotes(noteOrder: NoteOrder) {
        getNotesJob?.cancel()
        getNotesJob = noteUseCases.getNotes(noteOrder)
            .onEach { notes ->
                _state.value = state.value.copy(
                    notes = notes,
                    noteOrder = noteOrder
                )
            }
            .launchIn(viewModelScope)
    }
```



- *UseCases injection example*

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {
    @Provides
    @Singleton
    fun provideNoteUseCases(repository: NoteRepository): NoteUseCases {
        return NoteUseCases(
            getNotes = GetNotes(repository),
            deleteNote = DeleteNote(repository),
            addNote = AddNote(repository),
            getNote = GetNote(repository)
        )
    }
    @Provides
    @Singleton
    fun provideNoteRepository(db: NoteDatabase): NoteRepository {
        return NoteRepositoryImpl(db.noteDao)
    }

    @Provides
    @Singleton
    fun provideNoteDatabase(app: Application): NoteDatabase {
        return Room.databaseBuilder(
            app,
            NoteDatabase::class.java,
            NoteDatabase.DATABASE_NAME
        ).build()
    }
}
```

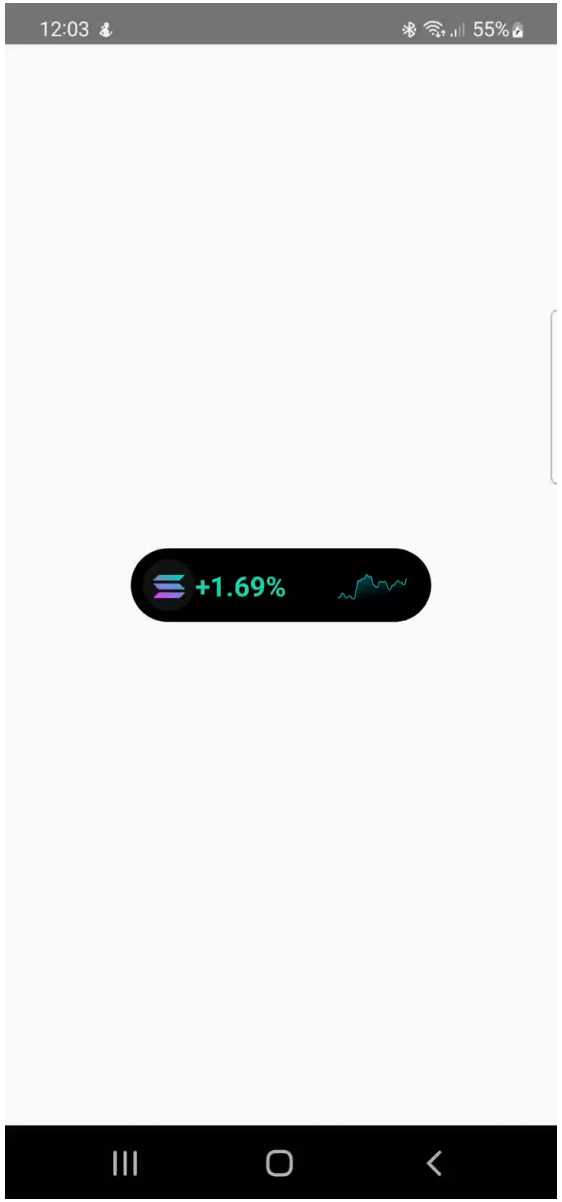
https://github.com/philipplackner/CleanArchitectureNoteApp/tree/testing

Jetpack Compose:

- *It is time to get back on track with Google..*
  - *Recommended adoption!*
    - *(as Kotlin was in the past)*

- *Why adopt Compose?*
  - *"Jetpack Compose is Android's modern toolkit for building native UI. It simplifies and accelerates UI development on Android bringing your apps to life with less code, powerful tools, and intuitive Kotlin APIs."*
    - *Less code*
    - *Intuitive*
    - *Accelerate development*
    - *Powerful*

- *Video demostration of MotionLayout animations*

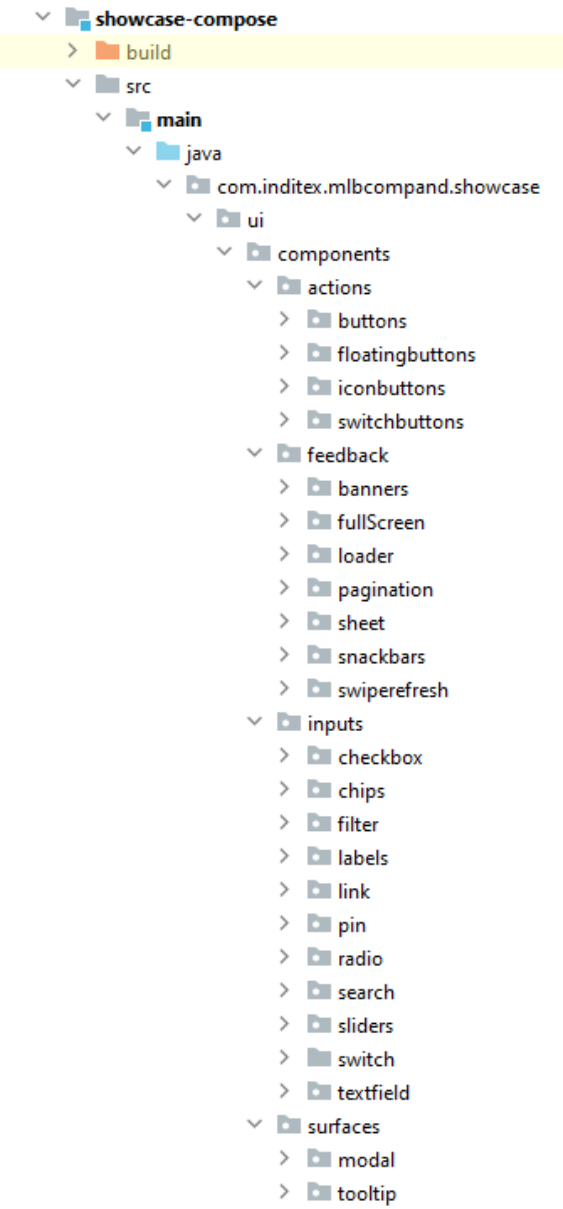


https://developer.android.com/jetpack/compose/why-adopt

https://twitter.com/RodrigoMartinD/status/1593259423312773120

Librería mlb-itxcomponentsandroid:

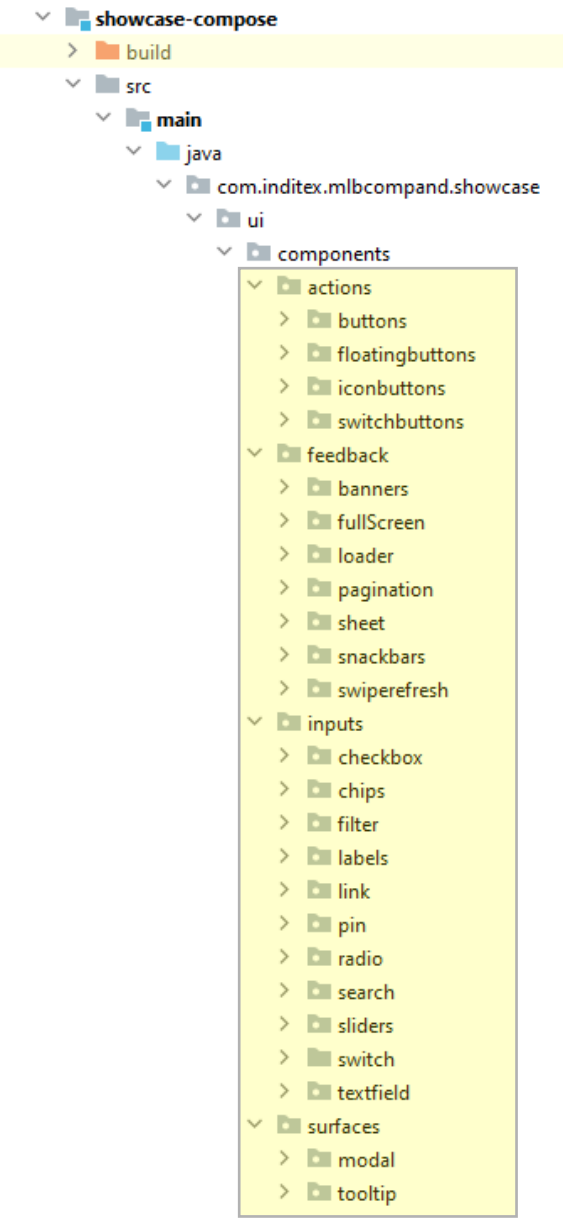
- Libreria rica en componentes
  - *Made in INDITEX*
- *"An atomic-based design guide has been followed for ease of use and maintenance."*
  - *Easy integration*
- *Documentation:* https://mlbcompand.docs.inditex.dev/mlbcompand/latest/home.html



https://github.com/inditex/mlb-itxcomponentsandroid

Librería mlb-itxcomponentsandroid:

- Libreria rica en componentes
  - *Made in INDITEX*
- *"An atomic-based design guide has been followed for ease of use and maintenance."*
  - *Easy integration*
- *Documentation:* https://mlbcompand.docs.inditex.dev/mlbcompand/latest/home.html
- *Gains:*
  - *Already implemented*
    - *Faster development*
  - *Design already certified*
  - *Use what we produce*
    - *Contrast impact*



https://github.com/inditex/mlb-itxcomponentsandroid

Conclusión:

- *When coding, always vouch for good:*
  - *Compreehensibility*
  - *Flexibility*
  - *Low error proneness*

Conclusión:

- *When coding, always vouch for good:*
  - *Compreehensibility*
  - *Flexibility*
  - *Low error proneness*
- *Make good use of Principles and Patterns*
  - *Android*
  - *Arquitecture*
  - *SOLID*

Conclusión:

- *When coding, always vouch for good:*
  - *Compreehensibility*
  - *Flexibility*
  - *Low error proneness*
- *Make good use of Principles and Patterns*
  - *Android*
  - *Arquitecture*
  - *SOLID*
- *Google exemples are just guidelines*
  - *Decide what should be used based on the project design and requirements.*
  - *Be minimalist.*

Conclusión:

- *When coding, always vouch for good:*
  - *Compreehensibility*
  - *Flexibility*
  - *Low error proneness*
- *Make good use of Principles and Patterns*
  - *Android*
  - *Arquitecture*
  - *SOLID*
- *Google exemples are just guidelines*
  - *Decide what should be used based on the project design and requirements.*
  - *Be minimalist.*
- *Test-Driven-Development (TDD) must be promoted*
  - *Refactor to make the code testable*
  - *Make the app reliable, fully usable, and easy to extend with new features*

Conclusión:

- *When coding, always vouch for good:*
  - *Compreehensibility*
  - *Flexibility*
  - *Low error proneness*
- *Make good use of Principles and Patterns*
  - *Android*
  - *Arquitecture*
  - *SOLID*
- *Google exemples are just guidelines*
  - *Decide what should be used based on the project design and requirements.*
  - *Be minimalist.*
- *Test-Driven-Development (TDD) must be promoted*
  - *Refactor to make the code testable*
  - *Make the app reliable, fully usable, and easy to extend with new features*
- *We must be aware of Android's future*
  - *Jetpack Compose is now 100% adopted and recommended by Android*

Conclusión:

- *When coding, always vouch for good:*
  - *Compreehensibility*
  - *Flexibility*
  - *Low error proneness*
- *Make good use of Principles and Patterns*
  - *Android*
  - *Arquitecture*
  - *SOLID*
- *Google exemples are just guidelines*
  - *Decide what should be used based on the project design and requirements.*
  - *Be minimalist.*
- *Test-Driven-Development (TDD) must be promoted*
  - *Refactor to make the code testable*
  - *Make the app reliable, fully usable, and easy to extend with new features*
- *We must be aware of Android's future*
  - *Jetpack Compose is now 100% adopted and recommended by Android*
- *Import the libraries our teams develop*
  - *Faster development and delivery*

Conclusión:

- *When coding, always vouch for good:*
  - *Compreehensibility*
  - *Flexibility*
  - *Low error proneness*
- *Make good use of Principles and Patterns*
  - *Android*
  - *Arquitecture*
  - *SOLID*
- *Google exemples are just guidelines*
  - *Decide what should be used based on the project design and requirements.*
  - *Be minimalist.*
- *Test-Driven-Development (TDD) must be promoted*
  - *Refactor to make the code testable*
  - *Make the app reliable, fully usable, and easy to extend with new features*
- *We must be aware of Android's future*
  - *Jetpack Compose is now 100% adopted and recommended by Android*
- *Import the libraries our teams develop*
  - *Faster development and delivery*

Daniel Moka @dmokafa · 20 de abr

If your code has no tests, then:

- it is not clean
- it is not complete
- it is not correct
- it is not reliable
- it is not documented
- it is not refactorable
- it is not verified

And most importantly: it is not quality.