Model-View-ViewModel
LiveData

Peter Borovanský
KAI, I-18

MS-Teams: 2sf3ph4, List, github

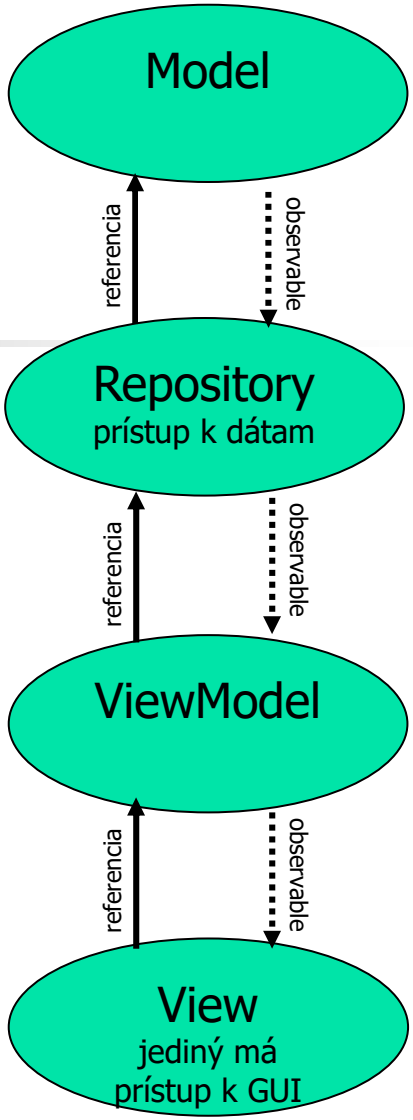
borovan 'at' ii.fmph.uniba.sk

Kap. 39 – 46 Modern Android Architecture with JetPack
Kap. 47 – 48 Navigation Architecture Component

Plán

- Model View Presenter (MVP)
- Model View Controller (MVC)
- Model View ViewModel (MVVM)
  - LiveData
  - DataBindings

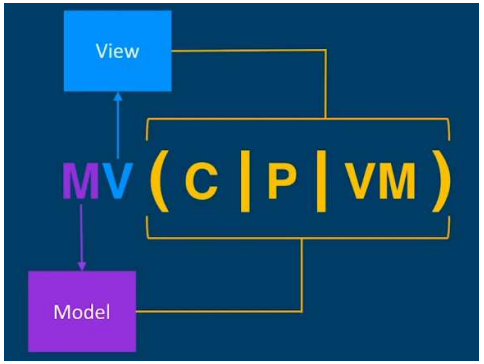
- Cvičenie - malé príklady:
  - konvertovacia kalkulačka
  - Pikatchus
  - hodinky



Alternatíva:

https://codelabs.developers.google.com/codelabs/kotlin-android-training-view-model/

MV [C | P | VM]



atribúty dobrého kódu

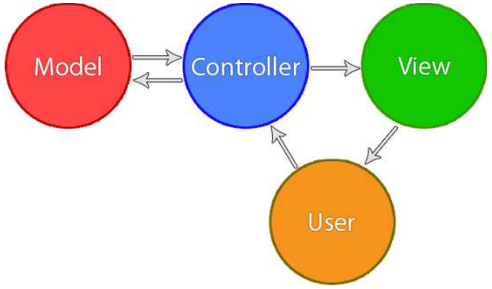
- stabilný – k drobným zmenám
- robustný – keď sa zväčšuje, komplikuje, vyvíja
- testovateľný nezávisle – GUI, aj Model
- modulárny

Tri bežne používané návrhové vzory: model-view

- Controller
- Presenter
- ViewModel

Majú spoločné:

- Model – implementuje tzv. business logic
  - cez Repository komunikuje s databázou, internetom, ...
  - vystavuje svoje dáta, komukoľvek, kto ich potrebuje
  - nemá nič spoločné s androidom, môžete k nemu napísať sériu j-unit testov
- View – zobrazuje dáta

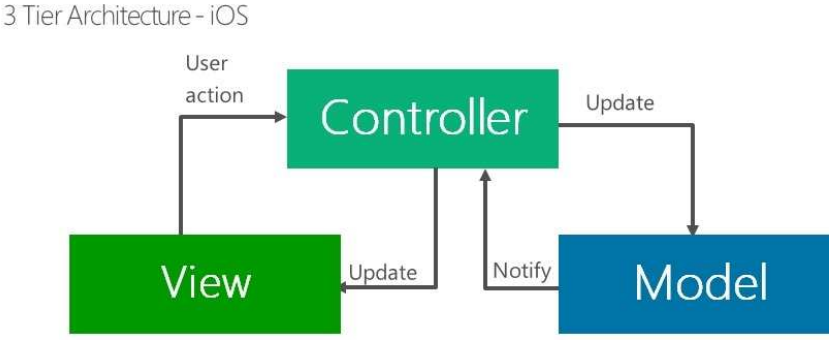


Architektonický *mess*
(MVC)

...vzniká, ak vizuálne komponenty (Views) sú v kóde zviazané s dátovými objektami a opačne, príklad:

```
prev.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        i++;
        i %= imgs.length;
        iv.setImageDrawable(imgs[i]);
    }
});
```

- preto sa pri návrhu GUI používajú návrhové vzory (design patterns)
  napr. Model-View-Controller



- motto: the architecture of most Android-apps *(in the pass time)* is a mess.
- Activita často reprezentuje rolu View aj Controllera

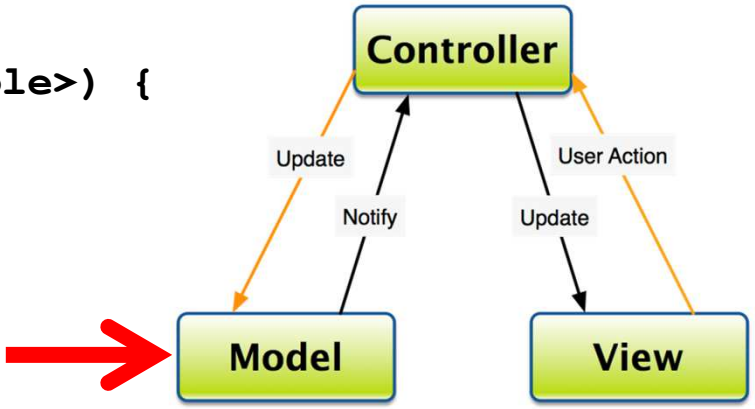
Model View Controller (MVC)

(Model sú len dáta netušiace nič o ich prezentácii)

```
class Model() : Observable() {
 private var indx = 0
 private var list = mutableListOf<Drawable>()

    fun addDrawables(imgs: List<Drawable>) {
        list.addAll(imgs)
    }
    val currentDrawable: Drawable
        get() = list[indx]

    fun nextValue() {
        indx++
        indx %= list.size
        setChanged()
        notifyObservers()
    }
```




```
fun prevValue() {
   indx--
   if (indx<0) indx = list.size-1
   setChanged()
   notifyObservers()
}
```

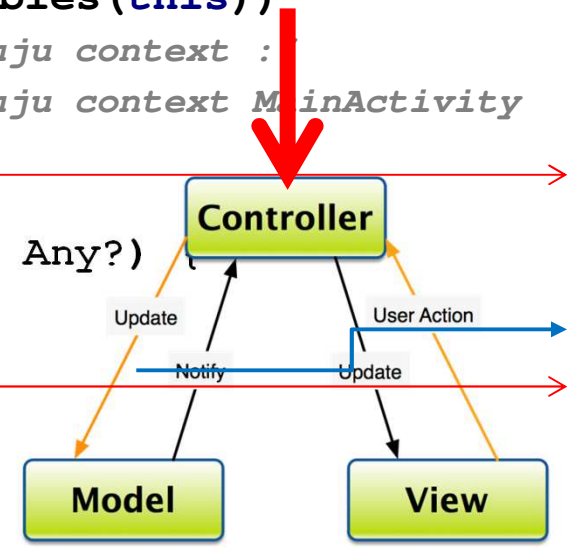
java.util.Observable
setChanged() - marks this Observable object as having been changed
notifyObservers()
notifyObservers(Object arg) - if hasChanged, then notify all of its observers and then call the clearChanged = no longer changed.

PikatchuMVC.zip

Model View **Controller** (MVC)

(Controller – komunikuje medzi modelom a view)

```
class MainActivity : AppCompatActivity(), Observer {
  lateinit var myModel: Model
  lateinit var myView: MyView
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    myModel = Model()           // inštancia business modelu
    myModel.addObserver(this) // this-Controler je observerom modelu
    myModel.addDrawables(Repository.allDravables(this))
                                // Repository potrebuju context :(
    myView = MyView(this) // views tiež potrebuju context MainActivity
  }
  // interface Observer
  override fun update(arg0: Observable,arg1: Any?) {
    myView.myupdate(myModel.currentDrawable)
  }
```



PikatchuMVC.zip

java.util.Observer
update(o : Observable, arg : Any?) - this method is called whenever the observed object is changed.

Model **View** Controller (MVC)

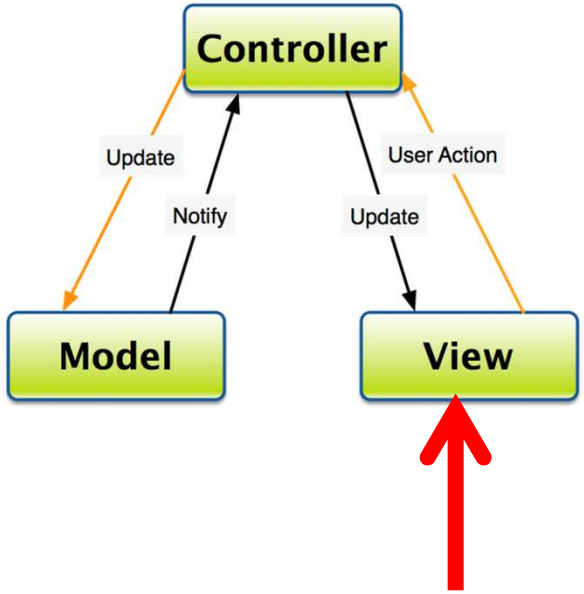
(View je GUI, zobrazenie Views, eventy)

View

- prezentuje dáta vo Views
- odchytáva eventy

```
class MyView(val main: MainActivity) {
                // pointer na Controller
  init {
    main.prevBtn.setOnClickListener {
        main.myModel.prevValue()
    }
    main.nextBtn.setOnClickListener {
        main.myModel.nextValue()
    }

myupdate(main.myModel.currentDrawable)
  }
  fun myupdate(im:Drawable) {
    main.imageView1.setImageDrawable(im)
  }
}
```



PikatchuMVC.zip

Model View Controller (MVC)
(Repository – sprístupňovač dát)

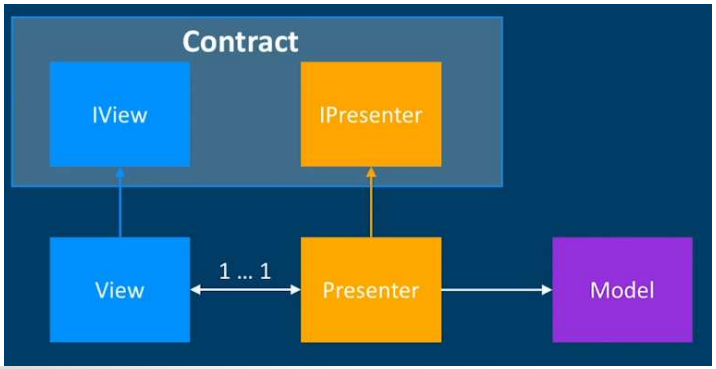
Repository jediné vie, či dáta
- sú lokálne
- sú z lokálnej databázy, napr. Room, resp. cloudovej databázy, napr. FireBase

alebo
- sú z netu, cez nejakú webovú službu, servis

```
class Repository {
  companion object {
    fun allDravables(context : Context) =
      listOf(
        R.drawable.butterfree,R.drawable.golbat,R.drawable.kakuna,
        R.drawable.raichu, R.drawable.venomoth, R.drawable.venusaur,
        R.drawable.pok0, R.drawable.pok1, R.drawable.pok2,
        R.drawable.pok3, R.drawable.pok4, R.drawable.pok5,
        R.drawable.pok6
      ).map {
         ContextCompat.getDrawable(
               context.applicationContext, it)!!
      }
    }
}
```

PikatchuMVC.zip

Model View Presenter




```
interface Login {
    interface View {
        fun setUserName(name : String)
        fun setPassword(passs : String)
        fun showValidationSuccessful()
        fun showValidationFailed()
        fun setPresenter(p : Login.Presenter)
    }
    interface Presenter {
        fun loginUser(name:String, pass:String)
    }
}
```

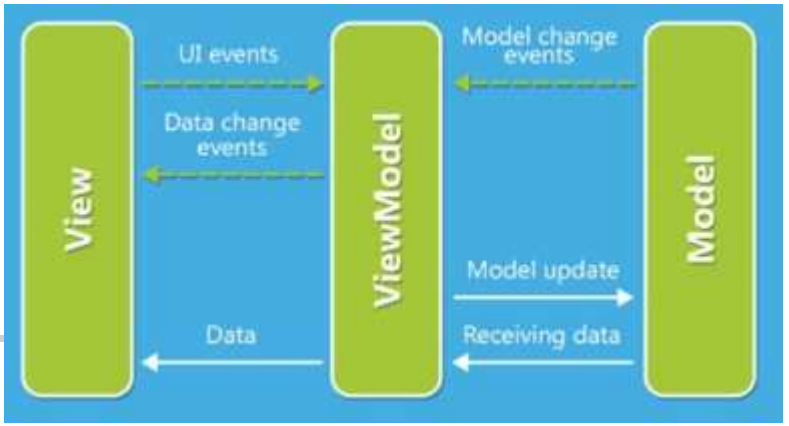
interface pre View

interface pre Presenter

```
class LoginView : Login.View {
  lateinit var mpresenter : Login.Presenter
  override fun setUserName(name: String) { }
  override fun setPassword(passs: String) { }
  override fun showValidationSuccessful() { }
  override fun showValidationFailed() { }
  override fun setPresenter(p: Login.Presenter) {
    mpresenter = p
  }
}
```

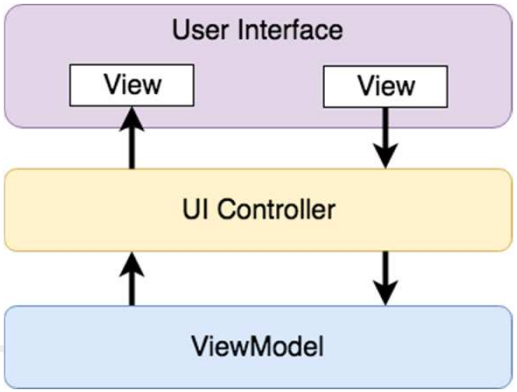
```
class LoginPresenter (view:Login.View):Login.Presenter {
  var mView : Login.View
  init {
    mView = view
    mView.setPresenter(this)
  }
  override fun loginUser(name: String, pass: String) { }
}
```

Čo je JetPack



- celý moderný vývoj iOS postavený na jazyku Swift je striktne založený na Model-View-Controller vzore (MVC)
- Model-View-Controller je založený na triedach Observable a Observer
- na mnohých príkladoch single activity apps sme videli, že sa mieša kód pre GUI s business logikou aplikácie
- Google si to uvedomil 2017 a navrhol JetPack pre multi-activity apps
- cieľom:
  - je oddeliť kód pre GUI od kódu s logikou
  - problémy so životným cyklom, napr. pri rotácii displaya
  - perzistenciu dát
- architektúra separácie GUI a logiky kódu založená na ViewModel, nie MVC
- MVVM pochádza od Microsoft, 2005
- ViewModel je analógia k Controlleru (MVC), či k Presenteru (MVP)
- ViewModel je také *lepidlo*, čo spája View a Model

Model View ViewModel



- ViewModel je jediný, čo vie o dátach a ich logike
- keď zmeníme GUI, ViewModel zostáva nezmenený
- ak sa zmení napr. orientácia, tak ViewModel stále drží pôvodné dáta

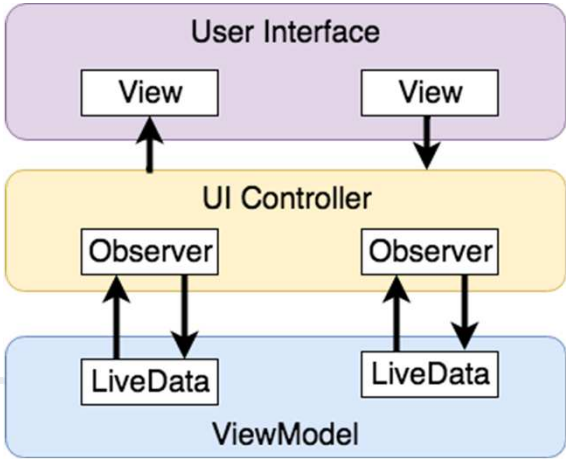
- View oznamuje ViewModelu, čo sa zmenilo, UI events
- View má referenciu na ViewModel, čo sa zmenilo, UI events
- ViewModel nemá žiadnu predstavu o View, len ponúka dáta (producer)
- View je v roli prijímateľa dát (consumer) a ViewModel sa nestará o to, kto dáta konzumuje, a či...

- dáta sa ale môžu meniť nezávisle na GUI, a aj často, napr. realtime data
- kedy sa má GUI dopytovať, či nemá dáta prekresliť, či sa náhodou nezmenili ?
- agresívne „spojité" pool-ovanie dát je náročné, tak sa to nerobí

https://www.techotopia.com/index.php/Modern_Android_App_Architecture_with_Jetpack

View ViewModel

(consumer producer)



Preto je na to generická trieda LiveData:

LiveData – Observer

- ViewModel vystavuje premennú LiveData<T>, resp. MutableLiveData<T>
- ktokoľvek kto sa stane observerom pre ViewModel sa dovzie o zmenách tejto premennej, teda observer dostane info, ak sa dáta zmenia
- ak aktivita-fragment prestane byť aktívny a opäť sa prebudí, dostane rovnaké dáta
- Ak aktivita-fragment zmení orientáciu, tak po zmene sa opäť obnovia jej pôvodné dáta

Výhody:

- nepríšete množstvo interface
- vzťah medzi View a ViewModel nie je silne zviazaný, ViewModel ani netuší, či a aké Views ho observujú (počúvajú)
- ergo, ViewModel sa ani nemusí zaujímať, či View ešte existuje, žije...

https://www.techotopia.com/index.php/Modern_Android_App_Architecture_with_Jetpack

Projekt Fragment+ViewModel

(verzia 1 – dostanete zadarmo)

```
class MainFragment : Fragment() {
    companion object {  // statická metoda
        fun newInstance() = MainFragment()
    }
    private lateinit var viewModel: MainViewModel

    override fun onCreateView(inflater: LayoutInflater,
                              container: ViewGroup?,
                              savedInstanceState: Bundle?): View {
        return inflater.inflate(R.layout.main_fragment, container, false)
    }
    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        viewModel = ViewModelProvider(this).get(MainViewModel::class.java)
        // TODO: Use the ViewModel
    }
}
```

```
import androidx.lifecycle.ViewModel

class MainViewModel : ViewModel() {
    // TODO: Implement the ViewModel
}
```

JetPack1.zip

Projekt Fragmet+ViewModel

(verzia 1 – ViewModel, ViewModelProvider)

```
class MainFragment : Fragment() {
  override fun onActivityCreated(savedInstanceState: Bundle?) {
    super.onActivityCreated(savedInstanceState)
    viewModel = ViewModelProvider(this).get(MainViewModel::class.java)
    convertBtn.setOnClickListener {
       if (inputAmount.text.isNotEmpty()) {
        viewModel.convertUSD2EURO = usd2euro.isChecked
         viewModel.setInputCurrencyAmount(inputAmount.text.toString())
        outputAmount.setText("%.2f".format(viewModel.outputCurrencyAmount))
}  }    }
```

```
class MainViewModel : ViewModel() {
    val dolar2euroRate = 1.1f
    var convertUSD2EURO = true
    var inputCurrencyAmount = 0f
    var outputCurrencyAmount = 0f

   fun setInputCurrencyAmount(value : String) {
        inputCurrencyAmount = value.toFloat()
        outputCurrencyAmount =
           if  (convertUSD2EURO) inputCurrencyAmount * dolar2euroRate
          else inputCurrencyAmount / dolar2euroRate
}  }
```

Pros:
máme oddelené views a dáta
Cons:
o GUI refresh sa staráme my

JetPack1.zip

LiveData

(verzia 2 – Observer, MutableLiveData<T>)

Pros:
observer sa automaticky dozvie o zmene premennej LiveData, na ktorú je priviazaný
Cons:
do GUI to musím explicitne zapísať my

```
class MainFragment : Fragment() {
  override fun onActivityCreated(savedInstanceState: Bundle?) {
    super.onActivityCreated(savedInstanceState)
    viewModel = ViewModelProvider(this).get(MainViewModel::class.java)
    val resultObserver = Observer<Float> {
      result -> outputAmount.setText("%.2f".format(result))
    }
    viewModel.outputCurrencyAmount.observe(viewLifecycleOwner,resultObserver)
    convertBtn.setOnClickListener {
      if (inputAmount.text.isNotEmpty()) {
        viewModel.convertUSD2EURO = usd2euro.isChecked
        viewModel.setInputCurrencyAmount(inputAmount.text.toString())
```

```
class MainViewModel : ViewModel() {
    val dolar2euroRate = 1.1f
    var convertUSD2EURO = true
    var inputCurrencyAmount = 0f
    var outputCurrencyAmount : MutableLiveData<Float> = MutableLiveData()
    fun setInputCurrencyAmount(value : String) {
       inputCurrencyAmount = value.toFloat()
       outputCurrencyAmount.value =
          if  (convertUSD2EURO) inputCurrencyAmount * dolar2euroRate
                                else inputCurrencyAmount / dolar2euroRate
```

JetPack2.zip

LiveData

(verzia 2++ viac kotlinish)

Pros:
observer sa automaticky dozvie o zmene premennej LiveData, na ktorú je priviazaný
Cons:
do GUI to musím explicitne zapísať my

```
class MainFragment : Fragment() {
  override fun onActivityCreated(savedInstanceState: Bundle?) {
     convertBtn.setOnClickListener {
      if (inputAmount.text.isNotEmpty()) {
         viewModel.convertUSD2EURO = usd2euro.isChecked
         viewModel.inputCurrencyAmount = inputAmount.text.toString().toFloat()
```

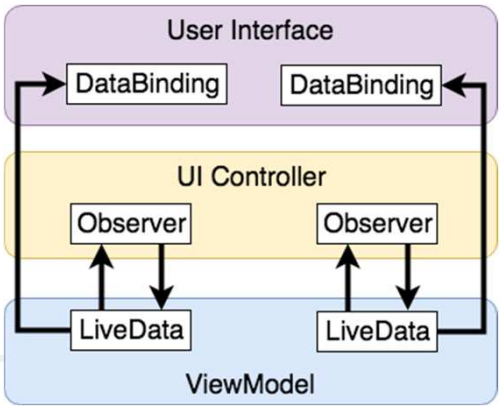
```
class MainViewModel : ViewModel() {
val dolar2euroRate = 1.1f
var convertUSD2EURO = true
    set(value) { field = value }

private val _outputCurrencyAmount:MutableLiveData<Float> = MutableLiveData()
val outputCurrencyAmount : LiveData<Float>
    get() = _outputCurrencyAmount

var inputCurrencyAmount = 0f
    set (value : Float) { field = value
        _outputCurrencyAmount.value =  if  (convertUSD2EURO)
                inputCurrencyAmount * dolar2euroRate
                else inputCurrencyAmount / dolar2euroRate
    }
```

JetPack2.zip

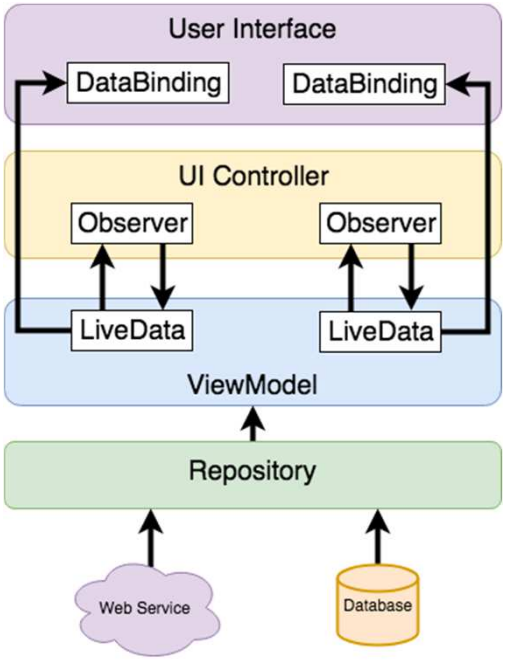
Data Binding



- ako zabezpečiť, aby sa dáta v observeri správne zobrazili v GUI
- ViewModel má priamo informáciu o konktrétnom view v .xml layout file, kde sa majú dáta zobraziť a refreshovať

Externé data:

- Repository slúži na dáta externých zdrojov
- je to vrstva, ktorá zakrýva pôvod, protokol dát



https://www.youtube.com/watch?v=2rO4r-JOQtA

https://www.techotopia.com/index.php/Modern_Android_App_Architecture_with_Jetpack

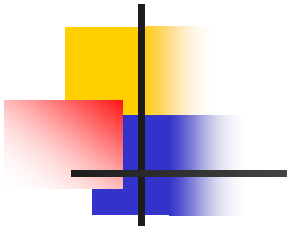
DataBinding
(build.gradle)

Neobjavuje, skopírujte do build.gradle

```
plugins {
    . . .
    id 'kotlin-android-extensions'
    id 'kotlin-kapt'
}
android {
    buildFeatures {
        dataBinding = true
    }
}
dependencies {
    annotationProcessor
        "com.android.databinding:compiler:$kotlin_version"
        ...
}
kapt {
    generateStubs = true
}
```

Kotlin annotation processor

JetPack3.zip



DataBinding

(fragment.xml)

```
<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools">
    <data>
        <variable
            name="myViewModel"
            type="com.example.jetpack3.ui.main.MainViewModel" />
    </data>
    <androidx.constraintlayout.widget.ConstraintLayout
      android:id="@+id/main"
      tools:context=".ui.main.MainFragment">

      <EditText
         android:text="@={myViewModel.inputCurrencyAmount}"
         android:hint="@string/input_currency_amount"/>

      <EditText
          android:id="@+id/outputAmount"
          android:text="@{String.valueOf(myViewModel.outputCurrencyAmount)}"
          android:text='@{safeUnbox(myViewModel.outputCurrencyAmount) == 0.0 ?"":
                         String.valueOf(safeUnbox(myViewModel.outputCurrencyAmount))}'/>

      <Button
          android:id="@+id/convertBtn"
          android:onClick="@{() -> myViewModel.convertValue()}" />

      <RadioGroup">
          <RadioButton
             android:id="@+id/usd2euro"
             android:checked="@={myViewModel.usd2euroChecked}"/>
          <RadioButton
             android:id="@+id/euro2usd"
             android:checked="@={myViewModel.euro2usdChecked}"/>
      </RadioGroup>
  </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

JetPack3.zip

DataBinding

(fragment.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android">
    <androidx.constraintlayout.widget.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:id="@+id/main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".ui.main.MainFragment">
        ...
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

JetPack3.zip

Data Binding

previazanie .xml komponentu s LiveData premennou

Binding Expressions má tvar "@{ ... }"

Jednosmerná väzba `@{..}`

- napr. Button, má zavolať zodpovedajúcu metódu pre onClickListener

```
android:onClick="@{() -> myViewModel.convertValue()}"
```

- hodnota z LiveData premennej sa má automaticky zobraziť vo View

```
android:text="@{String.valueOf(myViewModel.outputCurrency)}"
```

warning:

- `myViewModel.outputCurrency.getValue() is a boxed field but needs to be un-boxed to execute String.valueOf(viewModel.outputCurrency.getValue()).`

```
android:text='@{safeUnbox(myViewModel.outputCurrencyAmount) == 0.0 ?"":
        String.valueOf(safeUnbox(myViewModel.outputCurrencyAmount))}`
```

Dvojsmerná väzba `@={..}`

napr. EditText môže zmeniť MutableLiveData<>, a tiež naopak

```
android:text="@={myViewModel.inputCurrencyAmount}"
```

JetPack3.zip

DataBinding

(verzia 3 – databindings)

```
class MainFragment : Fragment() {
    private lateinit var viewModel: MainViewModel
    lateinit var binding : MainFragmentBinding
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View {

        binding = DataBindingUtil.inflate(inflater,
                              R.layout.main_fragment, container, false)
        binding.setLifecycleOwner(this)
        return binding.root
    }

    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        viewModel = ViewModelProvider(this).get(MainViewModel::class.java)
        binding.setVariable(myViewModel, viewModel)
    }
}
```

```
<data>
   <variable
      name="myViewModel"
      type="com.example.jetpack3.ui.main.MainViewModel" />
</data>
```

JetPack3.zip

DataBinding

(verzia 3 – databindings)

```
class MainViewModel : ViewModel() {
    val dolar2euroRate = 1.1f
    var usd2euroChecked : MutableLiveData<Boolean> = MutableLiveData()
    var euro2usdChecked : MutableLiveData<Boolean> = MutableLiveData()
    var inputCurrencyAmount : MutableLiveData<String> = MutableLiveData()
    var outputCurrencyAmount : MutableLiveData<Float> = MutableLiveData()

    fun convertValue() {
      inputCurrencyAmount.let {
        if ((it.value?:"").isNotEmpty()) {
          if (usd2euroChecked.value?:false)
            //outputCurrencyAmount.value=it.value?.toFloat()?.times(dolar2euroRate)
             outputCurrencyAmount.value = (it.value?:"0").toFloat() *
                                                  dolar2euroRate
          else
            //outputCurrencyAmount.value=it.value?.toFloat()?.div(dolar2euroRate)
            outputCurrencyAmount.value = (it.value?:"0").toFloat() / dolar2euroRate
        } else {
           outputCurrencyAmount.value = 0f
        }
}  } }
```

JetPack3.zip

Lifecycle a LiveData

Hodne zjednodušený main_fragment.xml

```
<layout
    <data>
        <variable
            name = "myViewModel"
            type = "com.example.jetpack4.ui.main.MainViewModel" />
    </data>
    <androidx.constraintlayout.widget.ConstraintLayout
    <Button
        android:onClick="@{() -> myViewModel.buttonClicked()}"
/>
    <EditText
        android:text="@={myViewModel.edittext}"
    />
    <TextView
        android:text="@{myViewModel.elapsedTime}"
    />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

JetPack4.zip

Lifecycle a LiveData

Synchrónna a asynchrónna zmena

```
class MainViewModel : ViewModel() {
    var edittext : MutableLiveData<String> = MutableLiveData("")
    var _elapsedTime = 0
    var elapsedTime:MutableLiveData<String> = MutableLiveData()
    init {
        object : CountDownTimer(100*1000, 1000) {
            override fun onTick(p0: Long) {
                _elapsedTime++
                elapsedTime.value =  _elapsedTime.toString()
            }
        }.start()
    }
    fun buttonClicked() {
        Log.d(TAG, "button clicked")
        edittext.value += "a"
}   }
```

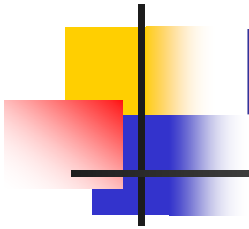
JetPack4.zip

Pikas MVVM

ViewModel

```
class PikaViewModel: ViewModel() {
    val index : MutableLiveData<Int> = MutableLiveData()
    val time : MutableLiveData<Int> = MutableLiveData()
    val currentImg: MutableLiveData<Drawable> = MutableLiveData()
    val finish: MutableLiveData<Boolean> = MutableLiveData()
    var list = mutableListOf<Drawable>()
    init {
        index.value = 0
        time.value = 0
        list = mutableListOf<Drawable>()
        object : CountDownTimer(20000,1000) {
            override fun onTick(p0: Long) {
                time.value =(time.value?:0)+1
            }
            override fun onFinish() {
                finish.value = true
            }
        }.start()
}   }
```

PikatchuMVVN.zip



Pikas MVVM

```
class MainFragment : Fragment() {
    private lateinit var viewModel: PikaViewModel
    lateinit var binding : MainFragmentBinding
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View {
        binding = DataBindingUtil.inflate(inflater,
                  R.layout.main_fragment, container, false)
        binding.setLifecycleOwner(this)
        return binding.root
    }
    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        viewModel = ViewModelProvider(this).get(PikaViewModel::class.java)
        binding.setVariable(pikaViewModel, viewModel)
        viewModel.addDrawables(Repository.allDravables(requireContext()))
        val finishObserver = Observer<Boolean> {
                  result -> activity?.finish()
        }
        viewModel.finish.observe(viewLifecycleOwner, finishObserver)
}   }
```

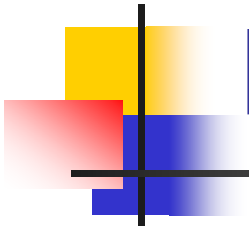
PikatchuMVVN.zip

Pikas MVVM

```
<?xml version="1.0" encoding="utf-8"?>

<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">
    <data>
        <variable name="pikaViewModel"
                  type="com.example.pikatchumvvm.PikaViewModel" />
    </data>
         ...
         <Button
                  android:onClick="@{()->pikaViewModel.prevValue()}"
         ...
         <Button
                  android:onClick="@{()->pikaViewModel.nextValue()}"
         ...
         <TextView
                  android:text="@{pikaViewModel.time.toString()}"
         <ImageView
                  android:imageDrawable="@{pikaViewModel.currentImg}"
</layout>
```

PikatchuMVVN.zip



Navigácia