Uniwersytet Wrocławski

Rafał Lewandków

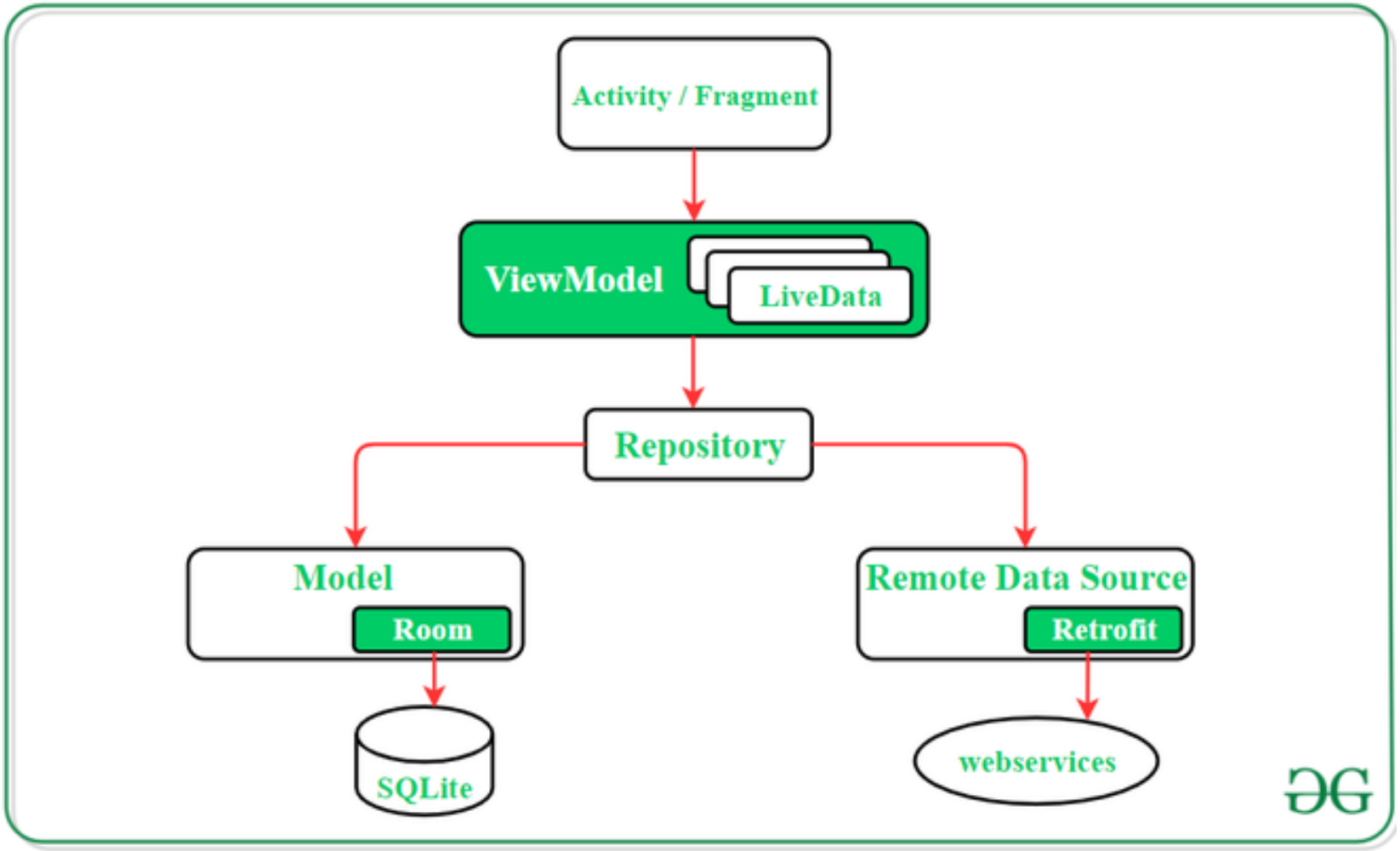
PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

WYKŁAD 8

Architektura Aplikacji 1

- ViewModel
- LiveData
- MVVM

MVVM


Activity / Fragment
ViewModel
LiveData
Repository
Model
Room
SQLite
Remote Data Source
Retrofit
webservices

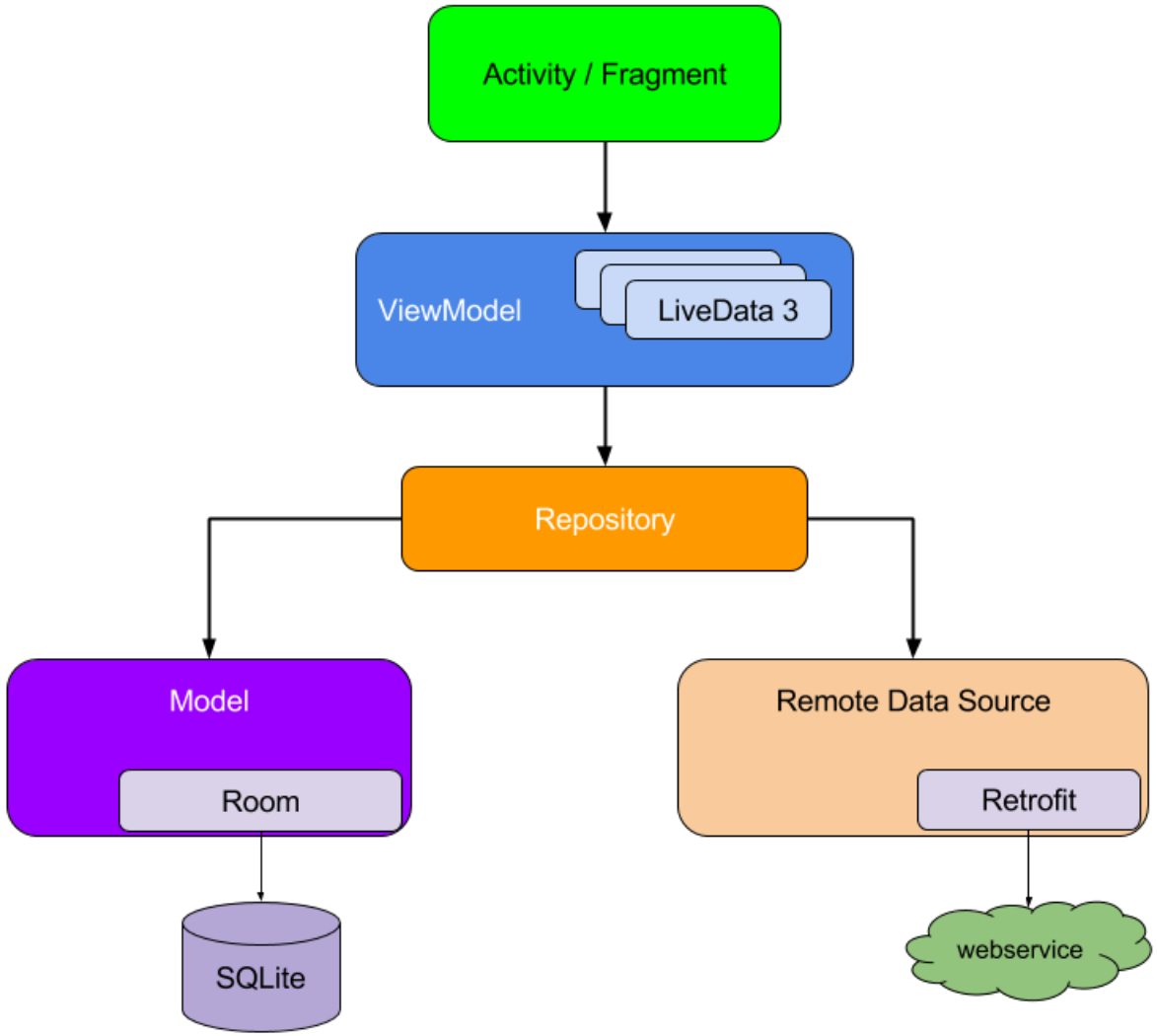
Uniwersytet Wrocławski

ViewModel

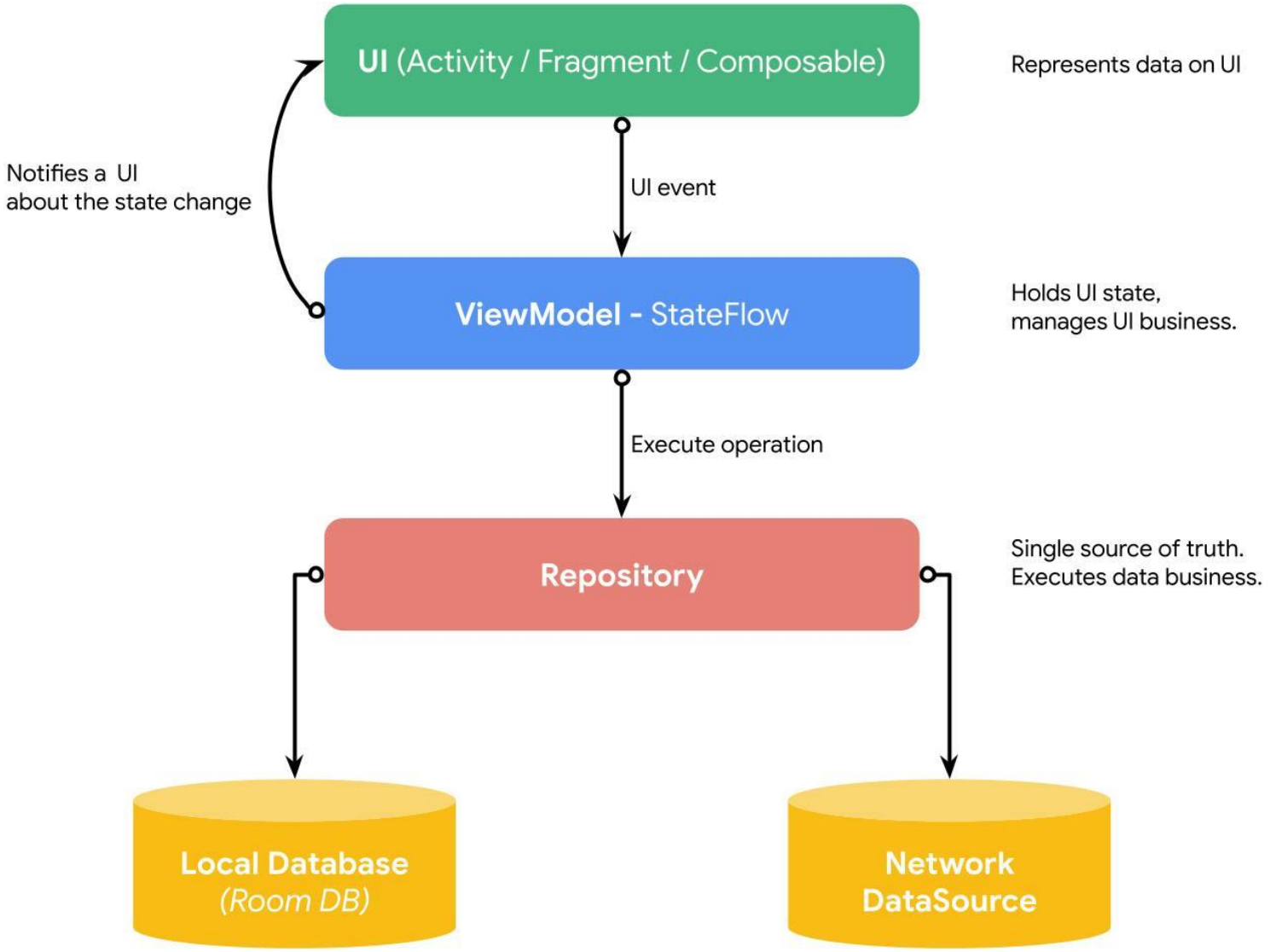
`ViewModel` to wzorzec projektowy stosowany w programowaniu, szczególnie w kontekście tworzenia aplikacji z interfejsem użytkownika. Celem `ViewModel` jest oddzielenie logiki biznesowej aplikacji od jej warstwy prezentacji.

ViewModel

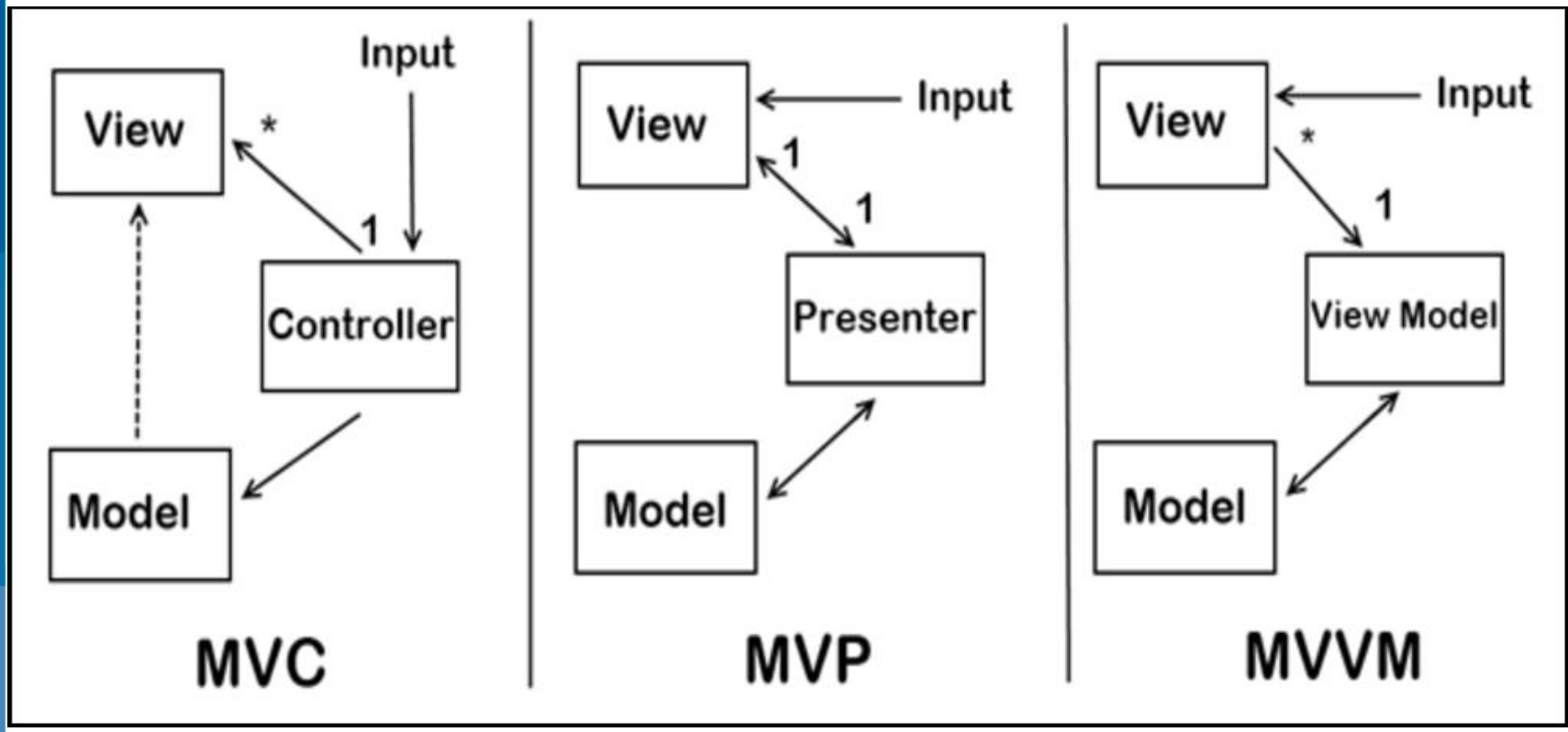
`ViewModel` to wzorzec projektowy stosowany w programowaniu, szczególnie w kontekście tworzenia aplikacji z interfejsem użytkownika. Celem `ViewModel` jest oddzielenie logiki biznesowej aplikacji od jej warstwy prezentacji.



ViewModel


UI (Activity / Fragment / Composable)
Represents data on UI
Notifies a UI
about the state change
UI event
ViewModel - StateFlow
Holds UI state,
manages UI business.
Execute operation
Repository
Single source of truth.
Executes data business.
Local Database
(Room DB)
Network
DataSource

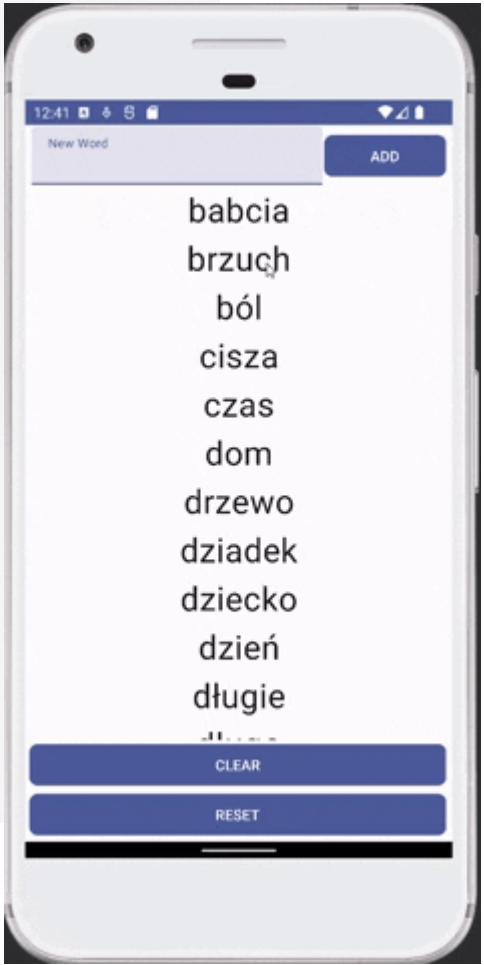
ViewModel


Input
View
*
1
Controller
Model
MVC
View
Input
1
1
Presenter
Model
MVP
View
Input
*
1
View Model
Model
MVVM

Uniwersytet Wrocławski

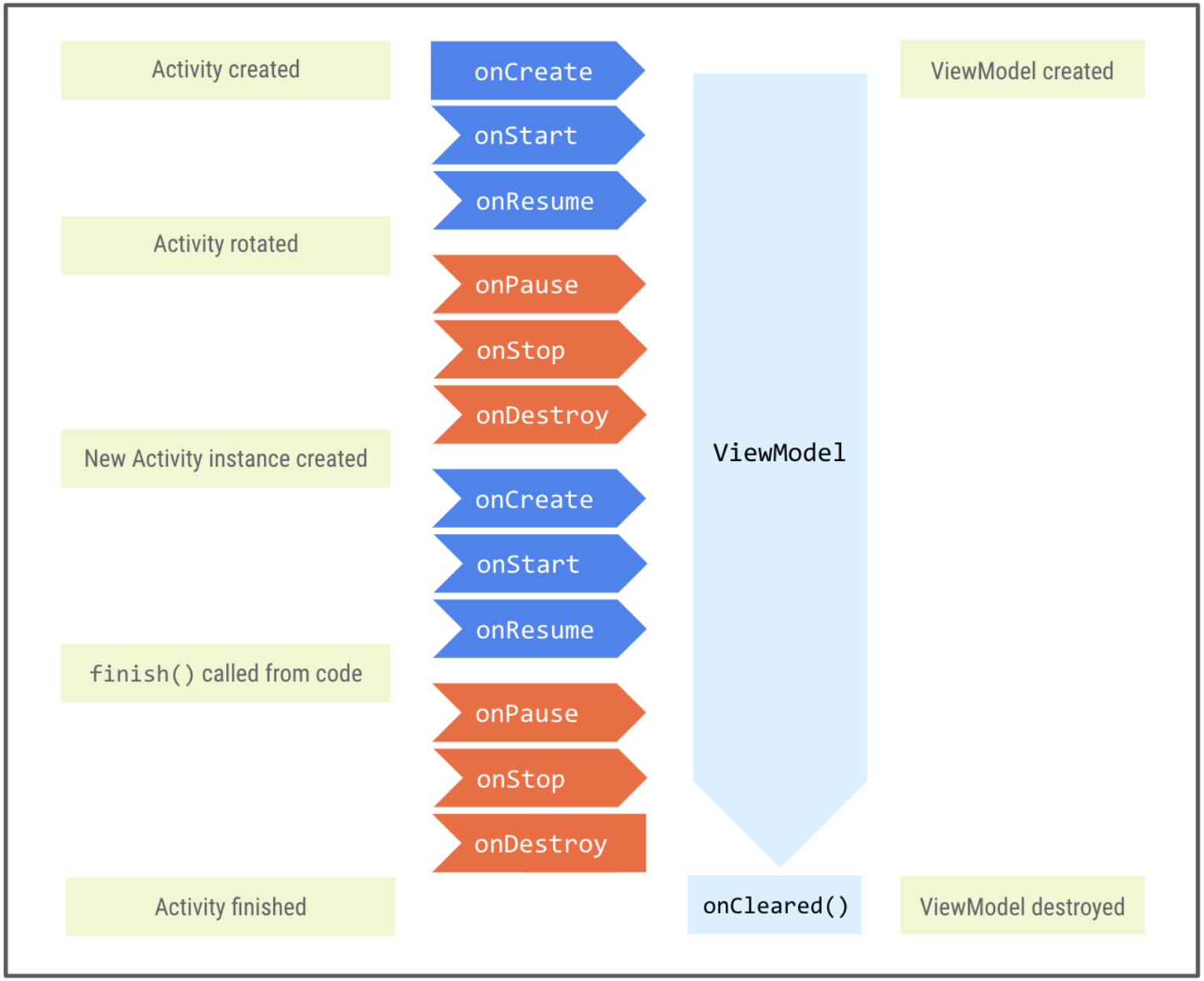
ViewModel

```
class WordViewModel : ViewModel() {
    private var _wordsList: MutableList<String> = mutableListOf()
    val wordList: List<String>
        get() = _wordsList

    init {
        reinitialize()
    }

    fun addWord(word: String){
        _wordsList.add(word)
        _wordsList.sort()
    }

    fun reinitialize(){
        _wordsList.clear()
        _wordsList.addAll(DataProvider.words)
        _wordsList.sort()
    }

    fun clear(){
        _wordsList.clear()
    }
}
```



Uniwersytet Wrocławski

ViewModel



Uniwersytet Wrocławski

RecyclerView

```
class WordViewHolder(private val binding: RvItemBinding) : RecyclerView.ViewHolder(binding.root) {
    fun bind(item: String) {
        binding.wordTextView.text = item
    }
}
```

Uniwersytet Wrocławski

RecyclerView

```
class WordViewHolder(private val binding: RvItemBinding) : RecyclerView.ViewHolder(binding.root) {
    fun bind(item: String) {
        binding.wordTextView.text = item
    }
}
```

```
class WordComparator : DiffUtil.ItemCallback<String>() {
    override fun areItemsTheSame(oldItem: String, newItem: String): Boolean {
        return oldItem === newItem
    }

    override fun areContentsTheSame(oldItem: String, newItem: String): Boolean {
        return oldItem == newItem
    }
}
```

Uniwersytet Wrocławski

RecyclerView

```
class WordViewHolder(private val binding: RvItemBinding) : RecyclerView.ViewHolder(binding.root) {
    fun bind(item: String) {
        binding.wordTextView.text = item
    }
}
```

```
class WordComparator : DiffUtil.ItemCallback<String>() {
    override fun areItemsTheSame(oldItem: String, newItem: String): Boolean {
        return oldItem === newItem
    }

    override fun areContentsTheSame(oldItem: String, newItem: String): Boolean {
        return oldItem == newItem
    }
}
```

```
class WordAdapter(wordComparator: WordComparator) : ListAdapter<String, WordViewHolder>(wordComparator) {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): WordViewHolder {
        return WordViewHolder(
            RvItemBinding.inflate(
                LayoutInflater.from(parent.context), parent, false
            )
        )
    }

    override fun onBindViewHolder(holder: WordViewHolder, position: Int) {
        val item = getItem(position)
        holder.bind(item)
    }
}
```

RecyclerView

```
private fun onAddWord() {
    val word = binding.wordEditText.text.toString()
    viewModel.addWord(word)
    wordAdapter.notifyDataSetChanged()
}

private fun onResetWords(){
    viewModel.reinitialize()
    wordAdapter.notifyDataSetChanged()
}

private fun onClearWords(){
    viewModel.clear()
    wordAdapter.notifyDataSetChanged()
}
```

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View {
    binding = FragmentListBinding.inflate(inflater)

    wordAdapter.submitList(viewModel.wordList)

    binding.rvList.apply{
        adapter = wordAdapter
        layoutManager = LinearLayoutManager(requireContext())
    }

    binding.addButton.setOnClickListener { onAddWord() }
    binding.resetButton.setOnClickListener { onResetWords() }
    binding.clearButton.setOnClickListener { onClearWords() }

    return binding.root
}
```

Compose

```
class WordViewModel : ViewModel() {
    private var _wordsList = mutableStateListOf<String>()
    val wordList: List<String>
        get() = _wordsList

    init {
        reinitialize()
    }

    fun addWord(word: String){
        _wordsList.add(word)
        _wordsList.sort()
    }

    fun reinitialize(){
        _wordsList.clear()
        _wordsList.addAll(DataProvider.words)
        _wordsList.sort()
    }

    fun clear(){
        _wordsList.clear()
    }
}
```

Compose

```
class WordViewModel : ViewModel() {
    private var _wordsList = mutableStateListOf<String>()
    val wordList: List<String>
        get() = _wordsList

    init {
        reinitialize()
    }

    fun addWord(word: String){
        _wordsList.add(word)
        _wordsList.sort()
    }

    fun reinitialize(){
        _wordsList.clear()
        _wordsList.addAll(DataProvider.words)
        _wordsList.sort()
    }

    fun clear(){
        _wordsList.clear()
    }
}
```

```
LazyColumn(modifier = Modifier.weight(1f)){
    items(viewModel.wordList.size){
        Text(
            text = viewModel.wordList[it],
            fontSize = 32.sp,
            textAlign = TextAlign.Center,
            modifier = Modifier
                .fillMaxWidth()
                .padding(2.dp)
        )
    }
}
```

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ListScreen(){

    var word by remember { mutableStateOf("") }
    val viewModel: WordViewModel = viewModel()
```

Compose

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ListScreen(){

    var word by remember { mutableStateOf("") }
    val viewModel: WordViewModel = viewModel()
```

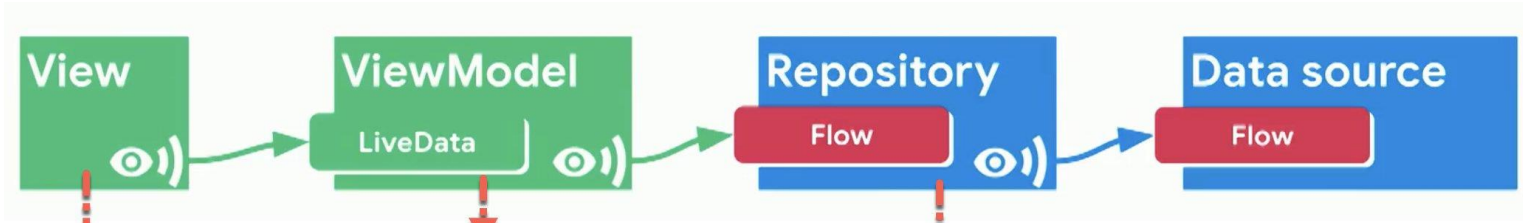
```
Button(
    modifier = Modifier
        .weight(1f)
        .padding(start = 2.dp, end = 4.dp),
    shape = RoundedCornerShape(8.dp),
    onClick = {
        if (word.isNotEmpty()) {
            viewModel.addWord(word)
        }
    }
) {
    Text(text = "ADD")
}
```

```
Button(
    modifier = Modifier
        .fillMaxWidth()
        .padding(start = 2.dp, end = 4.dp),
    shape = RoundedCornerShape(8.dp),
    onClick = { viewModel.clear() }
) {
    Text(text = "CLEAR")
}

Button(
    modifier = Modifier
        .fillMaxWidth()
        .padding(start = 2.dp, end = 4.dp),
    shape = RoundedCornerShape(8.dp),
    onClick = { viewModel.reinitialize() }
) {
    Text(text = "RESET")
}
```

LiveData

UWAGA

W aplikacjach opartych na języku **Kotlin** częściej wykorzystuje się `Flow`, `StateFlow`, lub `SharedFlow` (które poznamy na kolejnych zajęciach). Również można tworzyć aplikacje oparte na `LiveData` lecz jest to rzadziej spotykane.




```kotlin
class MyViewModel(myRepository: MyRepository) : ViewModel() {
    val data : LiveData<String> = myRepository.data
        .onStart { "Loading" }
        .map { "data converted for the view" }
        .asLiveData()
}
```

```kotlin
class MyFragment() : Fragment() {

    private val myViewModel : MyViewModel by viewModels<MyViewModel>()

    private val dataObserver = Observer<String> { it: String!
        // "Loading"
        // "data converted for the view"
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        observe(myViewModel.data, dataObserver)
    }
}

// Presentation layer extension
private fun <T> LifecycleOwner.observe(liveData: LiveData<T>, observer: Observer<T>) {
    liveData.observe( owner: this, observer)
}
```

```kotlin
class MyRepository() {
    val data: Flow<String> = flow { this: FlowCollector<String>
        emit( value: "networkDataModel")
    }
}
```

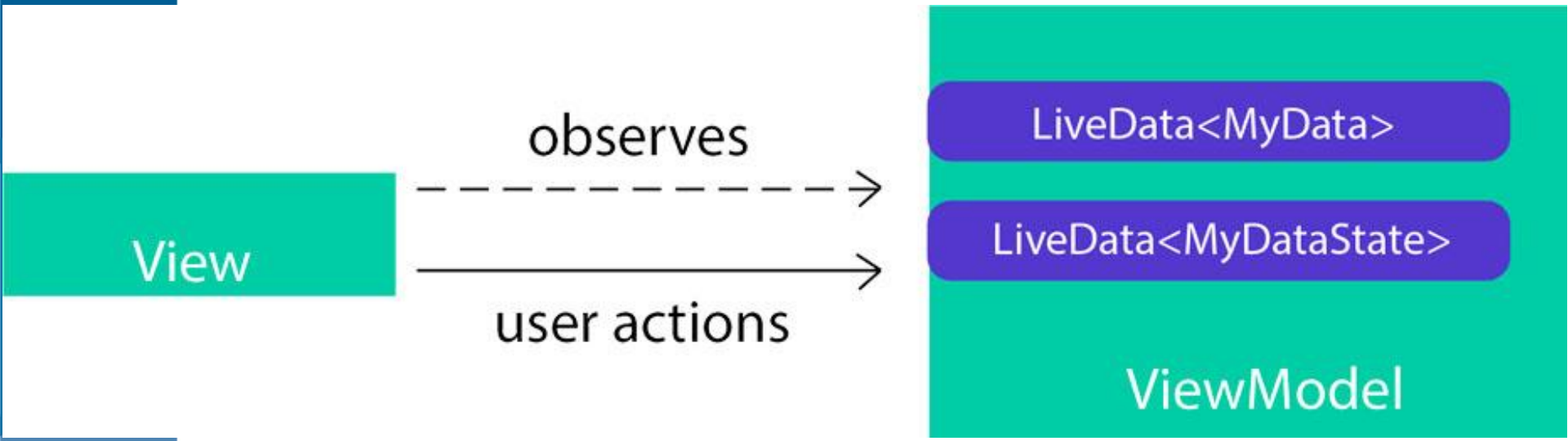
Uniwersytet Wrocławski

LiveData

UWAGA

W aplikacjach opartych na języku **Kotlin** częściej wykorzystuje się `Flow`, `StateFlow`, lub `SharedFlow` (które poznamy na kolejnych zajęciach). Również można tworzyć aplikacje oparte na `LiveData` lecz jest to rzadziej spotykane.

`LiveData` jest częścią bibliotek Androida i jest obiektem, który przechowuje dane **obserwowane** przez komponenty aplikacji, takie jak aktywności, fragmenty czy `ViewModel`. Jest zaprojektowany tak, aby dostarczać reaktywne i cykliczne powiadomienia o zmianach danych.



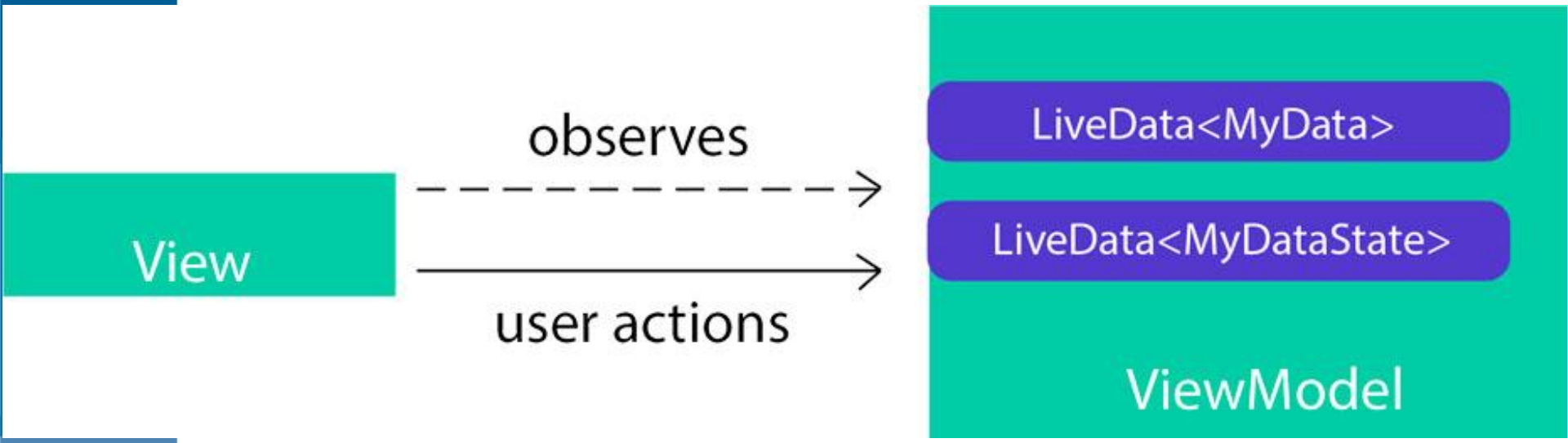
Uniwersytet Wrocławski

LiveData

UWAGA

W aplikacjach opartych na języku **Kotlin** częściej wykorzystuje się `Flow`, `StateFlow`, lub `SharedFlow` (które poznamy na kolejnych zajęciach). Również można tworzyć aplikacje oparte na `LiveData` lecz jest to rzadziej spotykane.

`LiveData` jest częścią bibliotek Androida i jest obiektem, który przechowuje dane **obserwowane** przez komponenty aplikacji, takie jak aktywności, fragmenty czy `ViewModel`. Jest zaprojektowany tak, aby dostarczać reaktywne i cykliczne powiadomienia o zmianach danych.



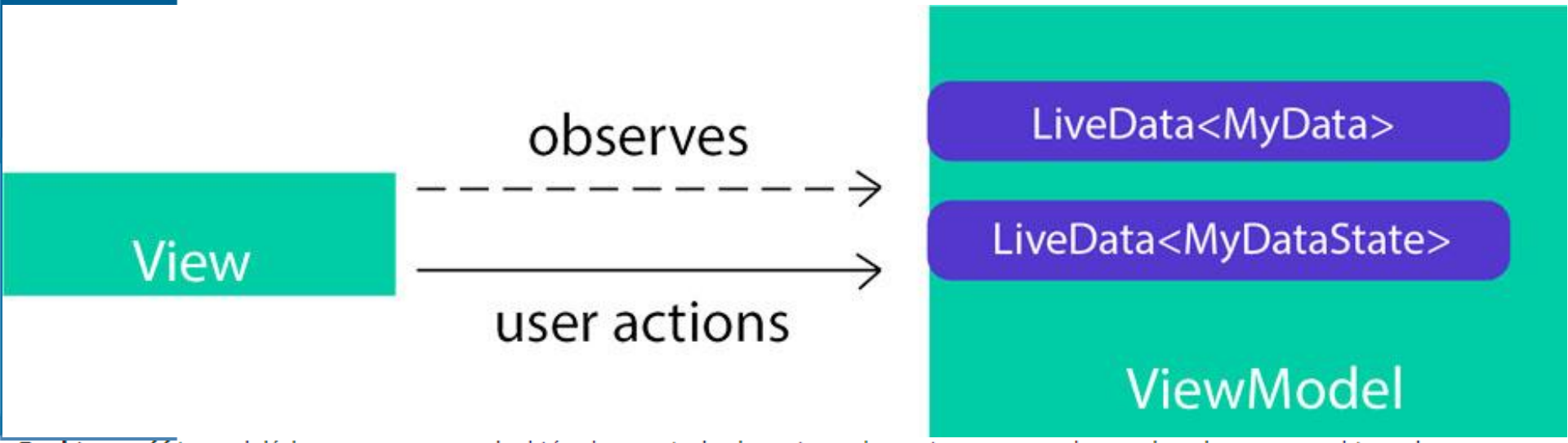
Reaktywność to podejście w programowaniu, które koncentruje się na tym, aby system reagował na zmiany i propagował te zmiany w sposób automatyczny. W kontekście aplikacji, reaktywność odnosi się do zdolności systemu do dynamicznego reagowania na zmiany danych i propagowania tych zmian do odpowiednich komponentów.

LiveData

UWAGA

W aplikacjach opartych na języku **Kotlin** częściej wykorzystuje się `Flow`, `StateFlow`, lub `SharedFlow` (które poznamy na kolejnych zajęciach). Również można tworzyć aplikacje oparte na `LiveData` lecz jest to rzadziej spotykane.

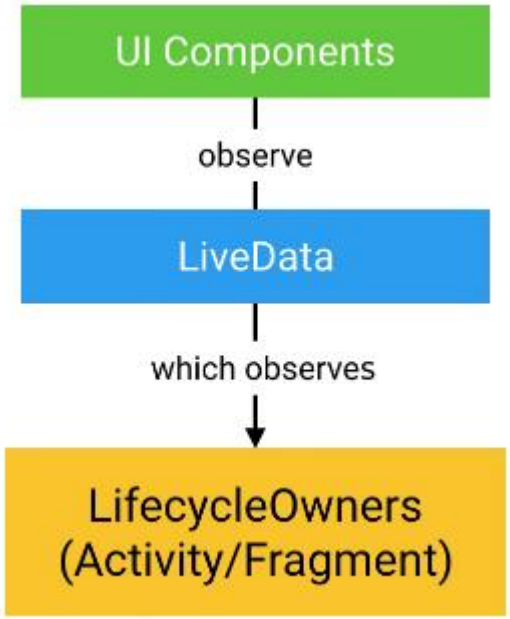
`LiveData` jest częścią bibliotek Androida i jest obiektem, który przechowuje dane **obserwowane** przez komponenty aplikacji, takie jak aktywności, fragmenty czy `ViewModel`. Jest zaprojektowany tak, aby dostarczać reaktywne i cykliczne powiadomienia o zmianach danych.



Reaktywność to podejście w programowaniu, które koncentruje się na tym, aby system reagował na zmiany i propagował te zmiany w sposób automatyczny. W kontekście aplikacji, reaktywność odnosi się do zdolności systemu do dynamicznego reagowania na zmiany danych i propagowania tych zmian do odpowiednich komponentów.

Opiera się na obsłudze zdarzeń, które są generowane w systemie w wyniku **zmian w danych**. Zdarzenia te mogą być przekształcane i łączone za pomocą różnych operacji, tworząc **strumienie danych**. Strumienie są sekwencją wartości, które mogą być emitowane i subskrybowane (obserwowane) przez komponenty. Automatycznie propaguje zmiany w danych do komponentów, które subskrybują te dane. Oznacza to, że komponenty **nie muszą** ręcznie monitorować i aktualizować danych, ponieważ system sam zarządza tym procesem.

LiveData



1.Zapewnia, że stan interfejsu użytkownika są zgodny z aktualnym stanem danych. LiveData powiadamia obiekty Observer, gdy zmienia się stan cyklu życia. Zamiast aktualizować interfejs użytkownika za każdym razem, gdy zmieniają się dane aplikacji, obserwator może aktualizować interfejs użytkownika za każdym razem, gdy następuje zmiana.

2.Brak wycieków pamięci. Obserwatorzy są związani z obiektami cyklu życia.

3.Brak błędów z powodu zatrzymanych aktywności. Jeśli cykl życia obserwatora jest nieaktywny, nie otrzymuje on żadnych zdarzeń LiveData.

4.Komponenty interfejsu użytkownika jedynie obserwują istotne dane i nie przerywają ani nie wznawiają obserwacji. LiveData automatycznie zarządza wszystkim tym, ponieważ jest „świadoma” istotnych zmian stanu cyklu życia podczas obserwacji.

5.Zawsze aktualne dane. Jeśli cykl życia staje się nieaktywny, otrzymuje najnowsze dane po ponownym staniu się aktywnym. Na przykład aktywność, która była w tle, otrzymuje najnowsze dane zaraz po powrocie do pierwszego planu.

Uniwersytet Wrocławski

LiveData

```
class CounterViewModel : ViewModel() {
    private var _counter = MutableLiveData(0)
    val counter: LiveData<Int>
        get() = _counter

    fun increase(){
        _counter.value = _counter.value?.inc()
    }

    fun decrease(){
        _counter.value = _counter.value?.dec()
    }

    fun clear(){ _counter.value = 0 }
}
```

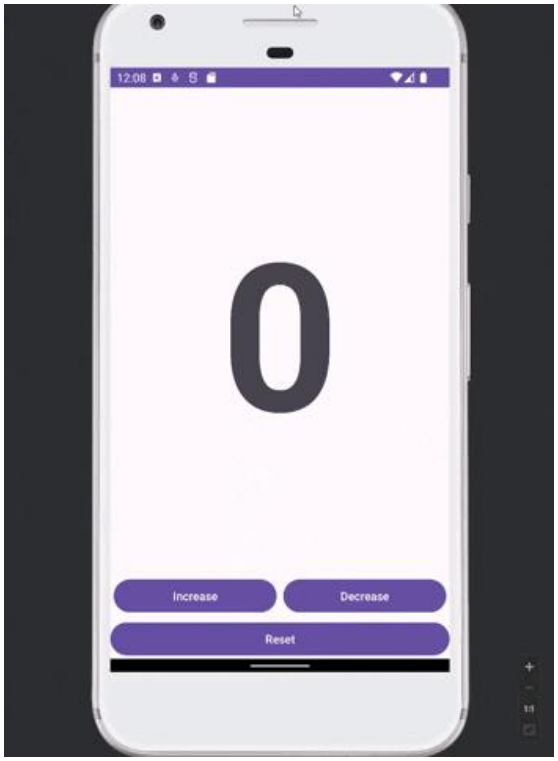
Uniwersytet Wrocławski

LiveData

```
private val viewModel: CounterViewModel by viewModels()
```

```
viewModel.counter.observe(viewLifecycleOwner) { newValue ->
    binding.showCount.text = newValue.toString()
}
```

```
binding.increaseButton.setOnClickListener { viewModel.increase() }
binding.decreaseButton.setOnClickListener { viewModel.decrease() }
binding.resetButtton.setOnClickListener { viewModel.clear() }
```




```
class CounterViewModel : ViewModel() {
    private var _counter = MutableLiveData(0)
    val counter: LiveData<Int>
        get() = _counter

    fun increase(){
        _counter.value = _counter.value?.inc()
    }

    fun decrease(){
        _counter.value = _counter.value?.dec()
    }

    fun clear(){ _counter.value = 0 }
}
```

LiveData

UWAGA

W aplikacjach opartych na **JetpackCompose** wykorzystuje się `State` (który widzieliśmy w poprzednim przykładzie), `StateFlow`, lub `ComposeFlow` (które poznamy na kolejnych zajęciach). Również można tworzyć aplikacje oparte na `LiveData` lecz jest to rzadziej spotykane. W praktyce posiadając `LiveData` w aplikacji konwertujemy go do `State` przez metodę `observeAsState`, z którą zapoznamy się w tym przykładzie.

```
@Composable
fun CounterScreen() {

    val viewModel: CounterViewModel = viewModel() // zapewnia dostęp do metod dostępowych i danych (tylko do
    val counterState = viewModel.counter.observeAsState() // przy każdej zmianie wartości counter,
                                                          // counterState otrzymuje aktualną wartość
```