Uniwersytet Wrocławski

Rafał Lewandków

PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 2

WYKŁAD 12

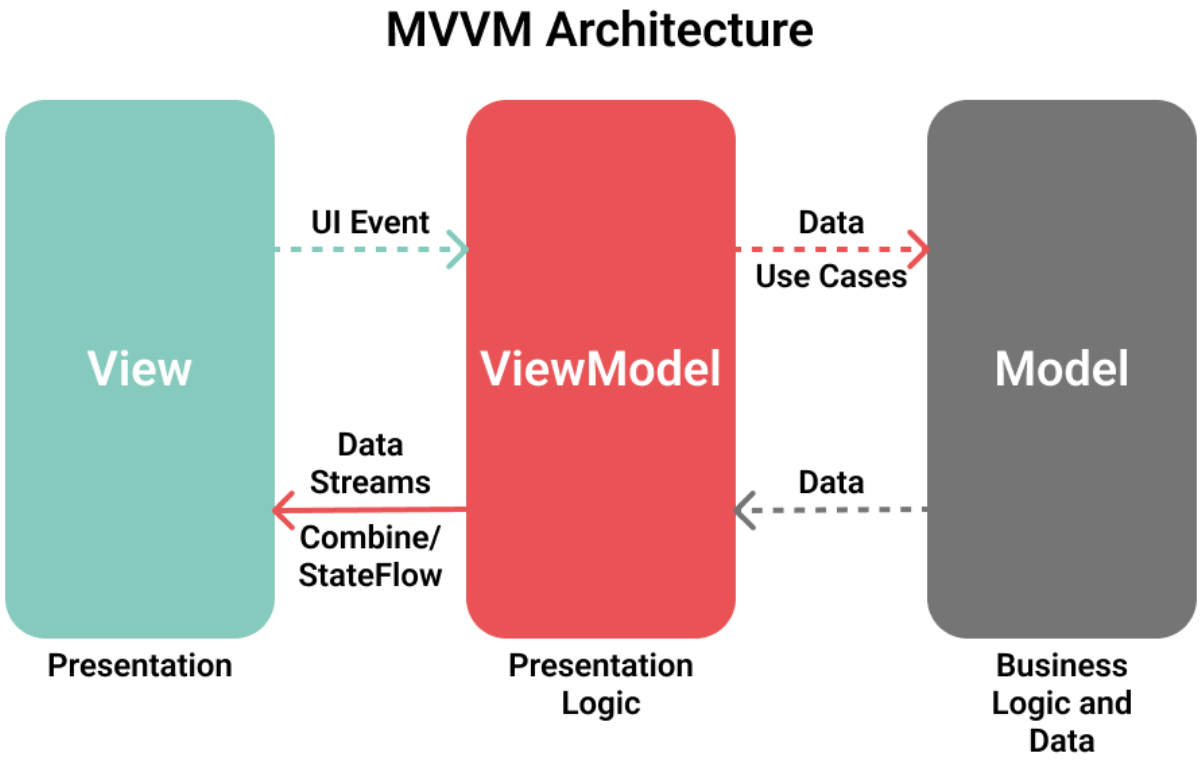
Czysta Architektura

- Warstwa Domeny
- Use Case

Warstwa Domeny

Do tej pory nasza architektura składała się z trzech głównych warstw: UI, ViewModel i Repository.

W przypadku łączenia danych z wielu repozytoriów lub wykonania złożonych reguł, warstwa Modelu zaczyna być zbyt mała.



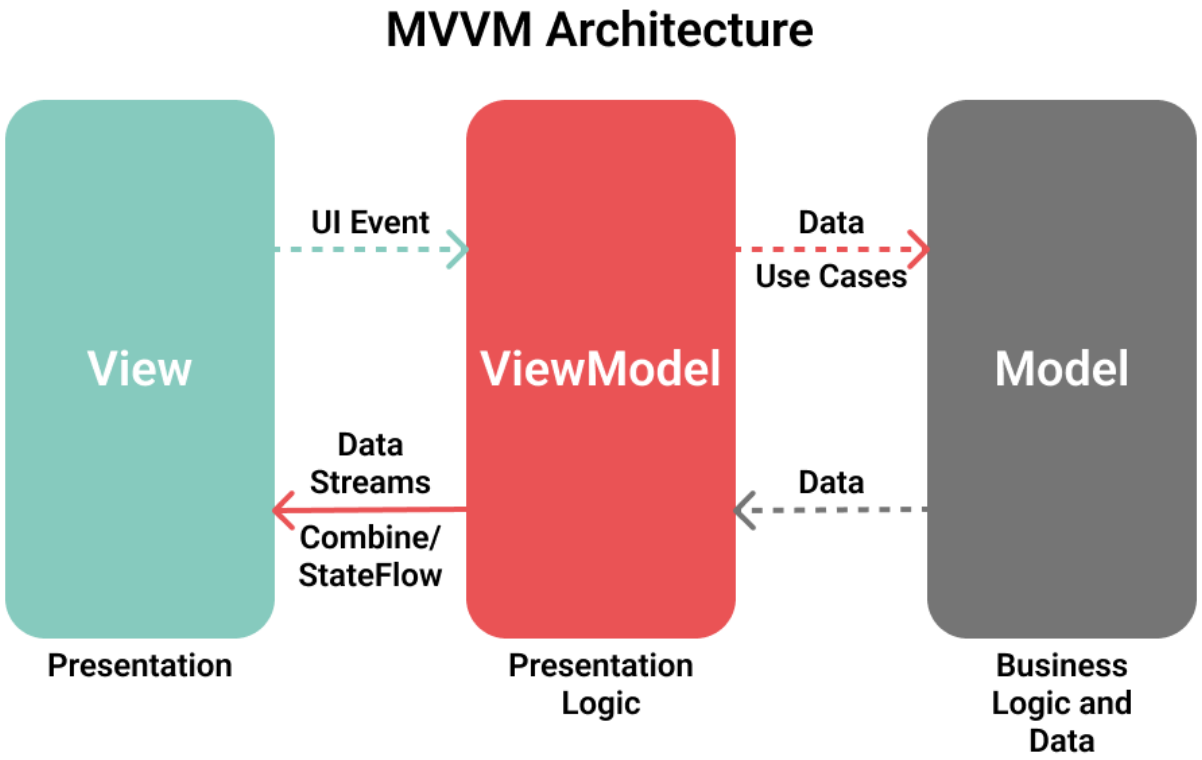
https://www.fastdev.com/blog/blog/building-maintainable-mobile-apps

Warstwa Domeny

Do tej pory nasza architektura składała się z trzech głównych warstw: UI, ViewModel i Repository.

W przypadku łączenia danych z wielu repozytoriów lub wykonania złożonych reguł, warstwa Modelu zaczyna być zbyt mała.

ViewModel musi pobrać dane użytkownika z UserRepository i jego listę zakupów z ShoppingRepository, a następnie połączyć je i odfiltrować. Taka logika *zaśmieca* ViewModel, którego głównym zadaniem powinno być **zarządzanie stanem UI**.



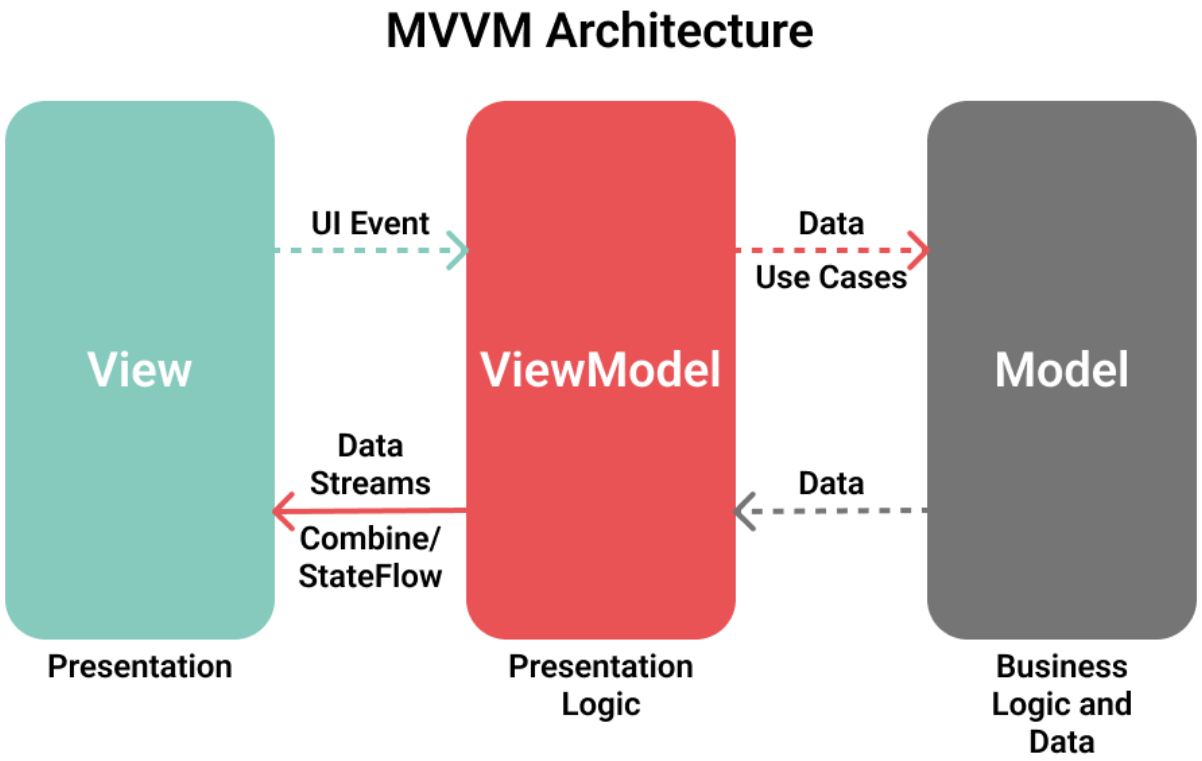
https://www.fastdev.com/blog/blog/building-maintainable-mobile-apps

Warstwa Domeny

Do tej pory nasza architektura składała się z trzech głównych warstw: UI, ViewModel i Repository.

W przypadku łączenia danych z wielu repozytoriów lub wykonania złożonych reguł, warstwa Modelu zaczyna być zbyt mała.

Repository powinno zarządzać **tylko źródłem danych**. Jeśli dodamy do niego logikę biznesową, (np. *jeśli użytkownik jest premium, zwróć 50 artykułów, w przeciwnym razie 10),* to już nie jest tylko dostęp do danych, ale **reguła biznesowa**.



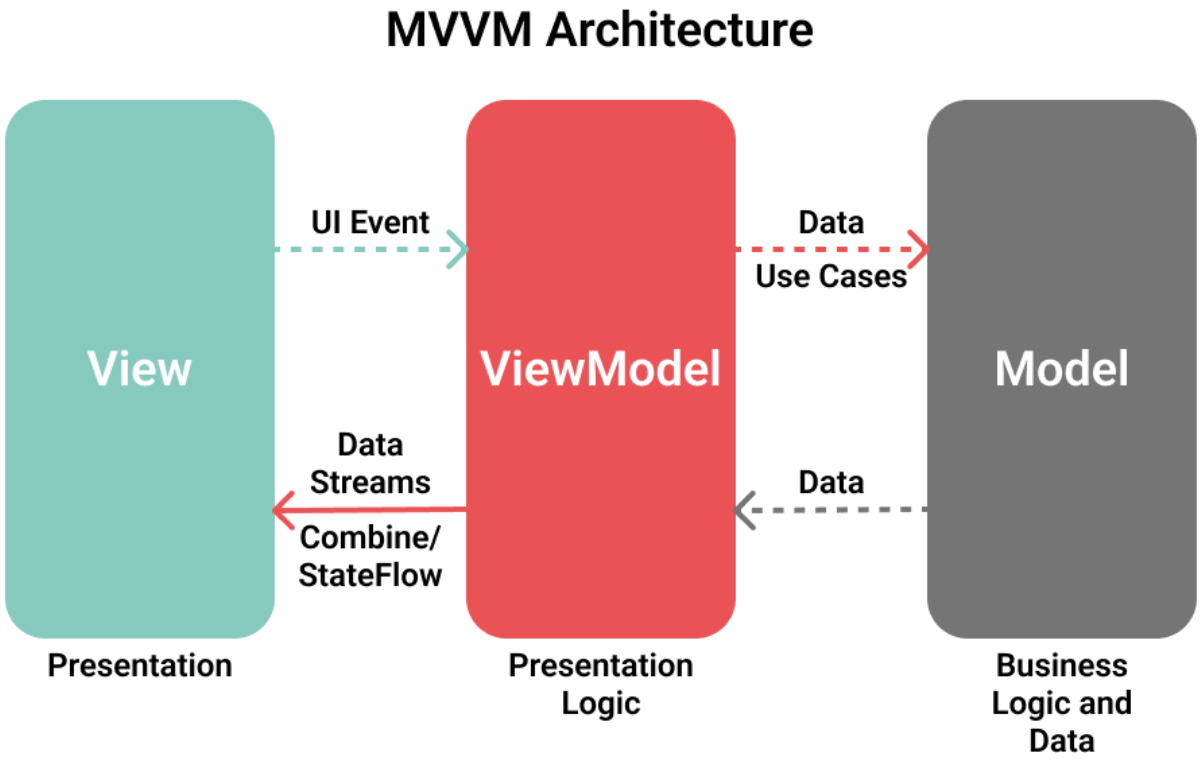
https://www.fastdev.com/blog/blog/building-maintainable-mobile-apps

Warstwa Domeny

Do tej pory nasza architektura składała się z trzech głównych warstw: UI, ViewModel i Repository.

W przypadku łączenia danych z wielu repozytoriów lub wykonania złożonych reguł, warstwa Modelu zaczyna być zbyt mała.

Jeśli ta sama, skomplikowana logika (np. *walidacja numeru NIP*) jest potrzebna w dwóch różnych ViewModelach, musi np. zostać powielona.



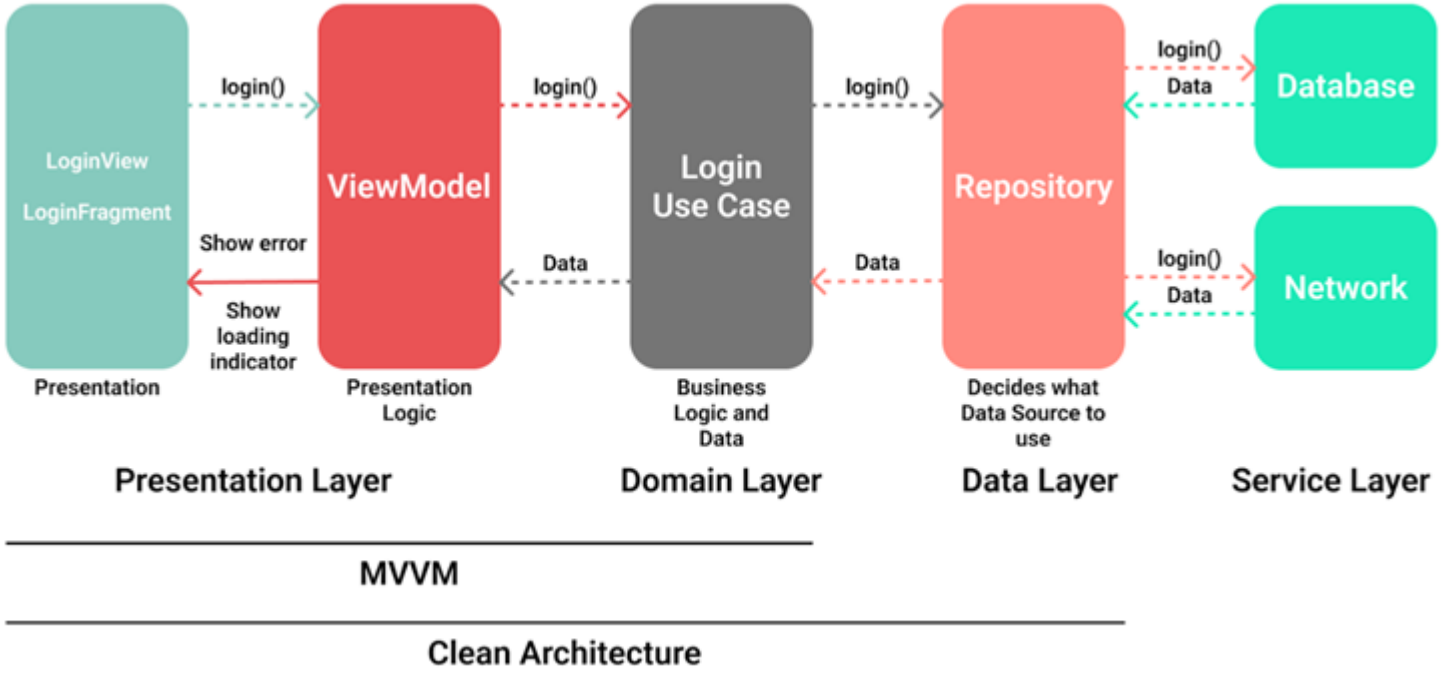
https://www.fastdev.com/blog/blog/building-maintainable-mobile-apps

Warstwa Domeny

Do tej pory nasza architektura składała się z trzech głównych warstw: UI, ViewModel i Repository.

W przypadku łączenia danych z wielu repozytoriów lub wykonania złożonych reguł, warstwa Modelu zaczyna być zbyt mała. Rozwiązaniem jest wprowadzenie nowej warstwy – **warstwy domeny.**

Jest to centralne miejsce na **logikę biznesową.** Narzędziem do jej implementacji jest wzorzec **UseCase**.



https://www.fastdev.com/blog/blog/building-maintainable-mobile-apps

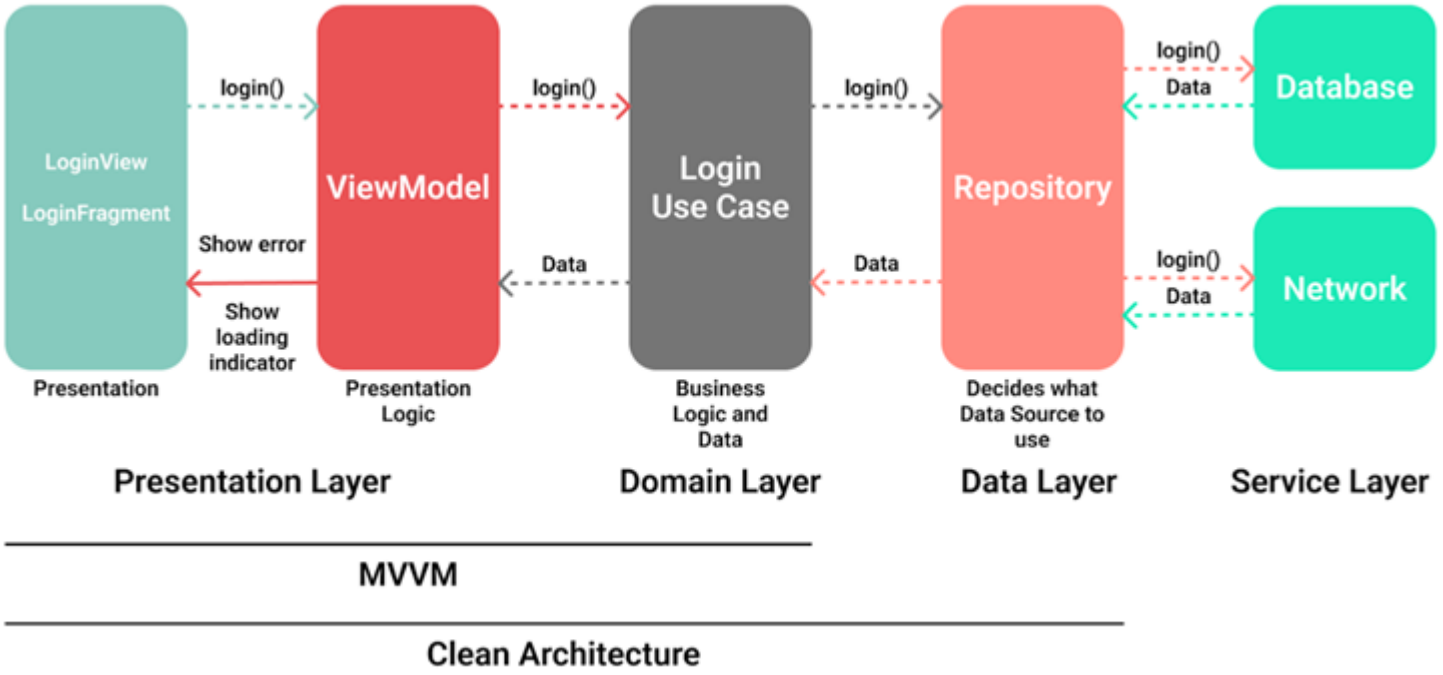
Warstwa Domeny

Do tej pory nasza architektura składała się z trzech głównych warstw: UI, ViewModel i Repository.

W przypadku łączenia danych z wielu repozytoriów lub wykonania złożonych reguł, warstwa Modelu zaczyna być zbyt mała. Rozwiązaniem jest wprowadzenie nowej warstwy – **warstwy domeny.**

Warstwa domeny - Zawiera najważniejsze **reguły biznesowe** (np. *jak obliczyć cenę, co walidować, jakie dane połączyć*).

UseCase - To prosta klasa, która reprezentuje **jedną, konkretną akcję biznesową** w aplikacji.



https://www.fastdev.com/blog/blog/building-maintainable-mobile-apps

Uniwersytet Wrocławski

Warstwa Domeny

Do tej pory nasza architektura składała się z trzech głównych warstw: UI, ViewModel i Repository.

W przypadku łączenia danych z wielu repozytoriów lub wykonania złożonych reguł, warstwa Modelu zaczyna być zbyt mała. Rozwiązaniem jest wprowadzenie nowej warstwy – **warstwy domeny.**

Warstwa domeny - Zawiera najważniejsze **reguły biznesowe** (np. *jak obliczyć cenę, co walidować, jakie dane połączyć*).

UseCase - To prosta klasa, która reprezentuje **jedną, konkretną akcję biznesową** w aplikacji.

- **ViewModel (Kierownik Budowy):** Zarządza całym ekranem (projektem).
- **Repository (Magazyn):** Dostarcza surowe materiały (dane).
- **Use Cases (Specjaliści):** Kierownik nie wykonuje sam całej pracy. Posiada wyspecjalizowanych fachowców:
  - **FormatujCenęUseCase**, *weź tę liczbę z magazynu i sformatuj ją jako walutę.*
  - **PobierzArtykułyDlaUżytkownikaPremiumUseCase**, *idź do magazynu i przynieś mi odpowiednie artykuły.*
  - **SprawdźPoprawnośćEmailaUseCase**, *powiedz mi, czy ten email jest prawidłowy.*

Uniwersytet Wrocławski

Warstwa Domeny

Do tej pory nasza architektura składała się z trzech głównych warstw: UI, ViewModel i Repository.

W przypadku łączenia danych z wielu repozytoriów lub wykonania złożonych reguł, warstwa Modelu zaczyna być zbyt mała. Rozwiązaniem jest wprowadzenie nowej warstwy – **warstwy domeny.**

Warstwa domeny - Zawiera najważniejsze **reguły biznesowe** (np. *jak obliczyć cenę, co walidować, jakie dane połączyć*).

UseCase - To prosta klasa, która reprezentuje **jedną, konkretną akcję biznesową** w aplikacji.

operator fun invoke to specjalna funkcja w Kotlinie, która pozwala na **wywoływanie instancji klasy tak, jakby była funkcją**.

```
class ValidatePasswordUseCase @Inject constructor() {
    operator fun invoke(password: String): Boolean {
        return password.length >= 8 && password.any { it.isDigit() }
    }
}
class GetFavoriteArticlesUseCase @Inject constructor(
    private val articlesRepository: ArticleRepository
) {
    operator fun invoke(): Flow<List<Article>> {
        return articlesRepository.getArticlesStream()
            .map { articles -> articles.filter { it.isFavorite } }
    }
}
```

Warstwa Domeny

Do tej pory nasza architektura składała się z trzech głównych warstw: UI, ViewModel i Repository.

W przypadku łączenia danych z wielu repozytoriów lub wykonania złożonych reguł, warstwa Modelu zaczyna być zbyt mała. Rozwiązaniem jest wprowadzenie nowej warstwy – **warstwy domeny.**

Warstwa domeny - Zawiera najważniejsze **reguły biznesowe** (np. *jak obliczyć cenę, co walidować, jakie dane połączyć*).

UseCase - To prosta klasa, która reprezentuje **jedną, konkretną akcję biznesową** w aplikacji.

operator fun invoke to specjalna funkcja w Kotlinie, która pozwala na **wywoływanie instancji klasy tak, jakby była funkcją**.

```
class ValidatePasswordUseCase @Inject constructor() {
    operator fun invoke(password: String): Boolean {
        return password.length >= 8 && password.any { it.isDigit() }
    }
}
class GetFavoriteArticlesUseCase @Inject constructor(
    private val articlesRepository: ArticleRepository
) {
    operator fun invoke(): Flow<List<Article>> {
        return articlesRepository.getArticlesStream()
            .map { articles -> articles.filter { it.isFavorite } }
    }
}
```

Bez invoke

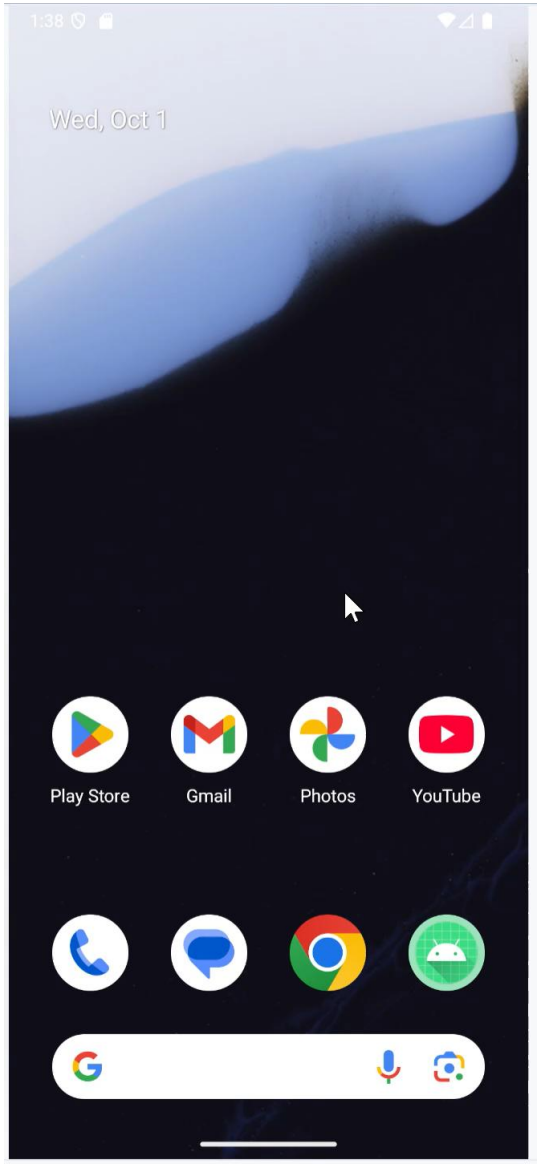
```
val useCase = ValidatePasswordUseCase()
val isPasswordValid = useCase.validate(password)
```

Z invoke

```
val useCase = ValidatePasswordUseCase()
val isPasswordValid = useCase(password)
```

Uniwersytet Wrocławski

Czysta Architektura



Czysta Architektura

```
class ProductRepository @Inject constructor() {
    // Symuluje cenę bazową produktu, np. pobraną z bazy danych lub API
    1 Usage
    fun getBasePriceFlow(): Flow<Double> = flowOf( value = 100.0)
}
```

prosty konstruktor (builder), który tworzy **zimny Flow** z ustalonej, z góry **znanej liczby argumentów**.

Cecha	flowOf(...)	flow { ... }
Przeznaczenie	Proste, **synchroniczne** emisje ze znanych danych.	Złożone, dynamiczne lub **asynchroniczne** emisje.
Możliwość suspend	**NIE**, wewnątrz nie można wywoływać delay() ani innych funkcji suspend.	**TAK**, blok kodu jest zawieszalny.

Uniwersytet Wrocławski

Czysta Architektura

Oblicza cenę produktu po dodaniu stałej stawki podatku VAT (23%). Jest to **logika biznesowa**.

```
class CalculateVatPriceUseCase @Inject constructor() {
    1 Usage
    private val vatRate = 0.23
    operator fun invoke(basePrice: Double): Double = basePrice * (1 + vatRate)
}
9 Usages
class ValidateCouponUseCase @Inject constructor() {
    operator fun invoke(couponCode: String): Double {
        return when (couponCode.uppercase()) {
            "PROMO10" -> 0.10 // 10% zniżki
            "SUMMER20" -> 0.20 // 20% zniżki
            else -> 0.0 // Brak zniżki
        }
    }
}
9 Usages
class ApplyDiscountUseCase @Inject constructor() {
    operator fun invoke(price: Double, discountRate: Double): Double {
        return price * (1 - discountRate)
    }
}
9 Usages
class FormatCurrencyUseCase @Inject constructor() {
    1 Usage
    private val currencyFormatter =
        NumberFormat.getCurrencyInstance(
            inLocale = Locale.forLanguageTag( languageTag = "pl-PL")
        )
    operator fun invoke(value: Double): String {
        return currencyFormatter.format( number = value)
    }
}
```

Uniwersytet Wrocławski

Czysta Architektura

Oblicza cenę produktu po dodaniu stałej stawki podatku VAT (23%). Jest to **logika biznesowa**.

Sprawdza, czy podany kod rabatowy (couponCode) jest prawidłowy i zwraca odpowiednią stopę zniżki (np. 0.10 dla 10%). Hermetyzuje **logikę walidacji kuponów**.

```
class CalculateVatPriceUseCase @Inject constructor() {
    1 Usage
    private val vatRate = 0.23
    operator fun invoke(basePrice: Double): Double = basePrice * (1 + vatRate)
}
9 Usages
class ValidateCouponUseCase @Inject constructor() {
    operator fun invoke(couponCode: String): Double {
        return when (couponCode.uppercase()) {
            "PROMO10" -> 0.10 // 10% zniżki
            "SUMMER20" -> 0.20 // 20% zniżki
            else -> 0.0 // Brak zniżki
        }
    }
}
9 Usages
class ApplyDiscountUseCase @Inject constructor() {
    operator fun invoke(price: Double, discountRate: Double): Double {
        return price * (1 - discountRate)
    }
}
9 Usages
class FormatCurrencyUseCase @Inject constructor() {
    1 Usage
    private val currencyFormatter =
        NumberFormat.getCurrencyInstance(
            inLocale = Locale.forLanguageTag( languageTag = "pl-PL")
        )
    operator fun invoke(value: Double): String {
        return currencyFormatter.format( number = value)
    }
}
```

Uniwersytet Wrocławski

Czysta Architektura

Oblicza cenę produktu po dodaniu stałej stawki podatku VAT (23%). Jest to **logika biznesowa**.

Sprawdza, czy podany kod rabatowy (couponCode) jest prawidłowy i zwraca odpowiednią stopę zniżki (np. 0.10 dla 10%). Hermetyzuje **logikę walidacji kuponów**.

Oblicza finalną cenę produktu po zastosowaniu zniżki. Przyjmuje cenę oraz stopę zniżki (zwróconą np. przez ValidateCouponUseCase) i zwraca ostateczną kwotę.

```
class CalculateVatPriceUseCase @Inject constructor() {
    1 Usage
    private val vatRate = 0.23
    operator fun invoke(basePrice: Double): Double = basePrice * (1 + vatRate)
}
9 Usages
class ValidateCouponUseCase @Inject constructor() {
    operator fun invoke(couponCode: String): Double {
        return when (couponCode.uppercase()) {
            "PROMO10" -> 0.10 // 10% zniżki
            "SUMMER20" -> 0.20 // 20% zniżki
            else -> 0.0 // Brak zniżki
        }
    }
}
9 Usages
class ApplyDiscountUseCase @Inject constructor() {
    operator fun invoke(price: Double, discountRate: Double): Double {
        return price * (1 - discountRate)
    }
}
9 Usages
class FormatCurrencyUseCase @Inject constructor() {
    1 Usage
    private val currencyFormatter =
        NumberFormat.getCurrencyInstance(
            inLocale = Locale.forLanguageTag( languageTag = "pl-PL")
        )
    operator fun invoke(value: Double): String {
        return currencyFormatter.format( number = value)
    }
}
```

Uniwersytet Wrocławski

Czysta Architektura

Oblicza cenę produktu po dodaniu stałej stawki podatku VAT (23%). Jest to **logika biznesowa**.

Sprawdza, czy podany kod rabatowy (couponCode) jest prawidłowy i zwraca odpowiednią stopę zniżki (np. 0.10 dla 10%). Hermetyzuje **logikę walidacji kuponów**.

Oblicza finalną cenę produktu po zastosowaniu zniżki. Przyjmuje cenę oraz stopę zniżki (zwróconą np. przez ValidateCouponUseCase) i zwraca ostateczną kwotę.

Formatuje wartość liczbową na poprawnie wyglądający ciąg znaków w polskiej walucie. Jest to Use Case odpowiedzialny za **logikę prezentacji danych**, zapewniając spójne formatowanie w całej aplikacji.

```
class CalculateVatPriceUseCase @Inject constructor() {
    1 Usage
    private val vatRate = 0.23
    operator fun invoke(basePrice: Double): Double = basePrice * (1 + vatRate)
}
9 Usages
class ValidateCouponUseCase @Inject constructor() {
    operator fun invoke(couponCode: String): Double {
        return when (couponCode.uppercase()) {
            "PROMO10" -> 0.10 // 10% zniżki
            "SUMMER20" -> 0.20 // 20% zniżki
            else -> 0.0 // Brak zniżki
        }
    }
}
9 Usages
class ApplyDiscountUseCase @Inject constructor() {
    operator fun invoke(price: Double, discountRate: Double): Double {
        return price * (1 - discountRate)
    }
}
9 Usages
class FormatCurrencyUseCase @Inject constructor() {
    1 Usage
    private val currencyFormatter =
        NumberFormat.getCurrencyInstance(
            inLocale = Locale.forLanguageTag( languageTag = "pl-PL")
        )
    operator fun invoke(value: Double): String {
        return currencyFormatter.format( number = value)
    }
}
```

Uniwersytet Wrocławski

Czysta Architektura

```kotlin
data class CartUiState(
    val basePrice: String = "PLN 0,00",
    val priceWithVat: String = "PLN 0,00",
    val finalPrice: String = "PLN 0,00",
    val couponCode: String = ""
)
```

Dzięki zastosowaniu biblioteki Hilt, ViewModel po prostu żąda potrzebne mu obiekty.

```kotlin
@HiltViewModel
class CartViewModel @Inject constructor(
    productRepository: ProductRepository,
    private val calculateVatPriceUseCase: CalculateVatPriceUseCase,
    private val validateCouponUseCase: ValidateCouponUseCase,
    private val applyDiscountUseCase: ApplyDiscountUseCase,
    private val formatCurrencyUseCase: FormatCurrencyUseCase
) : ViewModel() {
    2 Usages
    private val couponCodeFlow = MutableStateFlow( value = "")
    1 Usage
    fun onCouponCodeChanged(newCode: String) {
        couponCodeFlow.value = newCode
    }
    1 Usage
    val uiState: StateFlow<CartUiState> = combine(
        flow = productRepository.getBasePriceFlow(),
        flow2 = couponCodeFlow
    ) { basePrice, coupon ->
        // ViewModel działa jak koordynator, wywołując specjalistów (Use Cases)
        val priceWithVat = calculateVatPriceUseCase(basePrice)
        val discountRate = validateCouponUseCase( couponCode = coupon)
        val finalPrice = applyDiscountUseCase( price = priceWithVat, discountRate)

        // Przekształcamy surowe dane na sformatowane Stringi gotowe dla UI
        CartUiState(
            basePrice = formatCurrencyUseCase( value = basePrice),
            priceWithVat = formatCurrencyUseCase( value = priceWithVat),
            finalPrice = formatCurrencyUseCase( value = finalPrice),
            couponCode = coupon
        )
    }.stateIn(...)
}
```

Uniwersytet Wrocławski

Czysta Architektura

```kotlin
data class CartUiState(
    val basePrice: String = "PLN 0,00",
    val priceWithVat: String = "PLN 0,00",
    val finalPrice: String = "PLN 0,00",
    val couponCode: String = ""
)
```

Dzięki zastosowaniu biblioteki Hilt, ViewModel po prostu żąda potrzebne mu obiekty.

Wywoływany jak funkcja

```kotlin
@HiltViewModel
class CartViewModel @Inject constructor(
    productRepository: ProductRepository,
    private val calculateVatPriceUseCase: CalculateVatPriceUseCase,
    private val validateCouponUseCase: ValidateCouponUseCase,
    private val applyDiscountUseCase: ApplyDiscountUseCase,
    private val formatCurrencyUseCase: FormatCurrencyUseCase
) : ViewModel() {
    2 Usages
    private val couponCodeFlow = MutableStateFlow( value = "")
    1 Usage
    fun onCouponCodeChanged(newCode: String) {
        couponCodeFlow.value = newCode
    }
    1 Usage
    val uiState: StateFlow<CartUiState> = combine(
        flow = productRepository.getBasePriceFlow(),
        flow2 = couponCodeFlow
    ) { basePrice, coupon ->
        // ViewModel działa jak koordynator, wywołując specjalistów (Use Cases)
        val priceWithVat = calculateVatPriceUseCase(basePrice)
        val discountRate = validateCouponUseCase( couponCode = coupon)
        val finalPrice = applyDiscountUseCase( price = priceWithVat, discountRate)

        // Przekształcamy surowe dane na sformatowane Stringi gotowe dla UI
        CartUiState(
            basePrice = formatCurrencyUseCase( value = basePrice),
            priceWithVat = formatCurrencyUseCase( value = priceWithVat),
            finalPrice = formatCurrencyUseCase( value = finalPrice),
            couponCode = coupon
        )
    }.stateIn(...)
}
```

Uniwersytet Wrocławski

Czysta Architektura

```kotlin
data class CartUiState(
    val basePrice: String = "PLN 0,00",
    val priceWithVat: String = "PLN 0,00",
    val finalPrice: String = "PLN 0,00",
    val couponCode: String = ""
)
```

Dzięki zastosowaniu biblioteki Hilt, ViewModel po prostu żąda potrzebne mu obiekty.

Wywoływany jak funkcja

Rola ViewModel sprowadza się do **koordynacji** – wołania odpowiednich Use Case'ów i **aktualizowania stanu UI**.

```kotlin
@HiltViewModel
class CartViewModel @Inject constructor(
    productRepository: ProductRepository,
    private val calculateVatPriceUseCase: CalculateVatPriceUseCase,
    private val validateCouponUseCase: ValidateCouponUseCase,
    private val applyDiscountUseCase: ApplyDiscountUseCase,
    private val formatCurrencyUseCase: FormatCurrencyUseCase
) : ViewModel() {
    2 Usages
    private val couponCodeFlow = MutableStateFlow( value = "")
    1 Usage
    fun onCouponCodeChanged(newCode: String) {
        couponCodeFlow.value = newCode
    }
    1 Usage
    val uiState: StateFlow<CartUiState> = combine(
        flow = productRepository.getBasePriceFlow(),
        flow2 = couponCodeFlow
    ) { basePrice, coupon ->
        // ViewModel działa jak koordynator, wywołując specjalistów (Use Cases)
        val priceWithVat = calculateVatPriceUseCase(basePrice)
        val discountRate = validateCouponUseCase( couponCode = coupon)
        val finalPrice = applyDiscountUseCase( price = priceWithVat, discountRate)

        // Przekształcamy surowe dane na sformatowane Stringi gotowe dla UI
        CartUiState(
            basePrice = formatCurrencyUseCase( value = basePrice),
            priceWithVat = formatCurrencyUseCase( value = priceWithVat),
            finalPrice = formatCurrencyUseCase( value = finalPrice),
            couponCode = coupon
        )
    }.stateIn(...)
}
```