Uniwersytet Wrocławski

Rafał Lewandków

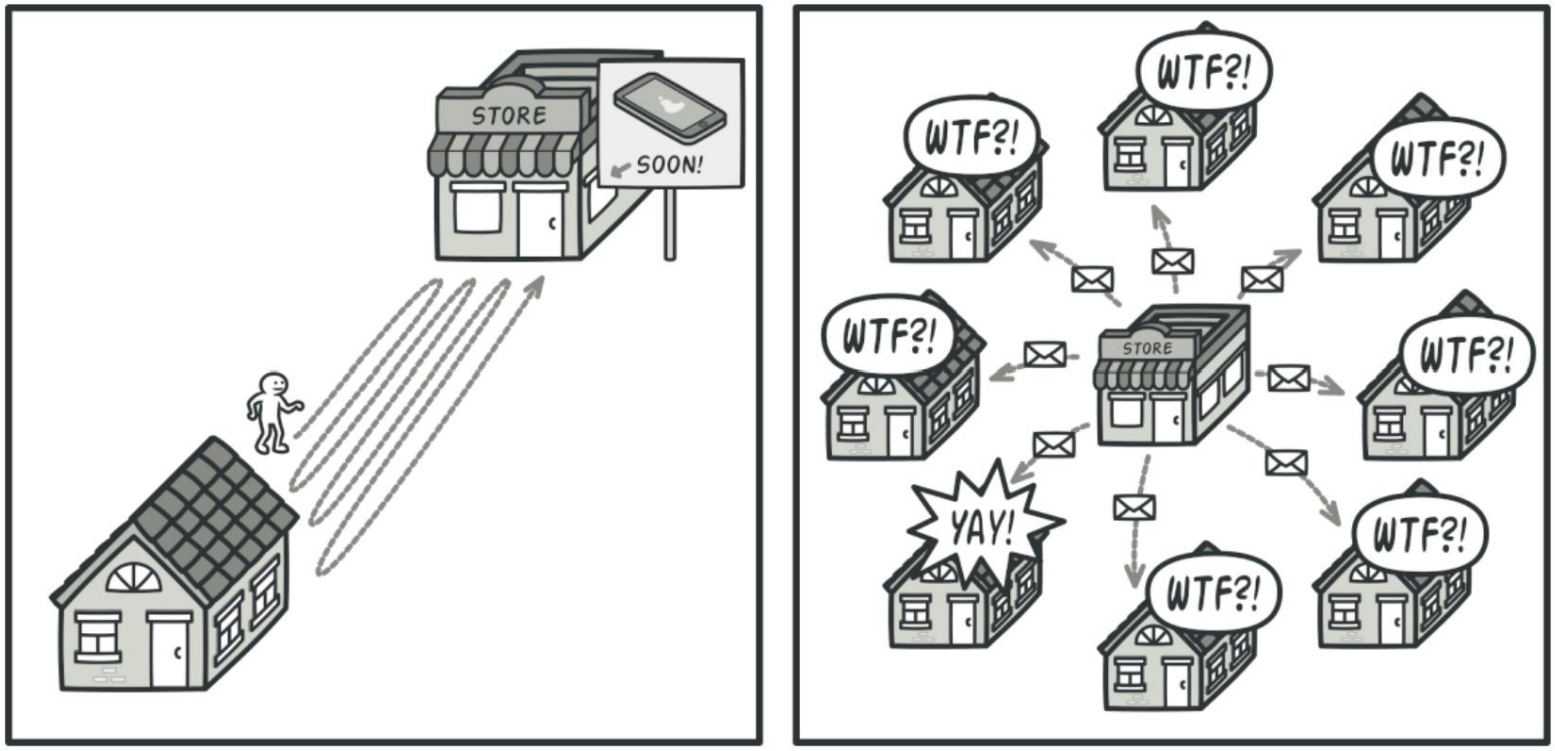
PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 1

WYKŁAD 13

- Wybrane Behawioralne Wzorce Projektowe

Obserwator

Observer to behawioralny wzorzec projektowy, który definiuje mechanizm subskrypcji pozwalający obiektom (obserwatorom) na reakcję na zmiany w innym obiekcie (obserwowanym).

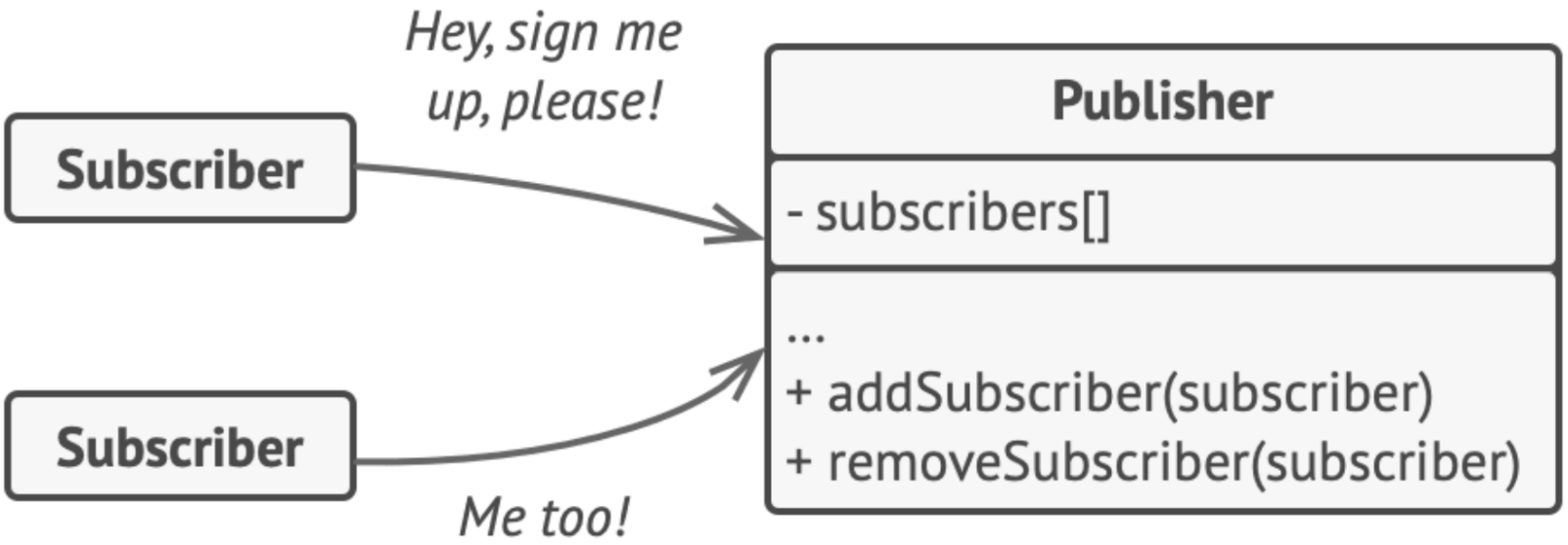


https://refactoring.guru

Obserwator

Kluczowe koncepty

- Subject (obserwowany) – obiekt, który powiadamia obserwatorów o zmianach.
- Observer (obserwator) – interfejs z metodą reakcji na update.

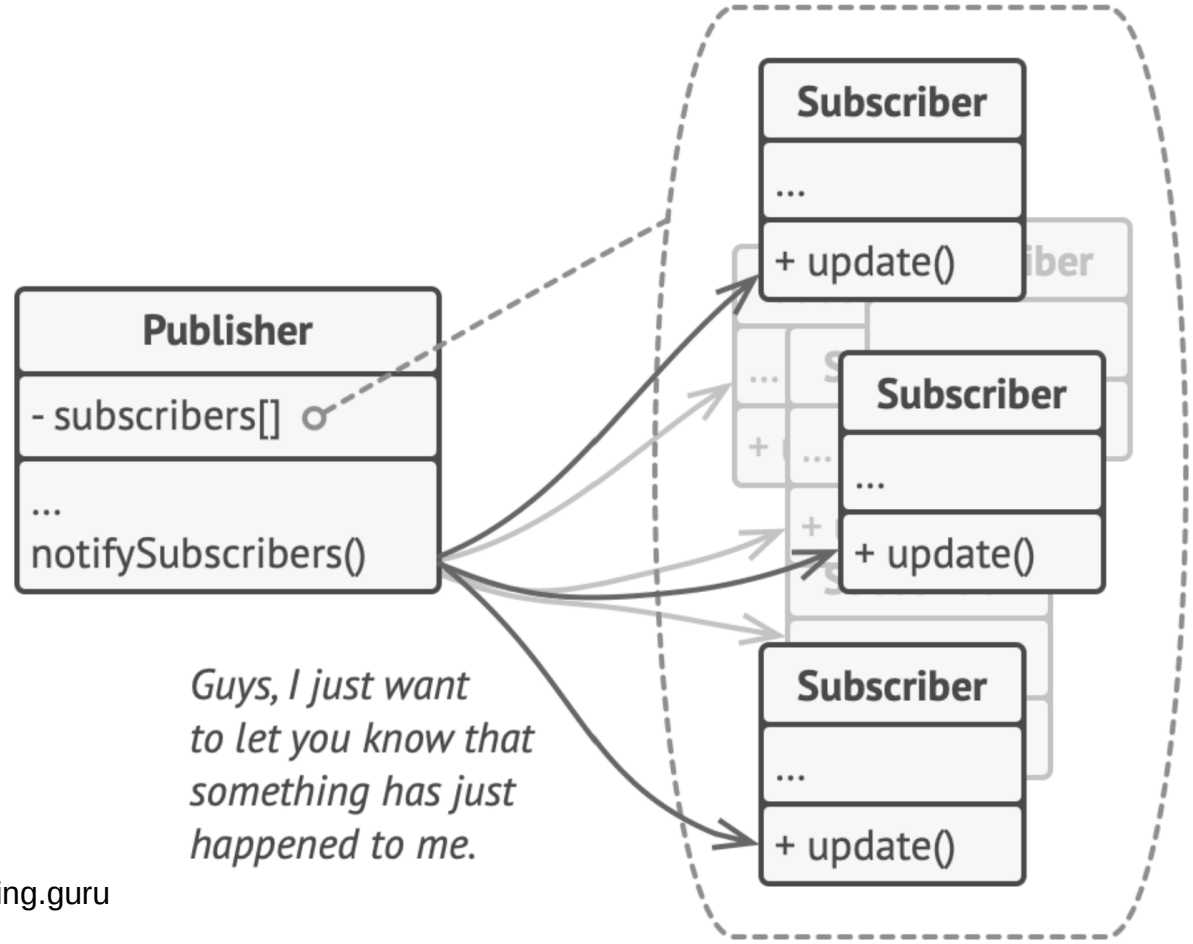


https://refactoring.guru

Obserwator

Kluczowe koncepty

- Subject (obserwowany) – obiekt, który powiadamia obserwatorów o zmianach.
- Observer (obserwator) – interfejs z metodą reakcji na update.



https://refactoring.guru

Obserwator

Kluczowe koncepty

- Subject (obserwowany) – obiekt, który powiadamia obserwatorów o zmianach.
- Observer (obserwator) – interfejs z metodą reakcji na update.

```
interface Observer {
    fun update(message: String)
}
```

Każdy obserwator posiada metodę **update**

Obserwator

Kluczowe koncepty

- Subject (obserwowany) – obiekt, który powiadamia obserwatorów o zmianach.
- Observer (obserwator) – interfejs z metodą reakcji na update.

```
interface Observer {
    fun update(message: String)
}
```

Każdy obserwator posiada metodę **update**

```
class NewsAgency {
    private val observers = mutableListOf<Observer>()

    fun addObserver(observer: Observer) {
        observers.add(observer)
    }

    fun removeObserver(observer: Observer) {
        observers.remove(observer)
    }

    fun notifyObservers(message: String) {
        observers.forEach { it.update(message) }
    }

    fun publishNews(news: String) {
        println("Publikujemy wiadomość: $news")
        notifyObservers(news)
    }
}
```

Obserwator

Kluczowe koncepty

- Subject (obserwowany) – obiekt, który powiadamia obserwatorów o zmianach.
- Observer (obserwator) – interfejs z metodą reakcji na update.

```
interface Observer {
    fun update(message: String)
}
```

Każdy obserwator posiada metodę **update**

```
class NewsAgency {
    private val observers = mutableListOf<Observer>()

    fun addObserver(observer: Observer) {
        observers.add(observer)
    }

    fun removeObserver(observer: Observer) {
        observers.remove(observer)
    }

    fun notifyObservers(message: String) {
        observers.forEach { it.update(message) }
    }

    fun publishNews(news: String) {
        println("Publikujemy wiadomość: $news")
        notifyObservers(news)
    }
}
```

Lista **wszystkich** obserwatorów

Obserwator

Kluczowe koncepty

- Subject (obserwowany) – obiekt, który powiadamia obserwatorów o zmianach.
- Observer (obserwator) – interfejs z metodą reakcji na update.

```
interface Observer {
    fun update(message: String)
}

class NewsAgency {
    private val observers = mutableListOf<Observer>()

    fun addObserver(observer: Observer) {
        observers.add(observer)
    }

    fun removeObserver(observer: Observer) {
        observers.remove(observer)
    }

    fun notifyObservers(message: String) {
        observers.forEach { it.update(message) }
    }

    fun publishNews(news: String) {
        println("Publikujemy wiadomość: $news")
        notifyObservers(news)
    }
}
```

Każdy obserwator posiada metodę **update**

Lista **wszystkich** obserwatorów

Metody **dodania i usunięcia** obserwatora

Obserwator

Kluczowe koncepty

- Subject (obserwowany) – obiekt, który powiadamia obserwatorów o zmianach.
- Observer (obserwator) – interfejs z metodą reakcji na update.

```
interface Observer {
    fun update(message: String)
}

class NewsAgency {
    private val observers = mutableListOf<Observer>()

    fun addObserver(observer: Observer) {
        observers.add(observer)
    }

    fun removeObserver(observer: Observer) {
        observers.remove(observer)
    }

    fun notifyObservers(message: String) {
        observers.forEach { it.update(message) }
    }

    fun publishNews(news: String) {
        println("Publikujemy wiadomość: $news")
        notifyObservers(news)
    }
}
```

Każdy obserwator posiada metodę **update**

Lista **wszystkich** obserwatorów

Metody **dodania i usunięcia** obserwatora

Metoda powiadamia **wszystkich** obserwatorów

Obserwator

Kluczowe koncepty

- Subject (obserwowany) – obiekt, który powiadamia obserwatorów o zmianach.
- Observer (obserwator) – interfejs z metodą reakcji na update.

```
interface Observer {
    fun update(message: String)
}

class NewsAgency {
    private val observers = mutableListOf<Observer>()

    fun addObserver(observer: Observer) {
        observers.add(observer)
    }

    fun removeObserver(observer: Observer) {
        observers.remove(observer)
    }

    fun notifyObservers(message: String) {
        observers.forEach { it.update(message) }
    }

    fun publishNews(news: String) {
        println("Publikujemy wiadomość: $news")
        notifyObservers(news)
    }
}
```

Każdy obserwator posiada metodę **update**

Lista **wszystkich** obserwatorów

Metody **dodania i usunięcia** obserwatora

Metoda powiadamia **wszystkich** obserwatorów

Metoda **publikująca**

Uniwersytet Wrocławski

Obserwator

Kluczowe koncepty

- Subject (obserwowany) – obiekt, który powiadamia obserwatorów o zmianach.
- Observer (obserwator) – interfejs z metodą reakcji na update.

```
interface Observer {
    fun update(message: String)
}

class NewsAgency {
    private val observers = mutableListOf<Observer>()

    fun addObserver(observer: Observer) {
        observers.add(observer)
    }

    fun removeObserver(observer: Observer) {
        observers.remove(observer)
    }

    fun notifyObservers(message: String) {
        observers.forEach { it.update(message) }
    }

    fun publishNews(news: String) {
        println("Publikujemy wiadomość: $news")
        notifyObservers(news)
    }
}
```

```
class NewsChannel(val name: String)
    : Observer {
    override fun update(message: String) {
        println(
            "$name otrzymał wiadomość: $message"
        )
    }
}
```

Obserwator

Kluczowe koncepty

- Subject (obserwowany) – obiekt, który powiadamia obserwatorów o zmianach.
- Observer (obserwator) – interfejs z metodą reakcji na update.

```
interface Observer {
    fun update(message: String)
}

class NewsAgency {
    private val observers = mutableListOf<Observer>()

    fun addObserver(observer: Observer) {
        observers.add(observer)
    }

    fun removeObserver(observer: Observer) {
        observers.remove(observer)
    }

    fun notifyObservers(message: String) {
        observers.forEach { it.update(message) }
    }

    fun publishNews(news: String) {
        println("Publikujemy wiadomość: $news")
        notifyObservers(news)
    }
}
```

```
class NewsChannel(val name: String)
    : Observer {
    override fun update(message: String) {
        println(
            "$name otrzymał wiadomość: $message"
        )
    }
}
```

```
fun main() {
    val agency = NewsAgency()
    val channel1 = NewsChannel("TVN")
    val channel2 = NewsChannel("Polsat")

    agency.addObserver(channel1)
    agency.addObserver(channel2)

    agency.publishNews("Nowy prezydent wybrany!")
    // Output:
    // Publikujemy wiadomość: Nowy prezydent wybrany!
    // TVN otrzymał wiadomość: Nowy prezydent wybrany!
    // Polsat otrzymał wiadomość: Nowy prezydent wybrany!
}
```

Uniwersytet Wrocławski

Obserwator

Kluczowe koncepty

- Subject (obserwowany) – obiekt, który powiadamia obserwatorów o zmianach.
- Observer (obserwator) – interfejs z metodą reakcji na update.

```kotlin
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<nazwa domyślna>") {
            property, oldValue, newValue ->
        println("$oldValue -> $newValue")
    }
}

fun main() {
    val user = User()
    user.name = "Anna"  // <nazwa domyślna> -> Anna
    user.name = "Kasia" // Anna -> Kasia
}
```

Uniwersytet Wrocławski

Obserwator

System eventów w aplikacji Android

```
object EventBus {
    private val listeners = mutableMapOf<String, (String) -> Unit>()

    fun subscribe(eventType: String, listener: (String) -> Unit) {
        listeners[eventType] = listener
    }

    fun publish(eventType: String, message: String) {
        listeners[eventType]?.invoke(message)
    }
}

fun main() {
    EventBus.subscribe("login") { message ->
        println("Login event: $message")
    }

    EventBus.publish("login", "Użytkownik zalogowany!")
    // Output: Login event: Użytkownik zalogowany!
}
```

Uniwersytet Wrocławski

Obserwator – ViewModel + LiveData

```
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel

class CounterViewModel : ViewModel() {
    // Dane obserwowane przez UI
    private val _counter = MutableLiveData(0)
    val counter = _counter // publiczna wersja tylko do odczytu

    fun increment() {
        _counter.value = _counter.value?.plus(1)
    }
}
```

Uniwersytet Wrocławski

Obserwator – ViewModel + LiveData

```
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel

class CounterViewModel : ViewModel() {
    // Dane obserwowane przez UI
    private val _counter = MutableLiveData(0)
    val counter = _counter // publiczna wersja tylko do odczytu

    fun increment() {
        _counter.value = _counter.value?.plus(1)
    }
}
```

```
@Composable
fun CounterScreen(viewModel: CounterViewModel = viewModel()) {
    // Subskrybcja LiveData w Compose (automatyczne obserwowanie zmian)
    val counter by viewModel.counter.observeAsState()

    Column {
        Text("Licznik: $counter")
        Spacer(modifier = Modifier.height(16.dp))
        Button(onClick = { viewModel.increment() }) {
            Text("Zwiększ licznik")
        }
    }
}
```

Stan

State to behawioralny wzorzec projektowy, który pozwala obiektowi **zmieniać swoje zachowanie w zależności od wewnętrznego stanu**. Traktuje stany jako **osobne obiekty** i **deleguje** do nich operacje, unikając rozgałęzień (if/else lub when).

Stan

State to behawioralny wzorzec projektowy, który pozwala obiektowi **zmieniać swoje zachowanie w zależności od wewnętrznego stanu**. Traktuje stany jako **osobne obiekty** i **deleguje** do nich operacje, unikając rozgałęzień (if/else lub when).

Przykłady zastosowań:

- **Automaty stanów** (np. odtwarzacz muzyki: Playing, Paused, Stopped).
- **Zamówienia w e-commerce** (New, Paid, Shipped, Cancelled).
- **Gry wideo** (postać: Walking, Running, Jumping).

Stan

State to behawioralny wzorzec projektowy, który pozwala obiektowi **zmieniać swoje zachowanie w zależności od wewnętrznego stanu**. Traktuje stany jako **osobne obiekty** i **deleguje** do nich operacje, unikając rozgałęzień (if/else lub when)

```
interface State {
    fun play(player: MediaPlayer)
    fun pause(player: MediaPlayer)
    fun stop(player: MediaPlayer)
}
```

Stan

State to behawioralny wzorzec projektowy, który pozwala obiektowi **zmieniać swoje zachowanie w zależności od wewnętrznego stanu**. Traktuje stany jako **osobne obiekty** i **deleguje** do nich operacje, unikając rozgałęzień (if/else lub when)

```
interface State {
    fun play(player: MediaPlayer)
    fun pause(player: MediaPlayer)
    fun stop(player: MediaPlayer)
}


class PlayingState : State {
    override fun play(player: MediaPlayer) {
        println("Odtwarzanie już trwa.")
    }

    override fun pause(player: MediaPlayer) {
        println("Pauzuję odtwarzanie.")
        player.changeState(PausedState())
    }

    override fun stop(player: MediaPlayer) {
        println("Zatrzymuję odtwarzanie.")
        player.changeState(StoppedState())
    }
}
```

Stan

State to behawioralny wzorzec projektowy, który pozwala obiektowi **zmieniać swoje zachowanie w zależności od wewnętrznego stanu**. Traktuje stany jako **osobne obiekty** i **deleguje** do nich operacje, unikając rozgałęzień (if/else lub when)

```
interface State {
    fun play(player: MediaPlayer)
    fun pause(player: MediaPlayer)
    fun stop(player: MediaPlayer)
}
```

```
class PlayingState : State {
    override fun play(player: Me
        println("Odtwarzanie już
    }

    override fun pause(player: M
        println("Pauzuję odtwarz
        player.changeState(Pause
    }

    override fun stop(player: Me
        println("Zatrzymuję odtw
        player.changeState(Stopp
    }
}
```

```
class PausedState : State {
    override fun play(player: MediaPlayer) {
        println("Wznawiam odtwarzanie.")
        player.changeState(PlayingState())
    }

    override fun pause(player: MediaPlayer) {
        println("Już jestem zapauzowany.")
    }

    override fun stop(player: MediaPlayer) {
        println("Zatrzymuję odtwarzanie.")
        player.changeState(StoppedState())
    }
}
```

Stan

State to behawioralny wzorzec projektowy, który pozwala obiektowi **zmieniać swoje zachowanie w zależności od wewnętrznego stanu**. Traktuje stany jako **osobne obiekty** i **deleguje** do nich operacje, unikając rozgałęzień (if/else lub when)

```
interface State {
    fun play(player: MediaPlayer)
    fun pause(player: MediaPlayer)
    fun stop(player: MediaPlayer)
}
```

```
class PlayingState
    override fun p
        println("O
    }

    override fun p
        println("P
        player.cha
    }

    override fun s
        println("Z
        player.cha
    }
}
```

```
class PausedState
    override fun p
        println("W
        player.cha
    }

    override fun p
        println("J
    }

    override fun s
        println("Z
        player.cha
    }
}
```

```
class StoppedState : State {
    override fun play(player: MediaPlayer) {
        println("Rozpoczynam odtwarzanie.")
        player.changeState(PlayingState())
    }

    override fun pause(player: MediaPlayer) {
        println("Nie można pauzować - odtwarzanie zatrzymane.")
    }

    override fun stop(player: MediaPlayer) {
        println("Już jestem zatrzymany.")
    }
}
```

Stan

State to behawioralny wzorzec projektowy, który pozwala obiektowi **zmieniać swoje zachowanie w zależności od wewnętrznego stanu**. Traktuje stany jako **osobne obiekty** i **deleguje** do nich operacje, unikając rozgałęzień (if/else lub when)

```
interface State {
    fun play(player: MediaPlayer)
    fun pause(player: MediaPlayer)
    fun stop(player: MediaPlayer)
}
```

```
class PlayingState : State {
    override fun play(player: MediaPlayer) {
        println("Odtwarzanie już trwa.")
    }

    override fun pause(player: MediaPlayer) {
        println("Pauzuję odtwarzanie.")
        player.changeState(PausedState())
    }

    override fun stop(player: MediaPlayer) {
        println("Zatrzymuję odtwarzanie.")
        player.changeState(StoppedState())
    }
}
```

```
class PausedState : State {
    override fun play(player: MediaPlayer) {
        println("Wznawiam odtwarzanie.")
        player.changeState(PlayingState())
    }

    override fun pause(player: MediaPlayer) {
        println("Już jestem zapauzowany.")
    }

    override fun stop(player: MediaPlayer) {
        println("Zatrzymuję odtwarzanie.")
        player.changeState(StoppedState())
    }
}
```

```
class StoppedState : State {
    override fun play(player: MediaPlayer) {
        println("Rozpoczynam odtwarzanie.")
        player.changeState(PlayingState())
    }

    override fun pause(player: MediaPlayer) {
        println("Nie można pauzować - odtwarzanie zatrzymane.")
    }

    override fun stop(player: MediaPlayer) {
        println("Już jestem zatrzymany.")
    }
}
```

```
class MediaPlayer {
    private var state: State = StoppedState()

    fun changeState(newState: State) {
        this.state = newState
    }

    fun play() {
        state.play(this)
    }

    fun pause() {
        state.pause(this)
    }

    fun stop() {
        state.stop(this)
    }
}
```

Stan

State to behawioralny wzorzec projektowy, który pozwala obiektowi **zmieniać swoje zachowanie w zależności od wewnętrznego stanu**. Traktuje stany jako **osobne obiekty** i **deleguje** do nich operacje, unikając rozgałęzień (if/else lub when)

```
interface State {
    fun play(player: MediaPlayer)
    fun pause(player: MediaPlayer)
    fun stop(player: MediaPlayer)
}
```

```
class PlayingState : State {
    override fun play(player: MediaPlayer) {
        println("Odtwarzanie już trwa.")
    }

    override fun pause(player: MediaPlayer) {
        println("Pauzuję odtwarzanie.")
        player.changeState(PausedState())
    }

    override fun stop(player: MediaPlayer) {
        println("Zatrzymuję odtwarzanie.")
        player.changeState(StoppedState())
    }
}
```

```
class PausedState : State {
    override fun play(player: MediaPlayer) {
        println("Wznawiam odtwarzanie.")
        player.changeState(PlayingState())
    }

    override fun pause(player: MediaPlayer) {
        println("Już jestem zapauzowany.")
    }

    override fun stop(player: MediaPlayer) {
        println("Zatrzymuję odtwarzanie.")
        player.changeState(StoppedState())
    }
}
```

```
class StoppedState : State {
    override fun play(player: MediaPlayer) {
        println("Rozpoczynam odtwarzanie.")
        player.changeState(PlayingState())
    }
}
```

```
class MediaPlayer {
    private var state: State = StoppedState()

    fun changeState(newState: State) {
        this.state = newState
    }

    fun play() {
        state.play(this)
    }

    fun pause() {
        state.pause(this)
    }

    fun stop() {
        state.stop(this)
    }
```

```
val player = MediaPlayer()

player.play()  // Rozpoczynam odtwarzanie. (Stopped → Playing)
player.pause() // Pauzuję odtwarzanie.     (Playing → Paused)
player.play()  // Wznawiam odtwarzanie.    (Paused → Playing)
player.stop()  // Zatrzymuję odtwarzanie.  (Playing → Stopped)
player.pause() // Nie można pauzować - odtwarzanie zatrzymane.
```

Uniwersytet Wrocławski

Wzorzec Stan vs Stan w Jetpack Compose

State w Jetpack Compose

Mechanizm śledzenia zmian danych i automatycznego odświeżania UI.

Używany głównie przez **mutableStateOf**, **remember**, **ViewModel** i **StateFlow**.

Wzorzec Stan vs Stan w Jetpack Compose

State w Jetpack Compose

Mechanizm śledzenia zmian danych i automatycznego odświeżania UI.

Używany głównie przez **mutableStateOf**, **remember**, **ViewModel** i **StateFlow**.

Wzorzec State Pattern

Wzorzec projektowy, który **enkapsuluje zachowanie obiektu** w zależności od jego stanu.

Stany są reprezentowane jako osobne klasy (np. **PlayingState**, **PausedState**).

Cecha	State w Compose	Wzorzec State Pattern
Cel	Reaktywność UI	Zarządzanie zachowaniem obiektu
Implementacja	mutableStateOf, ViewModel	Interfejsy + klasy stanów
Złożoność	Niska (dla prostych przypadków)	Wysoka (dla skomplikowanej logiki stanów)

Uniwersytet Wrocławski

Strategia

Strategia to behawioralny wzorzec projektowy, który definiuje rodzinę wymiennych algorytmów i pozwala wybierać je w czasie działania programu.

Uniwersytet Wrocławski

Strategia

Strategia to behawioralny wzorzec projektowy, który definiuje rodzinę wymiennych algorytmów i pozwala wybierać je w czasie działania programu.

```
interface PaymentStrategy {
    fun pay(amount: Double): String
}
```

Strategia

Strategia to behawioralny wzorzec projektowy, który definiuje rodzinę wymiennych algorytmów i pozwala wybierać je w czasie działania programu.

```
interface PaymentStrategy {
    fun pay(amount: Double): String
}

class CreditCardStrategy : PaymentStrategy {
    override fun pay(amount: Double): String {
        return "Płatność kartą: $amount zł"
    }
}

class PayPalStrategy : PaymentStrategy {
    override fun pay(amount: Double): String {
        return "Płatność PayPal: $amount zł"
    }
}

class BankTransferStrategy : PaymentStrategy {
    override fun pay(amount: Double): String {
        return "Przelew bankowy: $amount zł"
    }
}
```

Uniwersytet Wrocławski

Strategia

Strategia to behawioralny wzorzec projektowy, który definiuje rodzinę wymiennych algorytmów i pozwala wybierać je w czasie działania programu.

```
interface PaymentStrategy {
    fun pay(amount: Double): String
}

class CreditCardStrategy : PaymentStrategy {
    override fun pay(amount: Double): String {
        return "Płatność kartą: $amount zł"
    }
}

class PayPalStrategy : PaymentStrategy {
    override fun pay(amount: Double): String {
        return "Płatność PayPal: $amount zł"
    }
}

class BankTransferStrategy : PaymentStrategy {
    override fun pay(amount: Double): String {
        return "Przelew bankowy: $amount zł"
    }
}
```

```
class PaymentContext(
    private var strategy: PaymentStrategy
) {
    fun executePayment(amount: Double): String {
        return strategy.pay(amount)
    }

    fun changeStrategy(newStrategy: PaymentStrategy) {
        strategy = newStrategy
    }
}
```

Uniwersytet Wrocławski

Strategia

Strategia to behawioralny wzorzec projektowy, który definiuje rodzinę wymiennych algorytmów i pozwala wybierać je w czasie działania programu.

```
interface PaymentStrategy {
    fun pay(amount: Double): String
}

class CreditCardStrategy : PaymentStrategy {
    override fun pay(amount: Double): String {
        return "Płatność kartą: $amount zł"
    }
}

class PayPalStrategy : PaymentStrategy {
    override fun pay(amount: Double): String {
        return "Płatność PayPal: $amount zł"
    }
}

class BankTransferStrategy : PaymentStrategy {
    override fun pay(amount: Double): String {
        return "Przelew bankowy: $amount zł"
    }
}
```

```
class PaymentContext(
    private var strategy: PaymentStrategy
) {
    fun executePayment(amount: Double): String {
        return strategy.pay(amount)
    }

    fun changeStrategy(newStrategy: PaymentStrategy) {
        strategy = newStrategy
    }
}

fun main() {
    val context = PaymentContext(CreditCardStrategy())

    println(context.executePayment(100.0))

    context.changeStrategy(PayPalStrategy())
    println(context.executePayment(50.0))
}
```

Strategia vs Stan

Strategia to behawioralny wzorzec projektowy, który definiuje rodzinę wymiennych algorytmów i pozwala wybierać je w czasie działania programu.

Strategia	**State**
Algorytmy są **niezależne od siebie**	Stany **znają inne stany** (np. PausedState wie o PlayingState)
Wybór strategii jest **świadomy** (np. wybór płatności)	Zmiana stanu **dzieje się automatycznie** (np. po kliknięciu "pause")

Uniwersytet Wrocławski

Mediator

Mediator to behawioralny wzorzec projektowy, który centralizuje komunikację między obiektami w systemie, zamiast pozwalać im odwoływać się do siebie bezpośrednio. Działa jak "router" lub "dyspozytor" zdarzeń.

Mediator

Mediator to behawioralny wzorzec projektowy, który centralizuje komunikację między obiektami w systemie, zamiast pozwalać im odwoływać się do siebie bezpośrednio. Działa jak "router" lub "dyspozytor" zdarzeń.

Kluczowe elementy

Mediator – interfejs lub klasa pośrednicząca w komunikacji.

Colleague (Komponent) – obiekty, które komunikują się przez Mediatora (zamiast bezpośrednio).

Mediator

Mediator to behawioralny wzorzec projektowy, który centralizuje komunikację między obiektami w systemie, zamiast pozwalać im odwoływać się do siebie bezpośrednio. Działa jak "router" lub "dyspozytor" zdarzeń.

```
interface ChatMediator {
    fun sendMessage(message: String, sender: User)
    fun addUser(user: User)
}
```

Mediator

Mediator to behawioralny wzorzec projektowy, który centralizuje komunikację między obiektami w systemie, zamiast pozwalać im odwoływać się do siebie bezpośrednio. Działa jak "router" lub "dyspozytor" zdarzeń.

```
interface ChatMediator {
    fun sendMessage(message: String, sender: User)
    fun addUser(user: User)
}

class GroupChatMediator : ChatMediator {
    private val users = mutableListOf<User>()

    override fun addUser(user: User) {
        users.add(user)
    }

    override fun sendMessage(
        message: String, sender: User
    ) {
        users.filter { it != sender }.forEach {
            it.receive(message)
        }
    }
}
```

Mediator

Mediator to behawioralny wzorzec projektowy, który centralizuje komunikację między obiektami w systemie, zamiast pozwalać im odwoływać się do siebie bezpośrednio. Działa jak "router" lub "dyspozytor" zdarzeń.

```
interface ChatMediator {
    fun sendMessage(message: String, sender: User)
    fun addUser(user: User)
}

class GroupChatMediator : ChatMediator {
    private val users = mutableListOf<User>()

    override fun addUser(user: User) {
        users.add(user)
    }

    override fun sendMessage(
        message: String, sender: User
    ) {
        users.filter { it != sender }.forEach {
            it.receive(message)
        }
    }
}
```

```
class User(val name: String,
           private val mediator: ChatMediator
) {
    init {
        mediator.addUser(this)
    }

    fun send(message: String) {
        println("$name wysyła: '$message'")
        mediator.sendMessage(message, this)
    }

    fun receive(message: String) {
        println("$name otrzymał: '$message'")
    }
}
```

Uniwersytet Wrocławski

Mediator

Mediator to behawioralny wzorzec projektowy, który centralizuje komunikację między obiektami w systemie, zamiast pozwalać im odwoływać się do siebie bezpośrednio. Działa jak "router" lub "dyspozytor" zdarzeń.

```
interface ChatMediator {
    fun sendMessage(message: String, sender: User)
    fun addUser(user: User)
}

class GroupChatMediator : ChatMediator {
    private val users = mutableListOf<User>()

    override fun addUser(user: User) {
        users.add(user)
    }

    override fun sendMessage(
        message: String, sender: User
    ) {
        users.filter { it != sender }.forEach {
            it.receive(message)
        }
    }
}
```

```
class User(val name: String,
           private val mediator: ChatMediator
) {
    init {
        mediator.addUser(this)
    }

    fun send(message: String) {
        println("$name wysyła: '$message'")
        mediator.sendMessage(message, this)
    }

    fun receive(message: String) {
        println("$name otrzymał: '$message'")
    }
}
```

```
val mediator = GroupChatMediator()
val alice = User("Alice", mediator)
val bob = User("Bob", mediator)
val charlie = User("Charlie", mediator)

alice.send("Cześć wszystkim!")
```

```
// Output:
// Alice wysyła: 'Cześć wszystkim!'
// Bob otrzymał: 'Cześć wszystkim!'
// Charlie otrzymał: 'Cześć wszystkim!'
```