Uniwersytet Wrocławski

Rafał Lewandków

PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 1

WYKŁAD 12

- Wybrane Strukturalne i Kreacyjne Wzorce Projektowe
- Adnotacje

Uniwersytet Wrocławski

Wzorce projektowe

Wzorzec opisuje problem, który powtarza się wielokrotnie w danym środowisku, oraz podaje istotę jego rozwiązania w taki sposób, aby można było je zastosować miliony razy bez potrzeby powtarzania tej samej pracy.

Christopher Alexander "A pattern language", 1977

Wzorce projektowe

Wzorzec opisuje problem, który powtarza się wielokrotnie w danym środowisku, oraz podaje istotę jego rozwiązania w taki sposób, aby można było je zastosować miliony razy bez potrzeby powtarzania tej samej pracy.

Christopher Alexander "A pattern language", 1977

Wzorce projektowe (ang. *design patterns*) to sprawdzone i wielokrotnie przetestowane rozwiązania typowych problemów, które pojawiają się podczas projektowania oprogramowania.

Wzorce projektowe

Wzorzec opisuje problem, który powtarza się wielokrotnie w danym środowisku, oraz podaje istotę jego rozwiązania w taki sposób, aby można było je zastosować miliony razy bez potrzeby powtarzania tej samej pracy.

Christopher Alexander "A pattern language", 1977

Wzorce projektowe (ang. *design patterns*) to sprawdzone i wielokrotnie przetestowane rozwiązania typowych problemów, które pojawiają się podczas projektowania oprogramowania.

Kluczowe cechy wzorców projektowych:

- **Uniwersalność** – można je zastosować w różnych projektach i językach programowania.
- **Sprawdzone rozwiązania** – wynikają z doświadczeń wielu programistów.
- **Abstrakcyjność** – opisują koncepcję, a nie konkretną implementację.
- **Ułatwiają komunikację** – stanowią wspólny język dla zespołów deweloperskich.

Wzorce projektowe

Wzorzec opisuje problem, który powtarza się wielokrotnie w danym środowisku, oraz podaje istotę jego rozwiązania w taki sposób, aby można było je zastosować miliony razy bez potrzeby powtarzania tej samej pracy.

Christopher Alexander "A pattern language", 1977

Wzorce projektowe (ang. *design patterns*) to sprawdzone i wielokrotnie przetestowane rozwiązania typowych problemów, które pojawiają się podczas projektowania oprogramowania.

Kluczowe cechy wzorców projektowych:

- **Uniwersalność** – można je zastosować w różnych projektach i językach programowania.
- **Sprawdzone rozwiązania** – wynikają z doświadczeń wielu programistów.
- **Abstrakcyjność** – opisują koncepcję, a nie konkretną implementację.
- **Ułatwiają komunikację** – stanowią wspólny język dla zespołów deweloperskich.

Podział wzorców projektowych (wg. "*Gang of Four*" – GoF):

1.Wzorce kreacyjne – dotyczą tworzenia obiektów (np. Singleton, Fabryka).

2.Wzorce strukturalne – dotyczą kompozycji obiektów (np. Adapter, Dekorator).

3.Wzorce behawioralne – dotyczą komunikacji między obiektami (np. Observer).

Singleton

Singleton to wzorzec kreacyjny, który gwarantuje istnienie tylko jednej instancji danej klasy w całym systemie i zapewnia globalny dostęp do niej. Jest często używany do zarządzania zasobami współdzielonymi (np. połączeniami do bazy danych, konfiguracją).

Singleton

Singleton to wzorzec kreacyjny, który gwarantuje istnienie tylko jednej instancji danej klasy w całym systemie i zapewnia globalny dostęp do niej. Jest często używany do zarządzania zasobami współdzielonymi (np. połączeniami do bazy danych, konfiguracją).

Cechy Singletona:

- Tylko jedna instancja obiektu.
- Kontrolowany dostęp do tej instancji (np. przez metodę getInstance()).
- Prywatny konstruktor (blokuje tworzenie nowych obiektów z zewnątrz).

Uniwersytet Wrocławski

Singleton

Singleton to wzorzec kreacyjny, który gwarantuje istnienie tylko jednej instancji danej klasy w całym systemie i zapewnia globalny dostęp do niej. Jest często używany do zarządzania zasobami współdzielonymi (np. połączeniami do bazy danych, konfiguracją).

Cechy Singletona:

- Tylko jedna instancja obiektu.
- Kontrolowany dostęp do tej instancji (np. przez metodę getInstance()).
- Prywatny konstruktor (blokuje tworzenie nowych obiektów z zewnątrz).

```
class SettingsManager private constructor() {
    companion object {
        val instance: SettingsManager by lazy { SettingsManager() }
    }

    fun loadConfig() {
        println("Loading configuration...")
    }
}
```

Prywatny konstruktor

Singleton

Singleton to wzorzec kreacyjny, który gwarantuje istnienie tylko jednej instancji danej klasy w całym systemie i zapewnia globalny dostęp do niej. Jest często używany do zarządzania zasobami współdzielonymi (np. połączeniami do bazy danych, konfiguracją).

Cechy Singletona:

- Tylko jedna instancja obiektu.
- Kontrolowany dostęp do tej instancji (np. przez metodę getInstance()).
- Prywatny konstruktor (blokuje tworzenie nowych obiektów z zewnątrz).

Prywatny konstruktor

Umożliwia dostęp do instance

```
class SettingsManager private constructor() {
    companion object {
        val instance: SettingsManager by lazy { SettingsManager() }
    }

    fun loadConfig() {
        println("Loading configuration...")
    }
}
```

Singleton

Singleton to wzorzec kreacyjny, który gwarantuje istnienie tylko jednej instancji danej klasy w całym systemie i zapewnia globalny dostęp do niej. Jest często używany do zarządzania zasobami współdzielonymi (np. połączeniami do bazy danych, konfiguracją).

Cechy Singletona:

- Tylko jedna instancja obiektu.
- Kontrolowany dostęp do tej instancji (np. przez metodę getInstance()).
- Prywatny konstruktor (blokuje tworzenie nowych obiektów z zewnątrz).

```
class SettingsManager private constructor() {
    companion object {
        val instance: SettingsManager by lazy { SettingsManager() }
    }

    fun loadConfig() {
        println("Loading configuration...")
    }
}
```

Prywatny konstruktor

Umożliwia dostęp do instance

inicjalizuje obiekt dopiero przy pierwszym użyciu (**leniwa inicjalizacja**) **lazy** w Kotlinie jest **thread-safe**

Uniwersytet Wrocławski

Singleton

Słowo kluczowe

```
object DatabaseManager {
    fun connect() {
        println("Connected to database!")
    }
}
```

Singleton

Słowo kluczowe

```
object DatabaseManager {
    fun connect() {
        println("Connected to database!")
    }
}
```

Prywatny konstruktor

```
class ThreadSafeSingleton private constructor() {
    companion object {
        @Volatile
        private var instance: ThreadSafeSingleton? = null

        fun getInstance(): ThreadSafeSingleton {
            return instance ?: synchronized(this) {
                instance ?: ThreadSafeSingleton().also { instance = it }
            }
        }
    }

    fun doSomething() {
        println("Doing something safely!")
    }
}
```

Singleton

Słowo kluczowe

```
object DatabaseManager {
    fun connect() {
        println("Connected to database!")
    }
}
```

Prywatny konstruktor

```
class ThreadSafeSingleton private constructor() {
    companion object {
        @Volatile
        private var instance: ThreadSafeSingleton? = null

        fun getInstance(): ThreadSafeSingleton {
            return instance ?: synchronized(this) {
                instance ?: ThreadSafeSingleton().also { instance = it }
            }
        }
    }

    fun doSomething() {
        println("Doing something safely!")
    }
}
```

zapewnia, że zmiana wartości instance jest **natychmiast widoczna** dla wszystkich wątków (zapobiega problemom z **pamięcią podręczną** CPU).

Singleton

Słowo kluczowe

```
object DatabaseManager {
    fun connect() {
        println("Connected to database!")
    }
}
```

Prywatny konstruktor

przechowuje **jedyną instancję** Singletona (domyślnie null).

zapewnia, że zmiana wartości instance jest **natychmiast widoczna** dla wszystkich wątków (zapobiega problemom z **pamięcią podręczną** CPU).

```
class ThreadSafeSingleton private constructor() {
    companion object {
        @Volatile
        private var instance: ThreadSafeSingleton? = null

        fun getInstance(): ThreadSafeSingleton {
            return instance ?: synchronized(this) {
                instance ?: ThreadSafeSingleton().also { instance = it }
            }
        }
    }

    fun doSomething() {
        println("Doing something safely!")
    }
}
```

Singleton

Słowo kluczowe

```
object DatabaseManager {
    fun connect() {
        println("Connected to database!")
    }
}
```

Prywatny konstruktor

przechowuje **jedyną instancję** Singletona (domyślnie null).

zapewnia, że zmiana wartości instance jest **natychmiast widoczna** dla wszystkich wątków (zapobiega problemom z **pamięcią podręczną** CPU).

blokuje dostęp dla innych wątków, dopóki jeden wątek nie zakończy inicjalizacji.

```
class ThreadSafeSingleton private constructor() {
    companion object {
        @Volatile
        private var instance: ThreadSafeSingleton? = n

        fun getInstance(): ThreadSafeSingleton {
            return instance ?: synchronized(this) {
                instance ?: ThreadSafeSingleton().also { instance = it }
            }
        }
    }

    fun doSomething() {
        println("Doing something safely!")
    }
}
```

Singleton

Słowo kluczowe

```
object DatabaseManager {
    fun connect() {
        println("Connected to database!")
    }
}
```

Prywatny konstruktor

przechowuje **jedyną instancję** Singletona (domyślnie null).

zapewnia, że zmiana wartości instance jest **natychmiast widoczna** dla wszystkich wątków (zapobiega problemom z **pamięcią podręczną** CPU).

blokuje dostęp dla innych wątków, dopóki jeden wątek nie zakończy inicjalizacji.

tworzy nową instancję **tylko**, jeśli instance jest **nadal null** wewnątrz bloku **synchronized**.

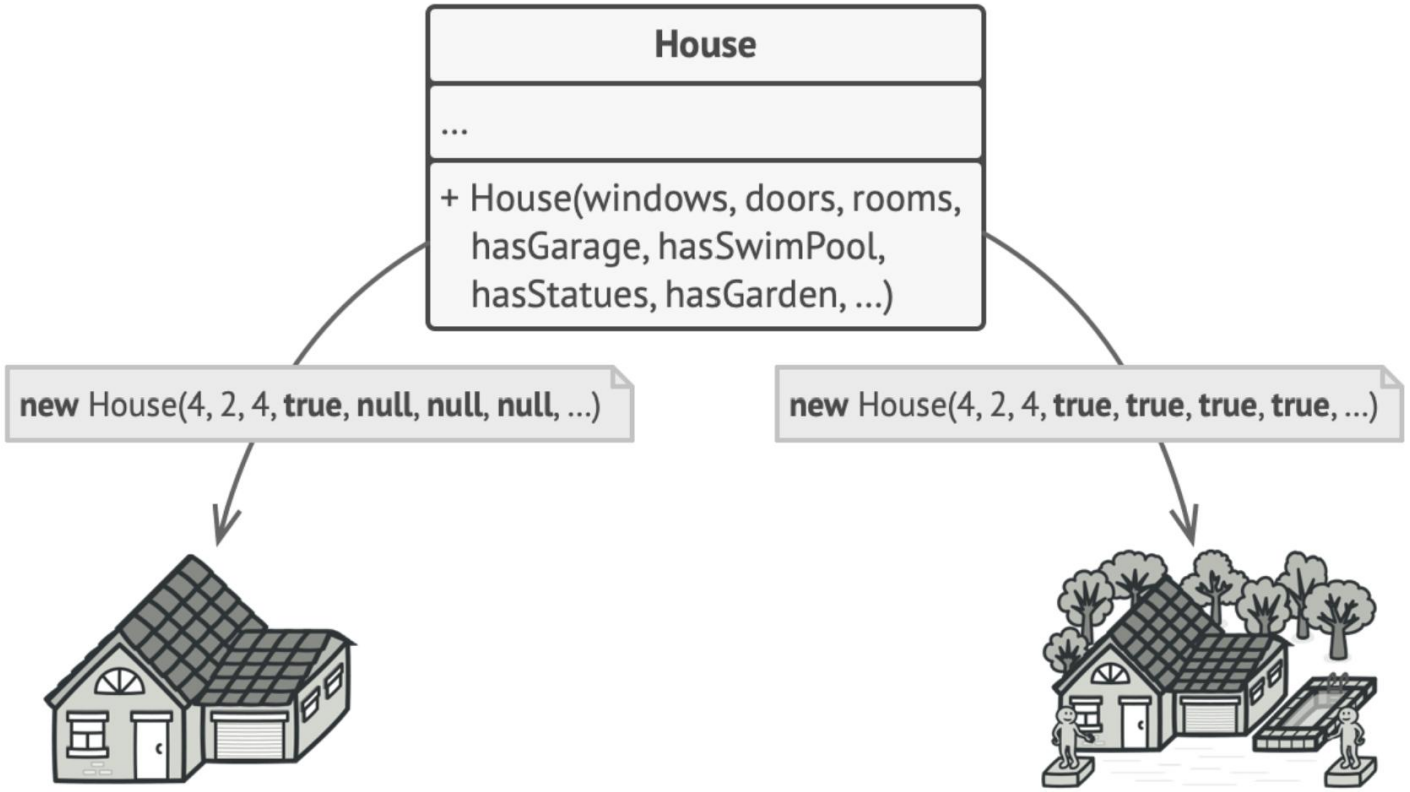
```
class ThreadSafeSingleton private constructor() {
    companion object {
        @Volatile
        private var instance: ThreadSafeSingleton? = n

        fun getInstance(): ThreadSafeSingleton {
            return instance ?: synchronized(this) {
                instance ?: ThreadSafeSingleton().also { instance = it }
            }
        }
    }

    fun doSomething() {
        println("Doing something safely!")
    }
}
```

Budowniczy

Budowniczy to wzorzec kreacyjny, który umożliwia konstruowanie złożonych obiektów krok po kroku, oddzielając proces budowania od jego reprezentacji. Jest szczególnie przydatny, gdy obiekt ma wiele pól konfiguracyjnych lub wymaga różnych wariantów konstrukcyjnych.



Budowniczy

require to funkcja wbudowana, która służy do **walidacji argumentów** lub stanu obiektu **na początku funkcji lub konstruktora**.

require(warunek) { "Komunikat błędu" }

```
data class Computer(
    val cpu: String,
    val gpu: String,
    val ramGB: Int,
    val storageGB: Int,
    val hasSSD: Boolean
) {
    init {
        require(ramGB > 0) { "RAM must be positive" }
        require(storageGB > 0) { "Storage must be positive" }
    }
}
```

Budowniczy

require to funkcja wbudowana, która służy do **walidacji argumentów** lub stanu obiektu **na początku funkcji lub konstruktora**.
require(warunek) { "Komunikat błędu" }

```
data class Computer(
    val cpu: String,
    val gpu: String,
    val ramGB: Int,
    val storageGB: Int,
    val hasSSD: Boolean
) {
    init {
        require(ramGB > 0) { "RAM must be positive" }
        require(storageGB > 0) { "Storage must be positive" }
    }
}
```

apply {} umożliwia **łańcuchowanie wywołań** (np. builder.setCPU(...).setRAM(...)).

```
class ComputerBuilder {
    private var cpu: String = "Intel i5"
    private var gpu: String = "Integrated"
    private var ramGB: Int = 8
    private var storageGB: Int = 256
    private var hasSSD: Boolean = false

    fun setCPU(cpu: String) = apply { this.cpu = cpu }
    fun setGPU(gpu: String) = apply { this.gpu = gpu }
    fun setRAM(ramGB: Int) = apply { this.ramGB = ramGB }
    fun setStorage(storageGB: Int) = apply { this.storageGB = storageGB }
    fun useSSD() = apply { this.hasSSD = true }

    fun build(): Computer {
        return Computer(cpu, gpu, ramGB, storageGB, hasSSD)
    }
}
```

Budowniczy

require to funkcja wbudowana, która służy do **walidacji argumentów** lub stanu obiektu **na początku funkcji lub konstruktora**.
require(warunek) { "Komunikat błędu" }

```
data class Computer(
    val cpu: String,
    val gpu: String,
    val ramGB: Int,
    val storageGB: Int,
    val hasSSD: Boolean
) {
    init {
        require(ramGB > 0) { "RAM must be positive" }
        require(storageGB > 0) { "Storage must be positive" }
    }
}
```

apply {} umożliwia **łańcuchowanie wywołań** (np. builder.setCPU(...).setRAM(...)).

```
class ComputerBuilder {
    private var cpu: String = "Intel i5"
    private var gpu: String = "Integrated"
    private var ramGB: Int = 8
    private var storageGB: Int = 256
    private var hasSSD: Boolean = false

    fun setCPU(cpu: String) = apply { this.cpu = cpu }
    fun setGPU(gpu: String) = apply { this.gpu = gpu }
    fun setRAM(ramGB: Int) = apply { this.ramGB = ramGB }
    fun setStorage(storageGB: Int) = apply { this.storageGB = storageGB }
    fun useSSD() = apply { this.hasSSD = true }

    fun build(): Computer {
        return Computer(cpu, gpu, ramGB, storageGB, hasSSD)
    }
}
```

```
val gamingPC = ComputerBuilder()
    .setCPU("AMD Ryzen 9")
    .setGPU("NVIDIA RTX 3080")
    .setRAM(32)
    .setStorage(1000)
    .useSSD()
    .build()
```

Budowniczy

require to funkcja wbudowana, która służy do **walidacji argumentów** lub stanu obiektu **na początku funkcji lub konstruktora**.
require(warunek) { "Komunikat błędu" }

```
data class Computer(
    val cpu: String,
    val gpu: String,
    val ramGB: Int,
    val storageGB: Int,
    val hasSSD: Boolean
) {
    init {
        require(ramGB > 0) { "RAM must be positive" }
        require(storageGB > 0) { "Storage must be positive" }
    }
}
```

apply {} umożliwia **łańcuchowanie wywołań** (np. builder.setCPU(...).setRAM(...)).

```
class ComputerBuilder {
    private var cpu: String = "Intel i5"
    private var gpu: String = "Integrated"
    private var ramGB: Int = 8
    private var storageGB: Int = 256
    private var hasSSD: Boolean = false

    fun setCPU(cpu: String) = apply { this.cpu = cpu }
    fun setGPU(gpu: String) = apply { this.gpu = gpu }
    fun setRAM(ramGB: Int) = apply { this.ramGB = ramGB }
    fun setStorage(storageGB: Int) = apply { this.storageGB = storageGB }
    fun useSSD() = apply { this.hasSSD = true }

    fun build(): Computer {
        return Computer(cpu, gpu, ramGB, storageGB, hasSSD)
    }
}
```

```
fun buildComputer(
    block: ComputerBuilder.() -> Unit)
: Computer {
    return ComputerBuilder().apply(block).build()
}

val officePC = buildComputer {
    setCPU("Intel i3")
    setRAM(16)
    setStorage(512)
}
```

Uniwersytet Wrocławski

Fabryka

Fabryka to wzorzec kreacyjny, który oddziela proces tworzenia obiektów od ich konkretnych klas, pozwalając na wybór odpowiedniej implementacji bez bezpośredniego użycia konstruktorów.

Uniwersytet Wrocławski

Fabryka

Fabryka to wzorzec kreacyjny, który oddziela proces tworzenia obiektów od ich konkretnych klas, pozwalając na wybór odpowiedniej implementacji bez bezpośredniego użycia konstruktorów.

```
interface Vehicle {
    fun drive()
}

class Car : Vehicle {
    override fun drive() = println("Jadę samochodem!")
}

class Bike : Vehicle {
    override fun drive() = println("Jadę rowerem!")
}

object VehicleFactory {
    fun createVehicle(type: String): Vehicle {
        return when (type.lowercase()) {
            "car" -> Car()
            "bike" -> Bike()
            else -> throw IllegalArgumentException("Nieznany typ pojazdu")
        }
    }
}
```

Uniwersytet Wrocławski

Fabryka

Fabryka to wzorzec kreacyjny, który oddziela proces tworzenia obiektów od ich konkretnych klas, pozwalając na wybór odpowiedniej implementacji bez bezpośredniego użycia konstruktorów.

```
interface Vehicle {
    fun drive()

    companion object Factory {
        fun create(type: String): Vehicle {
            return when (type.lowercase()) {
                "car" -> Car()
                "bike" -> Bike()
                else -> throw IllegalArgumentException("Nieznany typ pojazdu")
            }
        }
    }
}
```

Uniwersytet Wrocławski

Fabryka

Fabryka to wzorzec kreacyjny, który oddziela proces tworzenia obiektów od ich konkretnych klas, pozwalając na wybór odpowiedniej implementacji bez bezpośredniego użycia konstruktorów.

```
interface VehicleFactory {
    fun createVehicle(): Vehicle
    fun createEngine(): Engine
}

class CarFactory : VehicleFactory {
    override fun createVehicle() = Car()
    override fun createEngine() = PetrolEngine()
}

class BikeFactory : VehicleFactory {
    override fun createVehicle() = Bike()
    override fun createEngine() = HumanEngine()
}
```

Fabryka

Fabryka to wzorzec kreacyjny, który oddziela proces tworzenia obiektów od ich konkretnych klas, pozwalając na wybór odpowiedniej implementacji bez bezpośredniego użycia konstruktorów.

```
object VehicleFactory {
    private val creators = mutableMapOf<String, () -> Vehicle>()

    fun register(type: String, creator: () -> Vehicle) {
        creators[type] = creator
    }

    fun create(type: String): Vehicle {
        return creators[type]?.invoke()
            ?: throw IllegalArgumentException("Nieznany typ: $type")
    }
}
```

invoke() to specjalna funkcja, która pozwala na wywołanie obiektu jak funkcji

```
VehicleFactory.register("car") { Car() }
VehicleFactory.register("bike") { Bike() }
```

Uniwersytet Wrocławski

Adapter

Adapter to strukturalny wzorzec projektowy, który pozwala obiektom o niezgodnych interfejsach współpracować ze sobą.

Adapter

Adapter to strukturalny wzorzec projektowy, który pozwala obiektom o niezgodnych interfejsach współpracować ze sobą.

```
class CassettePlayer {
    fun playCassette(fileName: String) {
        println("Odtwarzanie z kasety: $fileName")
    }
}

interface MediaPlayer {
    fun play(fileName: String)
}

class CassetteAdapter(private val cassettePlayer: CassettePlayer) : MediaPlayer {
    override fun play(fileName: String) {
        cassettePlayer.playCassette(fileName)
    }
}
```

Adapter

Adapter to strukturalny wzorzec projektowy, który pozwala obiektom o niezgodnych interfejsach współpracować ze sobą.

```
val users = listOf(
    User("Alicja"),
    User("Bartek"),
    User("Celina"),
    User("Daniel")
)
UserList(users)
```

```
data class User(val name: String)

@Composable
fun UserList(users: List<User>) {
    LazyColumn {
        items(users.count()) { index ->
            UserItem(users[index])
        }
    }
}

@Composable
fun UserItem(user: User) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(8.dp)
    ) {
        Text(
            text = user.name,
            modifier = Modifier.padding(16.dp)
        )
    }
}
```

Adapter - ViewHolder

ViewHolder to wzorzec projektowy (często uważany za część wzorca **Adapter**), stosowany głównie w Androidzie w celu optymalizacji wydajności list.

Cel wzorca ViewHolder

Gdy przewijasz listę, widoki są recyklingowane – zamiast tworzyć nowe obiekty za każdym razem, system używa istniejących.

- Przechowuje referencje do widoków w pojedynczym elemencie listy,
- Przyspiesza renderowanie i zmniejsza zużycie pamięci.

Uniwersytet Wrocławski

Adapter - ViewHolder

ViewHolder to wzorzec projektowy (często uważany za część wzorca **Adapter**), stosowany głównie w Androidzie w celu optymalizacji wydajności list.

```
val users = listOf(
    User("Alicja"),
    User("Bartek"),
    User("Celina"),
    User("Daniel")
)
UserList(users)
```

```
data class User(val name: String)

@Composable
fun UserList(users: List<User>) {
    LazyColumn {
        items(users.count()) { index ->
            UserItem(users[index])
        }
    }
}

@Composable
fun UserItem(user: User) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(8.dp)
    ) {
        Text(
            text = user.name,
            modifier = Modifier.padding(16.dp)
        )
    }
}
```

Uniwersytet Wrocławski

Adnotacje

Adnotacje to metadane, które można dodawać do deklaracji klas, funkcji, właściwości, parametrów i innych elementów kodu. Nie wpływają bezpośrednio na logikę programu, ale **mogą być wykorzystywane przez kompilator**, narzędzia deweloperskie (np. Android Studio) lub tzw. **procesory adnotacji**.

Adnotacja to specjalny znacznik, który może:

- dodawać informacji do kodu,
- wpływać na działanie kompilatora lub frameworków,
- sterować generowaniem kodu.

```
@Entity
data class User(
    @PrimaryKey val id: Int,
    val name: String
)
```

Adnotacje

Procesor adnotacji (ang. annotation processor) to narzędzie, które analizuje adnotacje w czasie kompilacji i może generować dodatkowy kod, walidować strukturę kodu, itp. Programista oznacza kod adnotacjami. Kompilator uruchamia procesory adnotacji. Procesor odczytuje adnotacje i np. generuje klasy źródłowe, pliki XML, itp.

Adnotacje

Procesor adnotacji (ang. annotation processor) to narzędzie, które analizuje adnotacje w czasie kompilacji i może generować dodatkowy kod, walidować strukturę kodu, itp. Programista oznacza kod adnotacjami. Kompilator uruchamia procesory adnotacji. Procesor odczytuje adnotacje i np. generuje klasy źródłowe, pliki XML, itp.

Narzędzia do przetwarzania adnotacji

- **kapt -** Domyślne narzędzie do przetwarzania adnotacji w Kotlinie
- **ksp** (Kotlin Symbol Processing) - Nowsze i szybsze narzędzie niż kapt
- **annotationProcessor** (w Javie) - Używane w projektach Java, interoperacyjne z Kotlinem

Adnotacje

Procesor adnotacji (ang. annotation processor) to narzędzie, które analizuje adnotacje w czasie kompilacji i może generować dodatkowy kod, walidować strukturę kodu, itp. Programista oznacza kod adnotacjami. Kompilator uruchamia procesory adnotacji. Procesor odczytuje adnotacje i np. generuje klasy źródłowe, pliki XML, itp.

Narzędzia do przetwarzania adnotacji

- **kapt -** Domyślne narzędzie do przetwarzania adnotacji w Kotlinie
- **ksp** (Kotlin Symbol Processing) - Nowsze i szybsze narzędzie niż kapt

```
dependencies {
    val room_version = "2.7.1"

    implementation("androidx.room:room-runtime:$room_version")

    // If this project uses any Kotlin source, use Kotlin Symbol Processing (KSP)
    // See Add the KSP plugin to your project
    ksp("androidx.room:room-compiler:$room_version")

    // If this project only uses Java source, use the Java annotationProcessor
    // No additional plugins are necessary
    annotationProcessor("androidx.room:room-compiler:$room_version")

    // optional - Kotlin Extensions and Coroutines support for Room
    implementation("androidx.room:room-ktx:$room_version")
```

Adnotacje

Kotlin build.gradle.kts | Groovy build.gradle

```
plugins {
    id("com.google.devtools.ksp") version "2.0.21-1.0.27" apply false
}
```

Następnie włącz KSP w pliku `build.gradle.kts` na poziomie modułu:

Kotlin build.gradle.kts | Groovy build.gradle

```
plugins {
    id("com.google.devtools.ksp")
}
```

ksp

1. Uruchamia się podczas **kompilacji Kotlin** (nie JVM).
2. Otrzymuje **symboliczne reprezentacje** klas, funkcji, adnotacji itd. – np. KSClassDeclaration, KSPropertyDeclaration.
3. Analizuje je w kodzie źródłowym.
4. Generuje pliki Kotlin (.kt) lub inne (np. .java, .xml).
5. Te pliki są dodawane do projektu i kompilowane razem z resztą kodu.

ksp

1. Uruchamia się podczas **kompilacji Kotlin** (nie JVM).
2. Otrzymuje **symboliczne reprezentacje** klas, funkcji, adnotacji itd. – np. KSClassDeclaration, KSPropertyDeclaration.
3. Analizuje je w kodzie źródłowym.
4. Generuje pliki Kotlin (.kt) lub inne (np. .java, .xml).
5. Te pliki są dodawane do projektu i kompilowane razem z resztą kodu.

KSP API:

- **SymbolProcessor** - Główna klasa analizująca kod
- **Resolver** - Umożliwia wyszukiwanie adnotacji, klas, typów
- **KSAnnotated** - Dowolny element z adnotacją
- **KSClassDeclaration** - Deklaracja klasy
- **CodeGenerator** - Służy do generowania kodu źródłowego