Java_2

Zbigniew Gontar

January 2026

1 Modelowanie danych i obiektów w Javie

1.1 Cel lekcji

Celem niniejszej lekcji jest wprowadzenie zasad poprawnego modelowania danych i obiektów w języku Java na przykładzie prostego systemu rejestrującego wykonanie procesów.

Lekcja koncentruje się na:

- definiowaniu klas i ich odpowiedzialności,
- rozróżnieniu danych, stanu i zachowania,
- świadomym użyciu niemutowalności,
- projektowaniu konstruktorów i metod dostępowych.

1.2 Opis przykładu

Rozważamy system, który rejestruje wykonanie procesów w postaci zdarzeń. Każde zdarzenie opisuje fakt wykonania określonej aktywności w czasie i przynależy do konkretnego procesu.

Na tym etapie interesuje nas wyłącznie:

- poprawne zaprojektowanie klas,
- relacje pomiędzy obiektami,
- reprezentacja danych w pamięci programu.

1.3 Klasa jako nośnik danych niemutowalnych

W języku Java bardzo częstym wzorcem jest klasa reprezentująca rekord danych, którego zawartość nie powinna ulegać zmianie po utworzeniu.

```
import java.time.Instant;

public class Zdarzenie {

    private final String identyfikatorProcesu;
    private final String nazwaAktywnosci;
    private final Instant czas;

    public Zdarzenie(String identyfikatorProcesu,
                     String nazwaAktywnosci,
                     Instant czas) {
        this.identyfikatorProcesu = identyfikatorProcesu;
        this.nazwaAktywnosci = nazwaAktywnosci;
        this.czas = czas;
    }

    public String getIdentyfikatorProcesu() {
        return identyfikatorProcesu;
    }

    public String getNazwaAktywnosci() {
        return nazwaAktywnosci;
    }

    public Instant getCzas() {
        return czas;
    }
}
```

Zastosowane decyzje projektowe:

- wszystkie pola są `private final`,
- brak metod modyfikujących stan,
- konstruktor w pełni inicjalizuje obiekt.

Tak zaprojektowana klasa:

- jest bezpieczna,
- łatwa do testowania,
- odporna na błędy wynikające z niekontrolowanych zmian stanu.

1.4 Agregacja obiektów — klasa Proces

Kolejnym krokiem jest zaprojektowanie klasy, która będzie agregowała wiele obiektów klasy `Zdarzenie`.

```
import java.util.ArrayList;
import java.util.List;

public class Proces {

    private final String identyfikator;
    private final List<Zdarzenie> zdarzenia;

    public Proces(String identyfikator) {
        this.identyfikator = identyfikator;
        this.zdarzenia = new ArrayList<>();
    }

    public void dodajZdarzenie(Zdarzenie z) {
        if (!z.getIdentyfikatorProcesu().equals(identyfikator)) {
            throw new IllegalArgumentException(
                "Zdarzenie nale y do innego procesu"
            );
        }
        zdarzenia.add(z);
    }

    public List<Zdarzenie> getZdarzenia() {
        return zdarzenia;
    }
}
```

W tym miejscu wprowadzamy istotne pojęcia Javy:

- kolekcję `List`,
- hermetyzację (brak bezpośredniego dostępu do pól),
- kontrolę poprawności danych wejściowych.

1.5 Program testowy

```
import java.time.Instant;

public class Main {

    public static void main(String[] args) {

        Proces proces = new Proces("P1");

        proces.dodajZdarzenie(
            new Zdarzenie("P1", "Start", Instant.now())
```

```
        );

        proces.dodajZdarzenie(
            new Zdarzenie("P1", "Validate", Instant.now())
        );
    }
}
```

Na tym etapie student powinien rozumieć:

- różnicę między klasą a obiektem,
- rolę konstruktora,
- dlaczego dane są enkapsulowane.

1.6 Zadania

Zadanie 1. Dodaj do klasy `Zdarzenie` metodę `toString()`.

Zadanie 2. Zabezpiecz klasę `Proces` przed dodaniem wartości `null`.

2 Kolekcje i analiza danych w Javie

2.1 Cel lekcji

Celem lekcji jest nauczenie:

- pracy z kolekcjami w Javie,
- sortowania obiektów,
- iteracji po danych,
- wydzielania logiki analitycznej.

2.2 Porządkowanie danych

W praktyce dane często nie są wprowadzane w kolejności logicznej. Java udostępnia mechanizmy sortowania kolekcji przy użyciu komparatorów.

```
import java.util.Comparator;

public class AnalizaProcesu {

    public static void uporzadkujPoCzasie(Proces proces) {
        proces.getZdarzenia().sort(
            Comparator.comparing(Zdarzenie::getCzas)
        );
    }
}
```

W tym fragmencie pojawia się:

- referencja do metody,
- interfejs funkcyjny `Comparator`,
- modyfikacja kolekcji obiektów.

2.3 Analiza sekwencji

Po uporządkowaniu danych możemy analizować relacje pomiędzy zdarzeniami.

```
public class AnalizaSekwencji {

    public static void wypiszNastepstwa(Proces proces) {

        var zdarzenia = proces.getZdarzenia();

        for (int i = 0; i < zdarzenia.size() - 1; i++) {
            String a1 = zdarzenia.get(i).getNazwaAktywnosci();
            String a2 = zdarzenia.get(i + 1).getNazwaAktywnosci();
            System.out.println(a1 + " -> " + a2);
        }
    }
}
```

To ćwiczenie uczy:

- iteracji indeksowej,
- analizy danych sekwencyjnych,
- pracy na obiektach zamiast typów prostych.

2.4 Rozdzielenie odpowiedzialności

Zauważmy, że:

- klasa `Proces` przechowuje dane,
- klasy `AnalizaProcesu` i `AnalizaSekwencji` realizują logikę.

Jest to wstęp do zasady:

> klasy domenowe nie powinny zawierać nadmiaru logiki technicznej.

2.5 Zadania

Zadanie 1. Napisz metodę obliczającą czas trwania procesu.

Zadanie 2. Zmodyfikuj analizę tak, aby obsługiwała procesy z jednym zdarzeniem.

Zadanie 3. Wyjaśnij różnicę pomiędzy:

- przechowywaniem danych,
- ich analizą,
- prezentacją wyników.

3 Interfejsy, kontrakty i walidacja

3.1 Cel lekcji

Celem niniejszej lekcji jest wprowadzenie mechanizmów języka Java, które pozwalają projektować systemy:

- rozszerzalne,
- odporne na błędy,
- jasno określające swoje założenia.

W szczególności omówione zostaną:

- interfejsy jako kontrakty programistyczne,
- pre- i postwarunki metod,
- walidacja danych wejściowych,
- rola wyjątków w sygnalizowaniu naruszeń kontraktu.

3.2 Motywacja: dlaczego same klasy nie wystarczają

Dotychczasowe klasy (`Zdarzenie`, `Proces`) poprawnie reprezentują dane i relacje pomiędzy nimi. Brakuje jednak formalnego mechanizmu określającego:

- jakie operacje są dozwolone,
- jakie warunki muszą być spełnione przed ich wykonaniem,
- co metoda gwarantuje po zakończeniu.

Takie formalne zobowiązanie nazywane jest **kontraktem**.

3.3 Interfejs jako kontrakt

W języku Java interfejs służy do definiowania kontraktu, czyli zbioru metod, które dana klasa *musi* udostępniać, nie ujawniając sposobu ich implementacji.

3.4 Interfejs Walidowalny

Zdefiniujmy interfejs, który reprezentuje obiekt zdolny do sprawdzenia własnej poprawności.

```
public interface Walidowalny {

    void waliduj();
}
```

Znaczenie interfejsu:

- narzuca obecność metody `waliduj()`,
- nie określa sposobu walidacji,
- umożliwia jednolite traktowanie różnych obiektów.

3.5 Kontrakt dla klasy Zdarzenie

Rozszerzamy klasę `Zdarzenie` o implementację interfejsu `Walidowalny`.

```
import java.time.Instant;
import java.util.Objects;

public class Zdarzenie implements Walidowalny {

    private final String identyfikatorProcesu;
    private final String nazwaAktywnosci;
    private final Instant czas;

    public Zdarzenie(String identyfikatorProcesu,
                     String nazwaAktywnosci,
                     Instant czas) {

        this.identyfikatorProcesu = identyfikatorProcesu;
        this.nazwaAktywnosci = nazwaAktywnosci;
        this.czas = czas;

        waliduj();
    }

    @Override
    public void waliduj() {

        if (identyfikatorProcesu == null || identyfikatorProcesu.isBlank()) {
            throw new IllegalArgumentException(
                "Identyfikator procesu nie mo e by  pusty"
            );
```

```
        }

        if (nazwaAktywnosci == null || nazwaAktywnosci.isBlank()) {
            throw new IllegalArgumentException(
                "Nazwa aktywno ci nie mo e by  pusta"
            );
        }

        if (czas == null) {
            throw new IllegalArgumentException(
                "Znacznik czasu nie mo e by  null"
            );
        }
    }

    public String getIdentyfikatorProcesu() {
        return identyfikatorProcesu;
    }

    public String getNazwaAktywnosci() {
        return nazwaAktywnosci;
    }

    public Instant getCzas() {
        return czas;
    }
}
```

3.6 Prewarunki i niezmienniki

W powyższym przykładzie:

- konstruktor sprawdza poprawność danych wejściowych,
- po utworzeniu obiektu jego stan jest zawsze poprawny,
- walidacja nie jest opcjonalna.

Taki obiekt spełnia **niezmiennik klasowy**:

> Każde `Zdarzenie` posiada niepusty identyfikator procesu, niepustą nazwę aktywności oraz poprawny znacznik czasu.

3.7 Walidacja na poziomie agregatu

Agregaty również mogą i powinny posiadać kontrakty.

```
import java.util.ArrayList;
import java.util.List;

public class Proces implements Walidowalny {

    private final String identyfikator;
    private final List<Zdarzenie> zdarzenia;

    public Proces(String identyfikator) {

        if (identyfikator == null || identyfikator.isBlank()) {
            throw new IllegalArgumentException(
                "Identyfikator procesu nie mo e by  pusty"
            );
        }

        this.identyfikator = identyfikator;
        this.zdarzenia = new ArrayList<>();
    }

    public void dodajZdarzenie(Zdarzenie z) {

        if (z == null) {
            throw new IllegalArgumentException(
                "Zdarzenie nie mo e by  null"
            );
        }

        if (!z.getIdentyfikatorProcesu().equals(identyfikator)) {
            throw new IllegalArgumentException(
                "Zdarzenie nale y do innego procesu"
            );
        }

        zdarzenia.add(z);
    }

    @Override
    public void waliduj() {

        if (zdarzenia.isEmpty()) {
            throw new IllegalStateException(
                "Proces nie zawiera  adnych  zdarze "
            );
        }
    }
```

```
    public List<Zdarzenie> getZdarzenia() {
        return zdarzenia;
    }
}
```

3.8 Rola wyjątków

Wyjątki w Javie:

- sygnalizują naruszenie kontraktu,
- zatrzymują wykonanie w momencie błędu,
- wymuszają reakcję programisty.

W tej lekcji celowo stosowane są:

- `IllegalArgumentException` — błędne dane wejściowe,
- `IllegalStateException` — niepoprawny stan obiektu.

3.9 Interfejs jako punkt rozszerzeń

Zastosowanie interfejsu `Walidowalny` pozwala na jednolite traktowanie obiektów:

```
public class Walidator {

    public static void walidujWszystkie(
            List<Walidowalny> obiekty) {

        for (Walidowalny o : obiekty) {
            o.waliduj();
        }
    }
}
```

To wprowadza:

- polimorfizm,
- luźne powiązanie klas,
- możliwość dalszej rozbudowy systemu.

3.10 Zadania

Zadanie 1. Zdefiniuj interfejs `Analizowalny` z metodą `analizuj()`.

Zadanie 2. Zaimplementuj ten interfejs w klasie `Proces`.

Zadanie 3. Wyjaśnij różnicę pomiędzy:

- kontraktem wyrażonym przez interfejs,
- walidacją w konstruktorze,
- obsługą wyjątków.

4 Wyjątki w Javie: projektowanie i odpowiedzialne użycie

4.1 Cel lekcji

Celem niniejszej lekcji jest zrozumienie mechanizmu wyjątków w języku Java nie jako elementu składni, lecz jako narzędzia projektowego.

Po zakończeniu lekcji student powinien:

- rozumieć hierarchię wyjątków w Javie,
- umieć projektować własne wyjątki,
- świadomie wybierać pomiędzy wyjątkami checked i unchecked,
- wiedzieć, kiedy wyjątków **nie należy** używać.

4.2 Rola wyjątków w programie

Wyjątki służą do sygnalizowania sytuacji, w których:

- naruszony został kontrakt metody,
- dalsze poprawne wykonanie nie jest możliwe,
- wystąpił stan, którego nie da się obsłużyć lokalnie.

Wyjątki **nie służą** do:

- sterowania logiką programu,
- zastępowania instrukcji warunkowych,
- obsługi typowych, oczekiwanych przypadków.

4.3 Hierarchia wyjątków w Javie

Wszystkie wyjątki w Javie dziedziczą po klasie `Throwable`. Hierarchia ta dzieli się na dwie główne gałęzie:

- `Error`,
- `Exception`.

Klasy z gałęzi `Error` reprezentują błędy środowiska wykonawczego (np. brak pamięci) i nie powinny być obsługiwane w kodzie aplikacyjnym.

Klasy z gałęzi `Exception` reprezentują błędy aplikacyjne i dzielą się na:

- wyjątki checked,
- wyjątki unchecked (`RuntimeException`).

4.4 Checked vs unchecked exceptions

Wyjątki checked

- muszą być obsłużone lub zadeklarowane w sygnaturze metody,
- reprezentują sytuacje, z których wywołujący *może się* podnieść,
- często dotyczą operacji zewnętrznych (I/O, sieć, pliki).

Wyjątki unchecked

- nie muszą być deklarowane,
- zwykle sygnalizują błędy programistyczne,
- naruszają kontrakt metody lub klasy.

4.5 Projektowanie własnych wyjątków

W dobrze zaprojektowanym systemie wyjątki:

- mają znaczenie domenowe,
- są precyzyjne,
- nie ujawniają szczegółów implementacyjnych.

4.6 Bazowy wyjątek domenowy

Zaczynamy od zdefiniowania wspólnej klasy bazowej dla wyjątków domenowych.

```
public abstract class WyjatekProcesu
        extends RuntimeException {

    public WyjatekProcesu(String message) {
        super(message);
    }
}
```

Zastosowanie wspólnej klasy bazowej:

- upraszcza obsługę wyjątków,
- umożliwia grupowanie błędów,
- ułatwia logowanie i testowanie.

4.7 Wyjątki wyspecjalizowane

Na bazie wyjątku ogólnego definiujemy wyjątki szczegółowe.

```
public class NiepoprawneZdarzenieException
        extends WyjatekProcesu {

    public NiepoprawneZdarzenieException(String message) {
        super(message);
    }
}

public class NiezgodnyProcesException
        extends WyjatekProcesu {

    public NiezgodnyProcesException(String message) {
        super(message);
    }
}
```

Każdy wyjątek:

- opisuje **konkretny** problem,
- nie jest generyczny,
- może być obsłużony selektywnie.

4.8 Zastosowanie wyjątków w kodzie

Modyfikujemy klasę `Proces` tak, aby używała wyjątków domenowych.

```
public void dodajZdarzenie(Zdarzenie z) {

    if (z == null) {
        throw new NiepoprawneZdarzenieException(
            "Zdarzenie nie mo e by  null"
        );
    }

    if (!z.getIdentyfikatorProcesu().equals(identyfikator)) {
        throw new NiezgodnyProcesException(
            "Zdarzenie nale y do innego procesu"
        );
    }

    zdarzenia.add(z);
}
```

Dzięki temu:

- komunikat błędu jest jednoznaczny,
- warstwa wywołująca może reagować selektywnie,
- kod jest czytelniejszy niż przy użyciu wyjątków generycznych.

4.9 Kiedy NIE używać wyjątków

Wyjątki nie powinny być używane do:

- sprawdzania warunków biznesowych,
- obsługi normalnych wariantów wykonania,
- iteracyjnego przeszukiwania danych.

4.10 Antywzorzec: wyjątki jako sterowanie

```
// ANTYWZORZEC
try {
    Zdarzenie z = proces.getZdarzenia().get(10);
} catch (IndexOutOfBoundsException e) {
    // brak zdarzenia
}
```

Poprawne podejście:

```
if (proces.getZdarzenia().size() > 10) {
    Zdarzenie z = proces.getZdarzenia().get(10);
}
```

4.11 Wyjątki a kontrakty

Wyjątek powinien oznaczać:

> kontrakt metody został naruszony i dalsze wykonanie w tym kontekście nie ma sensu.

Jeżeli sytuacja:

- jest spodziewana,
- może być obsłużona lokalnie,
- nie narusza niezmienników,

to wyjątek **nie jest właściwym mechanizmem**.

4.12 Zadania

Zadanie 1. Dodaj wyjątek `PustyProcesException` rzucany w przypadku walidacji procesu bez zdarzeń.

Zadanie 2. Zdecyduj, czy wyjątek ten powinien być checked czy unchecked i uzasadnij wybór.

Zadanie 3. Podaj przykład sytuacji w tym projekcie, w której użycie wyjątku byłoby błędem projektowym.

5 Testy jednostkowe i weryfikacja kontraktów

5.1 Cel lekcji

Celem niniejszej lekcji jest wprowadzenie testów jednostkowych jako integralnego elementu procesu wytwarzania oprogramowania w Javie.

Po zakończeniu lekcji student powinien:

- rozumieć rolę testów jednostkowych,
- potrafić pisać testy w frameworku JUnit,
- testować kontrakty metod i klas,
- świadomie projektować testy negatywne i graniczne.

5.2 Rola testów jednostkowych

Test jednostkowy:

- weryfikuje zachowanie pojedynczej klasy lub metody,
- jest deterministyczny i powtarzalny,
- sprawdza jeden aspekt funkcjonalności.

Testy jednostkowe **nie służą** do:

- testowania interfejsu użytkownika,
- testowania integracji z bazą danych,
- wykrywania wszystkich możliwych błędów.

5.3 JUnit jako standard testów w Javie

W kursie wykorzystywany jest framework JUnit 5. Testy są umieszczane w osobnym katalogu źródłowym i uruchamiane automatycznie przez narzędzia budujące.

Podstawowe adnotacje:

- `@Test` — oznaczenie metody testowej,
- `@BeforeEach` — przygotowanie danych,
- `@AfterEach` — sprzątanie po teście.

5.4 Pierwszy test jednostkowy

Rozpocznijmy od przetestowania poprawnego utworzenia obiektu klasy `Zdarzenie`.

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import java.time.Instant;

public class ZdarzenieTest {

    @Test
    public void poprawneZdarzeniePowinnoZostacUtworzone() {

        Zdarzenie z = new Zdarzenie(
            "P1",
            "Start",
            Instant.now()
        );

        assertNotNull(z);
```

```
        assertEquals("P1", z.getIdentyfikatorProcesu());
        assertEquals("Start", z.getNazwaAktywnosci());
    }
}
```

Test ten weryfikuje:

- poprawność konstruktora,
- inicjalizację pól,
- brak wyjątków w poprawnym przypadku.

5.5 Testowanie kontraktów

Kontrakty definiowane w poprzednich lekcjach muszą być weryfikowane testami.

5.6 Test negatywny — naruszenie kontraktu

Sprawdzamy, czy konstruktor klasy `Zdarzenie` rzuca wyjątek dla niepoprawnych danych.

```
@Test
public void utworzenieZdarzeniaBezNazwyPowinnoRzucicWyjatek() {

    assertThrows(
        IllegalArgumentException.class,
        () -> new Zdarzenie("P1", "", Instant.now())
    );
}
```

Taki test:

- potwierdza istnienie kontraktu,
- dokumentuje oczekiwane zachowanie,
- zabezpiecza przed regresją.

5.7 Testowanie wyjątków domenowych

Testujemy metodę `dodajZdarzenie` w klasie `Proces`.

```
@Test
public void dodanieZdarzeniaZInnegoProcesuPowinnoRzucicWyjatek() {

    Proces proces = new Proces("P1");
    Zdarzenie z = new Zdarzenie(
        "P2",
        "Start",
```

```
            Instant.now()
        );

        assertThrows(
            NiezgodnyProcesException.class,
            () -> proces.dodajZdarzenie(z)
        );
    }
```

Test ten:

- sprawdza poprawność walidacji,
- weryfikuje użycie właściwego typu wyjątku,
- testuje kontrakt metody.

5.8 Testy graniczne

Testy graniczne koncentrują się na przypadkach leżących na granicy poprawności.

```
@Test
public void procesZJednymZdarzeniemJestPoprawny() {

    Proces proces = new Proces("P1");
    proces.dodajZdarzenie(
        new Zdarzenie("P1", "Start", Instant.now())
    );

    assertDoesNotThrow(() -> proces.waliduj());
}



@Test
public void pustyProcesPowinienBycNiepoprawny() {

    Proces proces = new Proces("P1");

    assertThrows(
        PustyProcesException.class,
        () -> proces.waliduj()
    );
}
```

5.9 Testy negatywne a testy graniczne

- test negatywny — sprawdza zachowanie dla danych błędnych,
- test graniczny — sprawdza zachowanie na granicy poprawności.

Oba typy testów są niezbędne do weryfikacji kontraktów.

5.10 Dobre praktyki testowania

- jeden test — jedno zachowanie,
- czytelne nazwy metod testowych,
- brak logiki warunkowej w testach,
- testy muszą być szybkie i niezależne.

5.11 Testy jako dokumentacja

Dobrze napisane testy:

- dokumentują zachowanie systemu,
- opisują kontrakty w sposób formalny,
- ułatwiają refaktoryzację kodu.

5.12 Zadania

Zadanie 1. Napisz test weryfikujący, że metoda `waliduj()` w klasie `Proces` przechodzi poprawnie dla procesu z dwoma zdarzeniami.

Zadanie 2. Napisz test negatywny sprawdzający, że dodanie `null` jako zdarzenia powoduje rzucenie wyjątku.

Zadanie 3. Wyjaśnij, dlaczego testy jednostkowe nie zastępują walidacji w kodzie produkcyjnym.