Wzorce projektowe

c.d.

Singleton

- Singleton należy do grupy wzorców konstrukcyjnych
- Jego celem jest ograniczenie ilości tworzonych instancji klasy do jednej
- Wzorzec daje pewność, że istnieje tylko jedna instancja klasy i dostarcza globalnego punktu dostępu do tej instancji
- Przykłady zastosowania: klasa konfiguracyjna, jeden punkt dostępu do bazy danych, jedna kolejka wydruku dokumentów lub manager okien

Singleton

Wersja I – Chciwa inicjalizacja

```java
public class Counter {

    private static final Counter instance = new Counter();
    private int count = 0;

    private Counter() {
    }

    public static Counter getInstance() {
        return instance;
    }

    public void add() {
        this.count++;
        System.out.println(this.count);
    }
}
```

```java
public class App {

    public static void main(String[] args) {
        Counter c = Counter.getInstance();
        c.add();
        c.add();

        c = Counter.getInstance();
        c.add();
        c.add();
    }

}
```

Singleton

Wersja I – Chciwa inicjalizacja

Charakterystyka

- Prywatny konstruktor
- Metoda statyczna umożliwiający spełnienie warunku globalnego punktu dostępu
- Inicjalizacja klasy następuje już w momencie jej załadowania – chciwa inicjalizacja

Singleton

Wersja II - Leniwa inicjalizacja

```java
public class Counter {

    private static Counter instance;
    private int count = 0;

    private Counter() {
    }

    public static Counter getInstance() {
        if (instance == null) {
            instance = new Counter();
        }
        return instance;
    }

    public void add() {
        this.count++;
        System.out.println(this.count);
    }
}
```

```java
public class App {

    public static void main(String[] args) {
        Counter c = Counter.getInstance();
        c.add();
        c.add();

        c = Counter.getInstance();
        c.add();
        c.add();
    }

}
```

Singleton

Wersja II – Leniwa inicjalizacja

Charakterystyka

- Odwlekamy w czasie moment inicjalizacji obiektu – leniwa inicjalizacja
- Oszczędzamy zasoby, jeżeli nie zajdzie potrzeba jej utworzenia

Singleton

Wersja III - Refleksje

```java
public class Counter2 {

    private static Counter2 instance;
    private int count = 0;

    private Counter2() {
        if (instance != null) {
            throw new IllegalStateException("Error");
        }
    }

    public static Counter2 getInstance() {
        if (instance == null) {
            instance = new Counter2();
        }
        return instance;
    }

    public void add() {
        this.count++;
        System.out.println(this.count);
    }
}
```

```java
Counter2 c = Counter2.getInstance();
c.add();
c.add();

Constructor<Counter2> constructor = Counter2.class.getDeclaredConstructor();

constructor.setAccessible(true);

c = constructor.newInstance();
c.add();
c.add();
```

Singleton

Wersja III - Refleksje

Charakterystyka

- Podczas wywołania wykorzystujemy refleksje – narzędzie umożliwiające modyfikować właściwości kodu programu w trakcie jego działania
- Próba utworzenia prywatnego obiektu zakończy się wyjątkiem

Singleton

Wersja IV - Wielowątkowość

```java
public class Counter {

    private static volatile Counter instance;
    private int count = 0;

    private Counter() {
        if (instance != null) {
            throw new IllegalStateException("Error");
        }
    }

    public static Counter getInstance() {
        if (instance == null) {
            synchronized (Counter.class) {
                if (instance == null) {
                    instance = new Counter();
                }
            }
        }
        return instance;
    }

    public void add() {
        this.count++;
        System.out.println(this.count);
    }
}
```

```java
public class App {

    public static void main(String[] args) {
        Counter c = Counter.getInstance();
        c.add();
        c.add();

        c = Counter.getInstance();
        c.add();
        c.add();
    }

}
```

Singleton

Wersja IV - Wielowątkowość

Charakterystyka

- Problem wielowątkowości polega na tym, że w tym samym momencie oba wątku mogą sprawdzić czy instancja singletonu jest null i ją stworzyć
- Metoda synchronized tworzy sekcje krytyczną, która ustawia w kolejce wątki chcące wywołać metodę getInstance
- Metoda synchronized jest za warunkiem *instance==null* – wywołujemy ją tylko wtedy gdy instancja singletonu nie została jeszcze utworzona – forma optymalizacji
- Blokada z podwójnym zatwierdzeniem (Double-checked locking optimization) – podwójny warunek na istnienie instancji singletonu zabezpiecza nas w pełni
- Typ volatile gwarantuje nam, że zmienna jest przechowywana w pamięci współdzielonej zamiast w cache
- Problem wielowątkowości pojawia się tylko przy leniwej inicjalizacji

Singleton

Wersja V - Static holder pattern

```java
public class Counter {

    private int count = 0;

    private Counter() {
        if (MyClass.instance != null) {
            throw new IllegalStateException("Error");
        }
    }

    public static Counter getInstance() {
        return MyClass.instance;
    }

    public void add() {
        this.count++;
        System.out.println(this.count);
    }

    private static class MyClass {

        private static final Counter instance = new Counter();
    }
}
```

```java
public class App {

    public static void main(String[] args) {
        Counter c = Counter.getInstance();
        c.add();
        c.add();

        c = Counter.getInstance();
        c.add();
        c.add();
    }

}
```

Singleton

Wersja V - Static holder pattern

Charakterystyka

- Singleton Static holder pattern – w prosty sposób łączy leniwą inicjalizację z optymalnym rozwiązaniem wielowątkowości
- Instancja singletonu powstaje w momencie wywołania statycznej metody zagnieżdżonej klasy MyClass
- Problem zarządzania instancjami zrzucamy na wirtualną maszynę javy

Singleton

Wersja VI - Enum

```
public enum Counter {
    instance;

    private int count = 0;

    public void add() {
        this.count++;
        System.out.println(this.count);
    }
}
```

```
public class App {

    public static void main(String[] args) throws Exception {
        Counter c = Counter.instance;
        c.add();
        c.add();

        c = Counter.instance;
        c.add();
        c.add();
    }
}
```

Singleton

Wersja VI - Enum

Charakterystyka

- Problem zarządzania instancjami zrzucamy na wirtualną maszynę javy – podobnie jak w podejściu static holder pattern
- Enum z definicji może mieć tylko jedną instancję
- Jest odporny na refleksję i klonowanie
- Brak możliwości rozszerzania klasy (dziedziczenia)
- Brak leniwej inicjalizacji

Singleton

Antywzorzec

- Problem z testowaniem – w singletonie przechowujemy aktualny stan aplikacji w ciągu całego jej życia, należy pamiętać by go czyścić podczas uruchamiania kolejnych testów
- Łamie zasadę pojedynczej odpowiedzialności (single responsibility principle) – z definicji zajmuje się wykonywaniem funkcji biznesowej oraz zarządzaniem instancją
- Łamie zasadę otwarte-zamknięte (Open/Closed principle) – nie można go rozbudować, chyba że połączymy go z fabryką
- Jest obiektowym zamiennikiem zmiennej globalnej

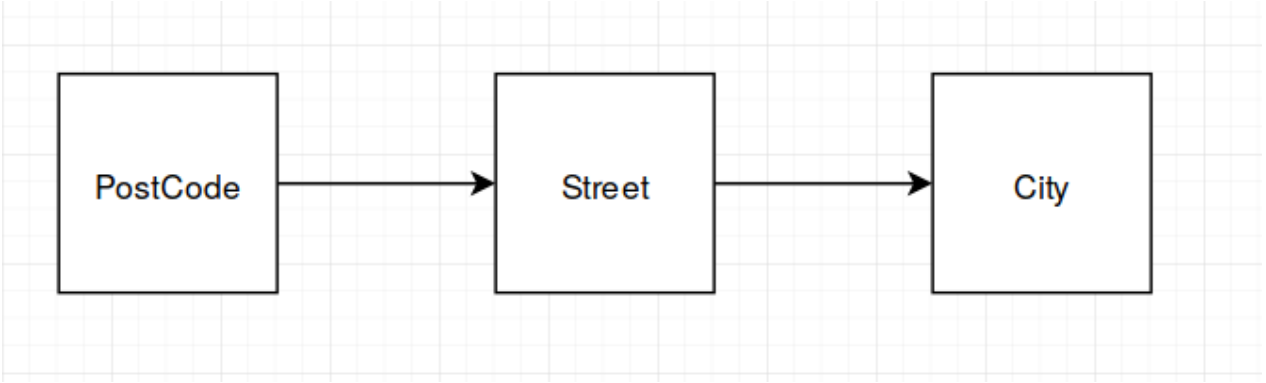
Chain of Responsibility

Łańcuch odpowiedzialności

- Łańcuch odpowiedzialności umożliwia pewnej liczbie klas podjęcie próby obsłużenia żądania podczas gdy żadna z nich nic nie wie o możliwościach innych
- Rezygnujemy z większych powiązań między klasami – jedynym powiązaniem jest przekazywane żądanie
- Żądanie jest tak długo przekazywane, aż natrafimy na klasę, która potrafi je obsłużyć
- Brak gwarancji, że każde żądanie będzie obsłużone
- Skutecznie zastępuje serię if’ów lub rozbudowanego switch’a
- W łatwy sposób można dodać kolejną obsługę żądania

Chain of Responsibility

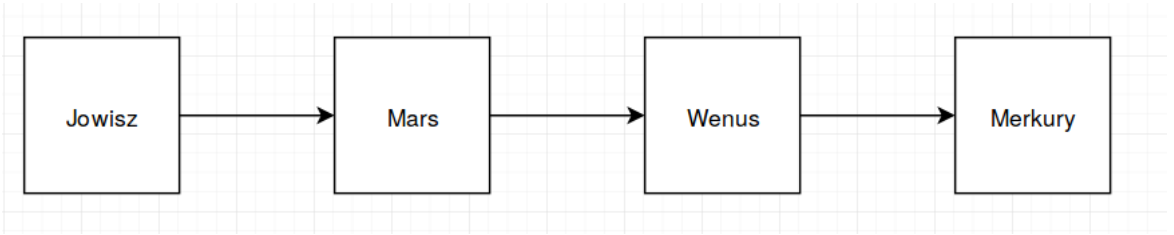
Łańcuch odpowiedzialności



- Łańcuch jest uporządkowany od klas obsługujących bardziej specyficzne rozwiązania do bardziej ogólnych

Chain of Responsibility

Łańcuch odpowiedzialności




```java
public interface Chain {

    public void setNext(Chain next);

    public String process(PlanetTypes type);
}
```

```java
public class CheckJowisz implements Chain {

    private Chain next;

    @Override
    public void setNext(Chain next) {
        this.next = next;
    }

    @Override
    public String process(PlanetTypes type) {
        if (type == PlanetTypes.JOWISZ) {
            return "Planeta Jowisz";
        } else {
            return this.next.process(type);
        }
    }
}
```

```java
public class ChainOfResponsibilityExample {

    public static void main(String[] args) {
        Chain jowisz = new CheckJowisz();
        Chain mars = new CheckMars();
        Chain wenus = new CheckWenus();
        Chain merkury = new CheckMerkury();

        jowisz.setNext(mars);
        mars.setNext(wenus);
        wenus.setNext(merkury);

        System.out.println(jowisz.process(PlanetTypes.MARS));
        System.out.println(jowisz.process(PlanetTypes.JOWISZ));

    }

}
```

Chain of Responsibility

Łańcuch odpowiedzialności