

Wzorce projektowe na przykładzie C++

Piotr Kowalczyk, Szymon Tonderys

Agenda

- Czym są wzorce projektowe
- Praktyczne przykłady użycia wzorców projektowych (ćwiczenia)
- Kiedy używać wzorców projektowych (podsumowanie)

Dobre praktyki

- S
- O
- L
- I
- D

Dobre praktyki

- **S**ingle responsibility principle
- **O**pen closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

Dobre praktyki

Przykładowe antywzorce w programowaniu

- *The Blob*
- *Golden Hammer*
- *Spaghetti Code*
- *Cut-and-Paste Programming*

Więcej informacji dla zainteresowanych tematem:

- <http://sourcemaking.com/antipatterns>

Czym są wzorce projektowe?

„Opis komunikujących się obiektów i klas, które przerabia się w celu rozwiązania ogólnego danego problemu przy dokładnie określonym kontekście.”

„Gang of Four” („Design Patterns”)



Po co stosujemy wzorce projektowe?

- Poprawa komunikacji
dzięki wspólnej (i spójnej) terminologii
- Poprawa jakości
większa odporność na błędy dzięki sprawdzonym rozwiązaniom
- Poprawa produktywności
unikanie wymyślania koła na nowo

Jak klasyfikujemy wzorce projektowe?

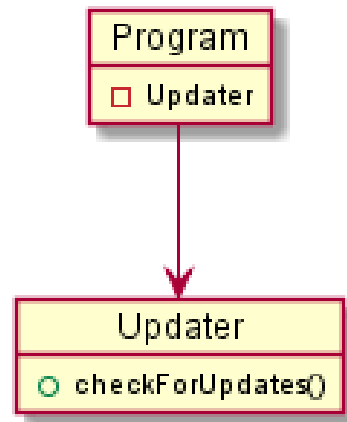
		Klasyfikacja według zastosowania		
		Kreacyjne	Strukturalne	Czynnościowe
Klasyfikacje według struktury wewnętrznej	Klasa	Factory Method	Adapter (class)	Interpreter Template Method
	Obiekt	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Przykład 01

Przykład 01

problem:

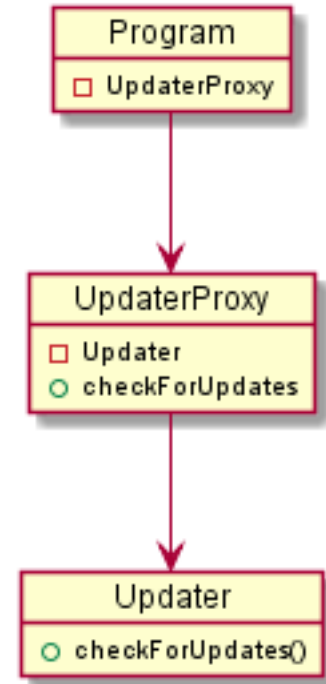
- Klient (Program) musi utworzyć obiekt `Updater`.
- `Updater` może wykonywać kosztowne operacje podczas inicjalizacji a nawet rzucać wyjątek.
- Nie możemy zmienić implementacji `Updatera`.



Przykład 01

rozwiązanie:

- Tworzymy pośrednika (`UpdaterProxy`) który zarządza połączeniem obiektu typu `Updater` z serwerem
- Pośrednik udostępnia ten sam interfejs co `Updater` dzięki czemu klient (`Program`) używa pośrednika tak samo jak instancji klasy `Updater`



Pośrednik (Proxy)



Pośrednik (Proxy)

podsumowanie:

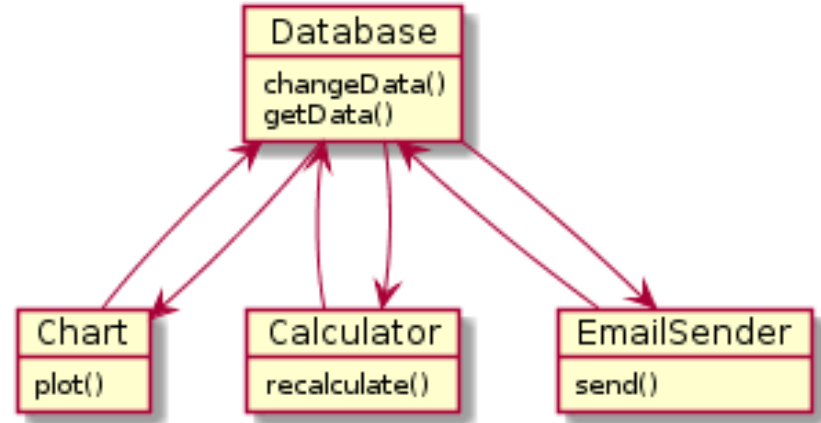
- Podstawowe cechy:
 - udostępnia ten sam interfejs co rzeczywista klasa jednak zmienia jej zachowanie
 - zawiera wskaźnik/referencję do rzeczywistego obiektu
 - kontroluje dostęp do rzeczywistego obiektu
 - zarządza czasem życia rzeczywistego obiektu
- Przykładowe zastosowania:
 - zarządzanie pamięcią (smart pointer)
 - tworzenie obiektów na żądanie (virtual proxy)
 - reprezentacja zdalnych obiektów (remote proxy)
 - kontrola dostępu do obiektu (protection proxy)

Przykład 02

Przykład 02

problem:

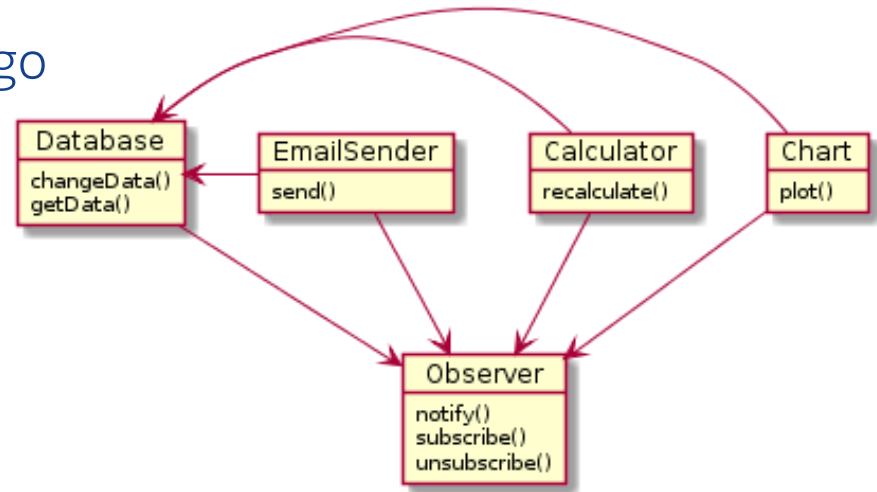
- Chcemy zamodelować interakcje pomiędzy obiektami należącymi do różnych warstw abstrakcji



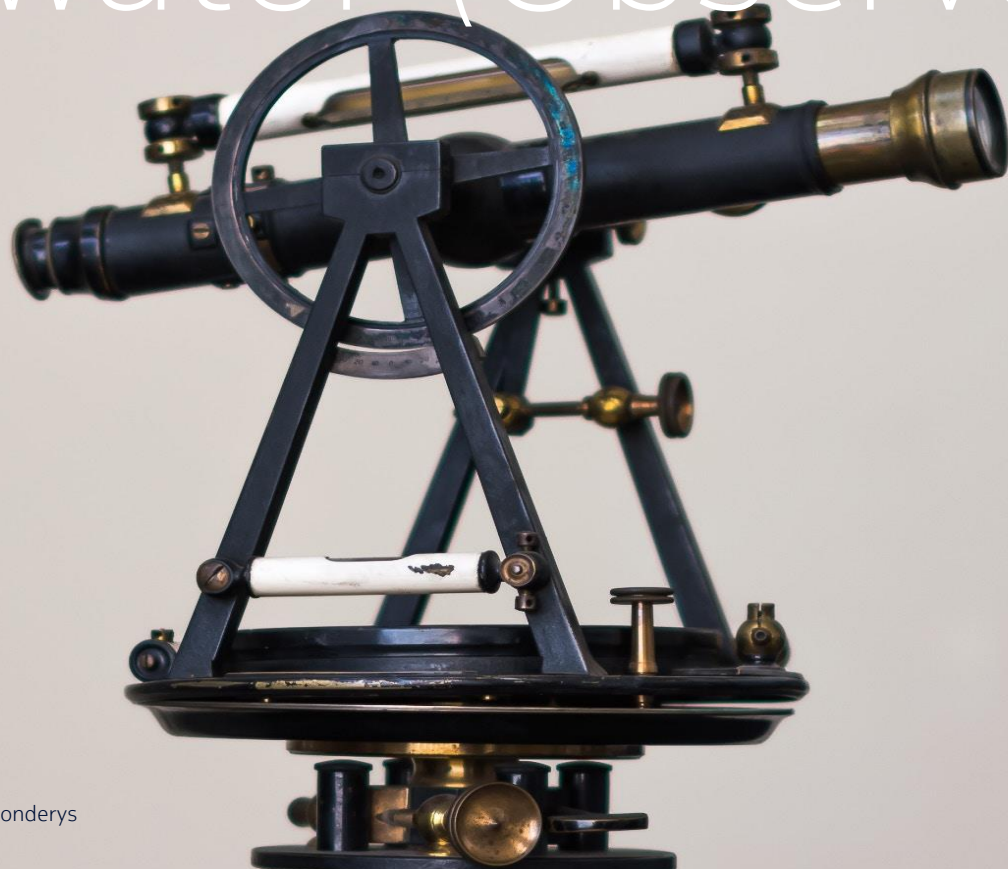
Przykład 02

rozwiązanie:

- Wszystkie klasy zależą od generycznego mechanizmu powiadamiania zaimplementowanego w Obserwatorze
- Obiekty mogą rejestrować się na wybrane zdarzenie dostarczając Obserwatorowi sposób jego obsłużenia



Obserwator (Observer)



Obserwator (Observer)

rozwiązanie:

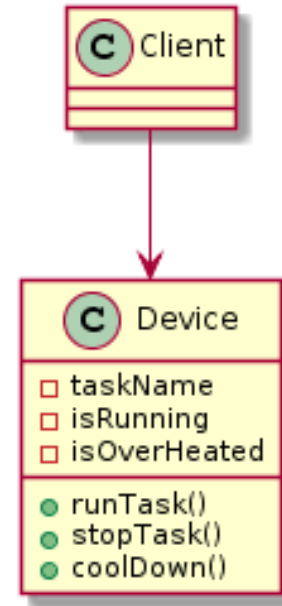
- Podstawowe cechy:
 - realizuje zasadę otwarte-zamknięte (Open-Closed Principle)
 - daje możliwość rejestrowania / wyrejestrowywania się na dane wydarzenie
 - wysyła powiadomienie jeśli obiekt obserwowany wykona określoną akcję
 - definiuje sposób obsługi zdarzenia (np. przy pomocy `std::function<void ()>`)
- Przykłady zastosowania:
 - notyfikacja dowolnie dużej liczby obiektów niepowiązanych ze sobą logicznie
 - konieczność wiązania obiektów (Obserwator - Obserwowany) w trakcie działania programu
- Uwaga: istnieją gotowe implementacje mechanizmu obserwatora (np. `boost::signals2`)

Przykład 03

Przykład 03

problem:

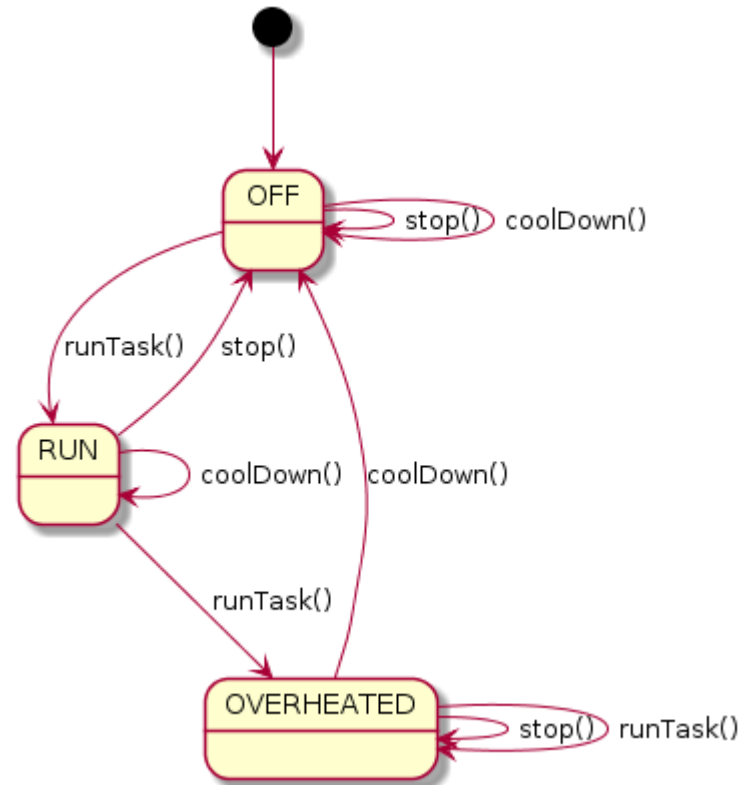
- Chcemy zamodelować maszynę, która będzie reagować na zdarzenia w sposób zależny od jej stanu
- Device może uruchomić, bądź zatrzymać zadanie (task).
- próba uruchomienia kolejnego zadania kończy się przegrzaniem (isOverheated)
- po schłodzeniu (coolDown) urządzenie może ponownie uruchomić zadanie (task)



Przykład 03

problem:

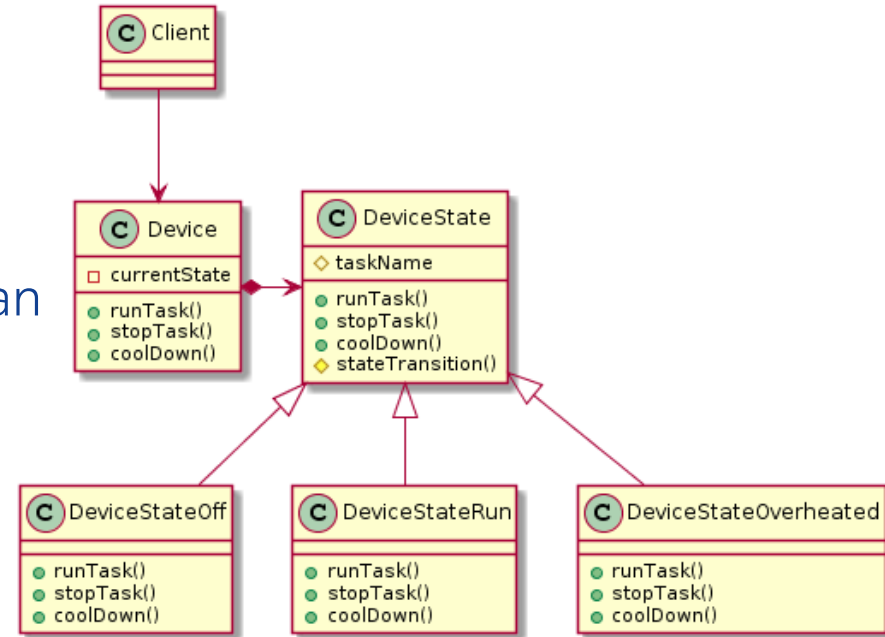
- Chcemy zamodelować maszynę, która będzie reagować na zdarzenia w sposób zależny od jej stanu
- Device może uruchomić, bądź zatrzymać zadanie (task).
- próba uruchomienia kolejnego zadania kończy się przegrzaniem (isOverheated)
- po schłodzeniu (coolDown) urządzenie może ponownie uruchomić zadanie (task)



Przykład 03

rozwiązanie:

- każdy stan reprezentujemy przez polimorficzne klasy implementujące interfejs (DeviceState)
- klasa Device przechowuje aktualny stan (currentState) oraz deleguje do niego obsługę wszystkich zdarzeń.
- każdy stan ma możliwość zmiany aktualnego stanu na inny



Stan (State)



Stan (State)

podsumowanie:

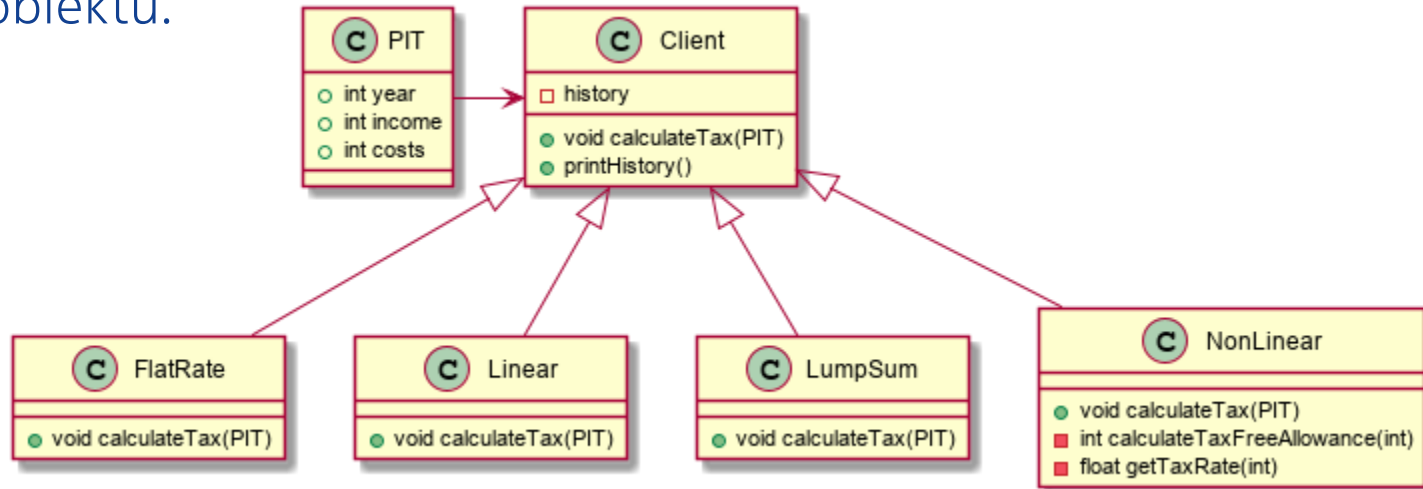
- Podstawowe cechy:
 - enkapsuluje zachowania każdego stanu w osobnej klasie
 - realizuje zasady otwarte - zamknięte oraz jednej odpowiedzialności
 - uwalnia od skomplikowanej logiki w metodach
 - stan obiektu jest zawsze spójny, ponieważ jest reprezentowany przez jednego membera
 - klasa zarządzająca stanem deleguje wywołania metod do aktualnego stanu
- Wady:
 - współdzielenie memberów pomiędzy stanami może okazać się kłopotliwe.
 - istnieje ryzyko duplikacji implementacji pewnych metod
 - istnieje wiele możliwości implementacji wzorca.

Przykład 04

Przykład 04

problem:

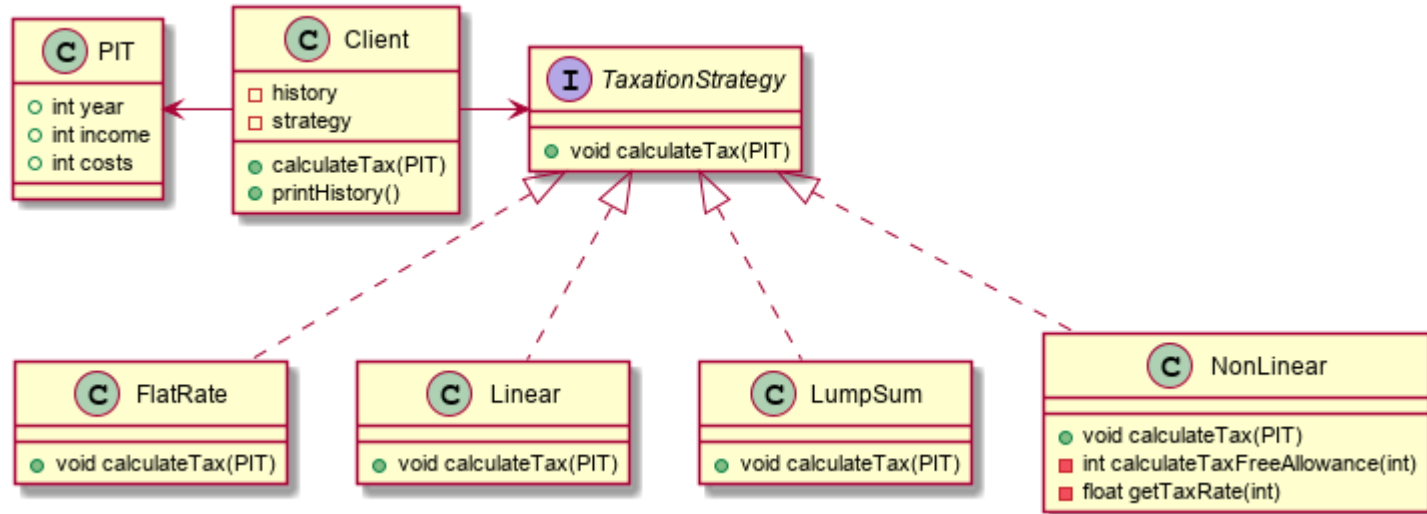
- Obiekt dziedziczący po klasie `Client` potrafi wyliczyć sobie podatek
- Sposób liczenia podatku jest zaimplementowany w klasie pochodnej
- Jeśli dany klient zechce zmienić sposób liczenia podatku trzeba utworzyć nową instancję obiektu.



Przykład 04

rozwiązanie:

- Klasa `Client` posiada obiekt abstrakcyjnego typu `TaxationStrategy`, którą można podmieniać w czasie działania programu
- Zmiana sposobu liczenia podatku nie modyfikuje `Clienta`, ani jego historii



Strategia (Strategy)



Strategia (Strategy)

Podsumowanie

- Podstawowe cechy:
 - klasa klienta nie ma pojęcia o istnieniu podrzędnych klas implementacyjnych
 - klient jest zwolniony z podejmowania decyzji o wyborze algorytmu
 - realizuje zasadę otwarte - zamknięte (Open/Closed Principle)
- Przykłady użycia:
 - inteligentne dobieranie algorytmu w zależności od rodzaju danych
 - ułatwia mockowanie dobrze zdefiniowanych algorytmów

Wzorce projektowe na przykładzie C++

Podsumowanie

Czym są wzorce projektowe?

„Opis komunikujących się obiektów i klas, które przerabia się w celu rozwiązania ogólnego danego problemu przy dokładnie określonym kontekście.”

„Gang of Four” („Design Patterns”)



Po co stosujemy wzorce projektowe?

- Poprawa komunikacji
dzięki wspólnej (i spójnej) terminologii
- Poprawa jakości
większa odporność na błędy dzięki sprawdzonym rozwiązaniom
- Poprawa produktywności
unikanie wymyślania koła na nowo

Kiedy stosujemy wzorce projektowe?

- Nie wymyślamy koła na nowo
- Nie używamy wzorców tylko dlatego, że je znamy
- Nie nadużywamy wzorców

Kiedy nie stosujemy wzorców projektowych?

- Nie
- Nie
- Nie

```
package com.seriouscompany.business.java.fizzbuzz.packagemakingpackage.implementation;

public final class LoopInitializer {

    private final int nStoredLoopFinalValue;

    public LoopInitializer(final int nLoopFinalValue) {
        super();
        this.nStoredLoopFinalValue = nLoopFinalValue;
    }

    public int getLoopInitializationPoint() {
        return this.nStoredLoopFinalValue;
    }
}

package com.seriouscompany.business.java.fizzbuzz.packagemakingpackage.implementation;

public final class FizzBuzzOutputGenerationStrategy implements OutputGenerationStrategy {

    private final DataPrinter printer;
    private final ISevenlyDivisibleStrategy strategy;

    public FizzBuzzOutputGenerationStrategy(final ISevenlyDivisibleStrategy strategy,
        final DataPrinter printer) {
        super();
        this.strategy = strategy;
        this.printer = printer;
    }

    @Override
    public DataPrinter getPrinter() {
        return this.printer;
    }

    @Override
    public ISevenlyDivisibleStrategy getStrategy() {
        return this.strategy;
    }
}

package com.seriouscompany.business.java.fizzbuzz.packagemakingpackage.implementation;

private final BuzzStringTurnerFactory _buzzStringTurnerFactory;
@Required
public BuzzStringPrinter(final BuzzStringTurnerFactory _buzzStringTurnerFactory,
    final SystemOutFizzBuzzOutputGenerationStrategy _outputStrategyFactory) {
    super();
    this._buzzStringTurnerFactory = _buzzStringTurnerFactory;
    this._outputStrategyFactory = _outputStrategyFactory;
}

public void print() {
    final StringStringTurner _aBuzzStringTurner = this._buzzStringTurnerFactory
        .createStringStringTurner();
    final FizzBuzzOutputStrategyOfFizzBuzzOutputGenerationStrategyAdapter _aOutputAdapter =
        new FizzBuzzOutputStrategyOfFizzBuzzOutputGenerationStrategyAdapter(
            this._outputStrategyFactory.createOutputStrategy());
    _aOutputAdapter.output(_aBuzzStringTurner.getReturnString());
}

@Override
public void printValue(final Object value) {
    this._outputStrategyFactory.createOutputStrategy().printValue(value);
}

package com.seriouscompany.business.java.fizzbuzz.packagemakingpackage.implementation;

public class IntegerDivider {

    private final FirstIsSmallerThanSecondDoubleComparator firstIsSmallerThanSecondDoubleComparator;
    private final FirstIsLargerThanSecondDoubleComparator firstIsLargerThanSecondDoubleComparator;

    @Required
    public IntegerDivider(final FirstIsLargerThanSecondDoubleComparator firstIsLargerThanSecondDoubleComparator,
        final FirstIsSmallerThanSecondDoubleComparator firstIsSmallerThanSecondDoubleComparator) {
        super();
        this.firstIsLargerThanSecondDoubleComparator = firstIsLargerThanSecondDoubleComparator;
        this.firstIsSmallerThanSecondDoubleComparator = firstIsSmallerThanSecondDoubleComparator;
    }
}

package com.seriouscompany.business.java.fizzbuzz.packagemakingpackage.implementation;

public class LoopInitializer {

    public static final String BL_0_ATTEMPT_TIMES_WIDE_TO_DIVIDE = "0";
    public static final String BLUZZ = "buzz";
    public static final String COM_SERIOUSCOMPAN_BUSINESS_J = "com.seriouscompany.business.java.fizzbuzz.packagemakingpackage.implementation";
    public static final String FIZZ = "fizz";
    public static final String LINE_SEPARATOR = "line.separator";
    public static final String LOOP_COMPONENT_FACTORY = "loop.component.factory";
    public static final String PRINTING_POINT = "printing.point";
    public static final String STANDARD_FIZZ_BUZZ = "standard.fizz.buzz";
    public static final String THE_INTEGER_COULD_NOT_BE_COM = "the integer could not be compared";
    public static final int RESULT_FIZZ_BUZZ_UPPER_LIMIT_90 = 90;
    public static final int INTEGER_DIVIDE_ZERO_VALUE = 0;
    public static final int INTEGER_DIVIDE_ZERO_VALUE = 0;
    public static final int LOOP_INC_VALUE = 1;
    public static final int LOOP_INIT_VALUE = 1;
}

package com.seriouscompany.business.java.fizzbuzz.packagemakingpackage.implementation;

public class Constants {

    public static final String BL_0_ATTEMPT_TIMES_WIDE_TO_DIVIDE = "0";
    public static final String BLUZZ = "buzz";
    public static final String COM_SERIOUSCOMPAN_BUSINESS_J = "com.seriouscompany.business.java.fizzbuzz.packagemakingpackage.implementation";
    public static final String FIZZ = "fizz";
    public static final String LINE_SEPARATOR = "line.separator";
    public static final String LOOP_COMPONENT_FACTORY = "loop.component.factory";
    public static final String PRINTING_POINT = "printing.point";
    public static final String STANDARD_FIZZ_BUZZ = "standard.fizz.buzz";
    public static final String THE_INTEGER_COULD_NOT_BE_COM = "the integer could not be compared";
    public static final int RESULT_FIZZ_BUZZ_UPPER_LIMIT_90 = 90;
    public static final int INTEGER_DIVIDE_ZERO_VALUE = 0;
    public static final int INTEGER_DIVIDE_ZERO_VALUE = 0;
    public static final int LOOP_INC_VALUE = 1;
    public static final int LOOP_INIT_VALUE = 1;
}

package com.seriouscompany.business.java.fizzbuzz.packagemakingpackage.implementation;

public class BuzzStringTurnerFactory implements StringStringTurnerFactory {

    private final BuzzStringTurner _fizzStringTurner;

    @Required
    public BuzzStringTurnerFactory(final BuzzStringTurner _fizzStringTurner) {
        super();
        this._fizzStringTurner = _fizzStringTurner;
    }

    @Override
    public StringStringTurner createStringStringTurner() {
        return this._fizzStringTurner;
    }
}

package com.seriouscompany.business.java.fizzbuzz.packagemakingpackage.implementation;

public class BuzzStringTurnerFactory implements StringStringTurnerFactory {

    private final BuzzStringTurner _aBuzzStringTurner;

    @Required
    public BuzzStringTurnerFactory(final BuzzStringTurner _aBuzzStringTurner) {
        super();
        this._aBuzzStringTurner = _aBuzzStringTurner;
    }

    @Override
    public StringStringTurner createStringStringTurner() {
        return this._aBuzzStringTurner;
    }
}
```

damn



Kiedy stosujemy wzorców projektowych?

Jeśli zidentyfikujemy problem, którego rozwiązaniem jest dany wzorzec

A wide-angle landscape photograph of a mountain valley at dusk or dawn. The foreground is a lush green meadow with a path of tall grass leading towards the center. In the middle ground, there are rolling green hills with small, dark wooden huts scattered across them. The background features jagged, rocky mountain peaks under a sky filled with soft, pink and orange clouds. The word "NOKIA" is superimposed in the center of the image in a bold, white, sans-serif font.

NOKIA