

---

# DOBRE PRAKTYKI PROGRAMOWANIA OBIEKTOWEGO NA PRZYKŁADZIE JĘZYKA JAVA

CLEAN CODE

---

## KOD NISKIEJ JAKOŚCI - KOSZTY

- ▶ Niewłaściwa / niepełna realizacja założonych celów biznesowych
- ▶ Rosnące koszty utrzymania i rozwoju
- ▶ Duża szansa na wystąpienie błędów
- ▶ Malejąca produktywność

## KOD NISKIEJ JAKOŚCI - PRZYCZYNY

- ▶ Brak podziału odpowiedzialności, odpowiedniej separacji i modularności
- ▶ Nieprzestrzeganie dobrych praktyk
- ▶ Błędy popełniane na poziomie analizy, projektowania i implementacji
- ▶ Niepoprawne zarządzanie zasobami (czas, ludzie, pieniądze)
- ▶ Złożoność rozwiązywanych problemów
- ▶ Brak testów

# KOD NISKIEJ JAKOŚCI - OZNAKI

- ▶ **Skostniałość** - system jest trudny w modyfikacji, zmiany wymagają kolejnych zmian
- ▶ **Delikatność** - zmiana w jednym miejscu systemu powoduje, że nie działa coś w innym miejscu
- ▶ **Nieprzenośność** - trudno podzielić system na fragmenty, które mogą być wielokrotnie używane
- ▶ **Lepkość** - łatwiej jest zrobić „hacka” i obejść fragment rozwiązania niż go zmodyfikować
- ▶ **Złożoność** - jest dużo skomplikowanego, często niepotrzebnego kodu
- ▶ **Powtórzenia** - kod wygląda jak pisany metodą ctrl+C / ctrl+V
- ▶ **Nieprzejrzystość** - projekt jest tak zawiły, że nikt nie wie o co w nim chodzi

# CZYSTY KOD

- ▶ I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.

**Bjarne Stroustrup, inventor of C++ and author of *The C++ Programming Language***

# CZYSTY KOD

- ▶ Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.

**Grady Booch**, author of *Object Oriented Analysis and Design with Applications*

# CZYSTY KOD

- ▶ Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.

**“Big” Dave Thomas, founder of OTI, godfather of the Eclipse strategy**



# ZASADY TWORZENIA „CZYSTEGO KODU”

- ▶ Dotyczą m.in. takich aspektów jak:
  - ▶ Nazewnictwo
  - ▶ Formatowanie
  - ▶ Stosowanie komentarzy
  - ▶ Podział odpowiedzialności
  - ▶ Testowanie
  - ▶ Wzorce projektowe
  - ▶ ...

# NAZEWNICTWO

- ▶ Nazwa (identyfikator) zmiennej, metody, klasy itd. powinna
  - ▶ wyjaśniać cel istnienia, pełnioną funkcję, sposób użycia, intencje autora
  - ▶ dać się wymówić
  - ▶ dać się wyszukać
  - ▶ być spójna
  - ▶ być zrozumiała
- ▶ Stosowanie przemyślanych nazw ułatwia czytanie i zmienianie kodu

```
int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```

# NAZEWNICTWO

- ▶ Nazwa klasy powinna być rzeczownikiem składającym się z jednego lub więcej członów np. Customer, Account, DataParser
- ▶ Nazwa metody powinna być czasownikiem składającym się z jednego lub więcej członów np. save(), deposit(int funds); Metody dostępowe powinny być oznaczone odpowiednim przedrostkiem set, get, is
- ▶ Nazwy zmiennych powinny być rzeczownikiem składającym się z jednego lub więcej członów

# NAZEWNICTWO - ANTYWZORCE

- ▶ Nazwy niezgodne z rzeczywistością
- ▶ Dodawanie liczby do nazwy
- ▶ Przekręcanie nazw
- ▶ Nazwy jednoliterowe
- ▶ Nieznane skróty
- ▶ Wymyślne nazwy

## ZŁE NAZEWNICTWO

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList) {  
        if (x[0] == 4) {  
            list1.add(x);  
        }  
    }  
    return list1;  
}
```

## DOBRE NAZEWNICTWO

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new ArrayList<int[]>();  
    for (int[] cell : gameBoard) {  
        if (cell[STATUS_VALUE] == FLAGGED) {  
            flaggedCells.add(cell);  
        }  
    }  
    return flaggedCells;  
}
```

# ZŁE NAZEWNICTWO

```
class DtaRcrd102 {  
    private final String pszqint = "102";  
    private Date genymdhms;  
    private Date modymdhms;  
};
```

# DOBRE NAZEWNICTWO

```
class Customer {  
    private final String RECORD_ID = "102";  
    private Date generationTimestamp;  
    private Date modificationTimestamp;  
}
```



# FORMATOWANIE KODU

- ▶ Jest obowiązkowe (znacząco poprawia czytelność i komunikację)
- ▶ Powinno być spójne na poziomie całego projektu (style guide)
- ▶ Dotyczy między innymi
  - ▶ Stosowania wcięć i separatorów w postaci pustej linii
  - ▶ Odpowiedniego łamania instrukcji
  - ▶ Długości linii
  - ▶ Kolejności deklaracji składników klasy
  - ▶ Konwencji nazewniczych

# KOMENTARZE

- ▶ Komentarze powinny
  - ▶ Być stosowane w ostateczności - tworzymy samo opisujący się kod (nie dotyczy dokumentacji publicznego API)
  - ▶ Nieść konkretną i wartościową informację
  - ▶ Być aktualne i spójne
- ▶ Szczególnym przypadkiem użycia komentarzy są informacje prawne, autorskie. Powinny one być jak najbardziej zwięzłe

# FUNKCJE / METODY

- ▶ Poprawna funkcja / metoda powinna
  - ▶ Być krótka (kilka instrukcji)
  - ▶ Realizować jedno zadanie (zgodne z nadaną nazwą)
  - ▶ Zachowywać jednolity poziom abstrakcji
  - ▶ Przyjmować niewiele argumentów (najlepiej wcale)
  - ▶ Nie powodować efektów ubocznych
  - ▶ Przyjmować / zwracać wartości null

# KLASY

- ▶ Prawidłowe klasy powinny
  - ▶ Być małe i skupiać się na realizacji jednego zadania
  - ▶ Zachowywać się jak czarne skrzynki - powinny ukrywać dane i dostarczać operacje, które pozwalają na nich bezpiecznie operować (enkapsulacja i ukrywanie danych)
  - ▶ Być konfigurowane zewnętrznymi zależnościami

# THE LAW OF DEMETER

- ▶ Obiekty nie powinny udostępniać ich wewnętrznej struktury poprzez metody dostępowe (łańcuszki wywołań)
- ▶ Inaczej metoda *f* klasy *C* powinna wołać tylko metody
  - ▶ Klasy *C*
  - ▶ Obiektów stworzonych przez metodę *f*
  - ▶ Obiektów przekazanych do metody *f* jako argument
  - ▶ Obiektów będących zmiennymi instancyjnymi klasy *C*

## OBSŁUGA BŁĘDÓW / SYTUACJI WYJĄTKOWYCH

- ▶ Preferujemy wyjątki zamiast kodów błędu
- ▶ Tam gdzie to możliwe stosujemy wyjątki typu RuntimeException
- ▶ Klasa wyjątku i komunikat błędu powinny dostarczać pełną informację o tym co się stało
- ▶ Poziom abstrakcji wyjątku powinien być dostosowany do warstwy aplikacji

# TWORZENIE TESTOWALNEGO KODU

- ▶ Stosowanie dobrych praktyk związanych z programowaniem obiektowym
- ▶ Przykłady
  - ▶ Programowanie przez interfejsy
  - ▶ SOLID principles
  - ▶ Stosowanie hermetyzacji
  - ▶ Unikanie duplikacji
  - ▶ Wzorce projektowe
  - ▶ ...

# FRAMEWORK xUNIT

- ▶ Formalnie napisany przez Kenta Becka dla języka SmallTalk, następnie przepisany do Javy i innych języków
- ▶ Odpowiedzialny za:
  - ▶ Przygotowanie testów
  - ▶ Wykonywanie testów
  - ▶ Posprzątanie po testach
  - ▶ Raportowanie o błędach i statystykach wykonania



## KLASY TESTÓW

- ▶ Zwykłe klasy Javy (POJO) posiadające
  - ▶ Jedną lub więcej metod testowych
  - ▶ Zero lub więcej metod inicjujących/sprzątających

# METODY TESTÓW

- ▶ Publiczne
- ▶ Bezargumentowe
- ▶ Oznaczone adnotacją @Test
- ▶ Nie zwracają wyniku (void)
- ▶ Niestatyczne

# KLASA TESTU

```
import org.junit.Assert;
import org.junit.Ignore;
import org.junit.Test;

public class WordOrderTest {

    @Test
    public void thisIsATestMethod() {
        Assert.assertEquals(7, 2 + 3);
    }

    @Ignore("Not ready yet")
    @Test(timeout = 100)
    public void thisIsAnotherTestMethod() {
        WorldOrder newWorldOrder = new WorldOrder();
        Assert.assertFalse(newWorldOrder.isComing());
    }

}
```

# KONSTRUKCJA TESTU

- ▶ Biblioteka junit udostępnia zbiór predefiniowanych asercji w postaci statycznych metod klasy `org.junit.Assert` np.:
  - ▶ `assertEquals`
  - ▶ `assertArrayEquals`
  - ▶ `assertTrue`
  - ▶ `assertFalse`
  - ▶ `assertNull`
  - ▶ `assertNotNull`
- ▶ Niespełnienie asercji kończy się wyrzuceniem wyjątku a tym samym oblanie testu

# MATCHERS

- ▶ Mechanizm umożliwiający eleganckie rozszerzanie istniejącego zbioru asercji
- ▶ Użycie
  - ▶ `assertThat(someObject, [matchesThisCondition]);`
    - ▶ `someObject` - obiekt lub wartość będąca kontekstem asercji
    - ▶ `matchesThisCondition` - obiekt typu `Matcher`, realizujący fizyczną weryfikację warunku
- ▶ Standardowa kolekcja/biblioteka obiektów weryfikujących
  - ▶ <https://github.com/hamcrest/JavaHamcrest>

# WERYFIKACJA STANÓW WYJĄTKOWYCH

```
@Test(expected = IllegalArgumentException.class)
public void ensureThatInvalidPhoneNumberYieldsProperException() {
    FaxMachine fax = new FaxMachine();
    fax.connect("+n0t-a-ph0n3-Numb3r"); // should throw an exception
}
```

# WERYFIKACJA STANÓW WYJĄTKOWYCH

```
@Test
public void ensureThatInvalidPhoneNumberYieldsProperException() {
    String invalidPhoneNumber = "+n0t-a-val1d-ph0n3-Numb3r";
    FaxMachine fax = new FaxMachine();
    try {
        fax.connect(invalidPhoneNumber);
        Assert.fail("should've raised an exception by now");
    } catch (IllegalArgumentException expected) {
        Assert.assertThat(expected.getMessage(), containsString(invalidPhoneNumber));
    }
}
```

## CYKL ŻYCIA TESTU

- ▶ Utworzenie instancji klasy testu
- ▶ Wykonanie metod inicjujących
- ▶ Wykonanie metody testu
- ▶ Wykonanie metod sprzątających



# INICJALIZACJA I SPRZĄTANIE

- ▶ Metody oznaczone adnotacjami `@Before/@After` zostaną wykonane odpowiednio przed/po metodzie testu; kolejność wykonania nie jest gwarantowana
- ▶ Statyczne metody oznaczone adnotacjami `@BeforeClass/@AfterClass` zostaną wykonane tylko raz, odpowiednio przed/po wykonaniu wszystkich testów z danej klasy; kolejność wykonania nie jest gwarantowana

# CECHY DOBRYCH TESTÓW

- ▶ Szybkie
- ▶ Spójne
- ▶ Izolowane
- ▶ Powtarzalne
- ▶ Celowe

# OBIEKTY ZASTĘPCZE (TEST DOUBLES)

- ▶ Zastępują zewnętrzne zależności klasy i umożliwiają testowanie w pełnej izolacji
- ▶ Implementowane ręcznie lub generowanie z użyciem zewnętrznych bibliotek np. Mockito

# KLASYFIKACJA OBIEKTÓW ZASTĘPCZYCH

- ▶ **Dummy** - zwraca wartość podaną wprost
- ▶ **Stub** - zwraca predefiniowaną odpowiedź dla zadanych parametrów wejściowych
- ▶ **Fake** - prosta implementacja imitująca działanie obiektu docelowego
- ▶ **Spy** - pozwala testować zachowania (interakcje) obiektu docelowego
- ▶ **Mock** - programowalny obiekt zachowujący się zgodnie z narzuconymi oczekiwaniami

# MOCKITO

- ▶ Popularny, darmowy, otwarty framework automatyzujący tworzenie i użycie obiektów zastępczych
- ▶ Oferuje proste i czyste API
- ▶ Nie wymaga wstępnej konfiguracji

# MOCKITO - WERYFIKACJA INTERAKCJI

```
01 //Let's import Mockito statically so that the code looks clearer
02 import static org.mockito.Mockito.*;
03
04 //mock creation
05 List mockedList = mock(List.class);
06
07 //using mock object
08 mockedList.add("one");
09 mockedList.clear();
10
11 //verification
12 verify(mockedList).add("one");
13 verify(mockedList).clear();
```

# MOCKITO - STUBBING

```
01 //You can mock concrete classes, not only interfaces
02 LinkedList mockedList = mock(LinkedList.class);
03
04 //stubbing
05 when(mockedList.get(0)).thenReturn("first");
06 when(mockedList.get(1)).thenThrow(new RuntimeException());
07
08 //following prints "first"
09 System.out.println(mockedList.get(0));
10
11 //following throws runtime exception
12 System.out.println(mockedList.get(1));
13
14 //following prints "null" because get(999) was not stubbed
15 System.out.println(mockedList.get(999));
16
17 //Although it is possible to verify a stubbed invocation, usually it's just redundant
18 //If your code cares what get(0) returns then something else breaks (often before even verify() gets executed).
19 //If your code doesn't care what get(0) returns then it should not be stubbed. Not convinced? See here.
20 verify(mockedList).get(0);
```

# MOCKITO - MATCHERS

```
01 //stubbing using built-in anyInt() argument matcher
02 when(mockedList.get(anyInt())).thenReturn("element");
03
04 //stubbing using hamcrest (let's say isValid() returns your own hamcrest matcher):
05 when(mockedList.contains(argThat(isValid()))).thenReturn("element");
06
07 //following prints "element"
08 System.out.println(mockedList.get(999));
09
10 //you can also verify using an argument matcher
11 verify(mockedList).get(anyInt());
```



# MOCKITO - KONTROLA WYWOŁAŃ METOD

```
01 //using mock
02 mockedList.add("once");
03
04 mockedList.add("twice");
05 mockedList.add("twice");
06
07 mockedList.add("three times");
08 mockedList.add("three times");
09 mockedList.add("three times");
10
11 //following two verifications work exactly the same - times(1) is used by default
12 verify(mockedList).add("once");
13 verify(mockedList, times(1)).add("once");
14
15 //exact number of invocations verification
16 verify(mockedList, times(2)).add("twice");
17 verify(mockedList, times(3)).add("three times");
18
19 //verification using never(). never() is an alias to times(0)
20 verify(mockedList, never()).add("never happened");
21
22 //verification using atLeast()/atMost()
23 verify(mockedList, atLeastOnce()).add("three times");
24 verify(mockedList, atLeast(2)).add("five times");
25 verify(mockedList, atMost(5)).add("three times");
```

# MOCKITO - KONTROLA KOLEJNOŚCI WYWOŁAŃ METOD

```
01 // A. Single mock whose methods must be invoked in a particular order
02 List singleMock = mock(List.class);
03
04 //using a single mock
05 singleMock.add("was added first");
06 singleMock.add("was added second");
07
08 //create an inOrder verifier for a single mock
09 InOrder inOrder = inOrder(singleMock);
10
11 //following will make sure that add is first called with "was added first, then with "was added second"
12 inOrder.verify(singleMock).add("was added first");
13 inOrder.verify(singleMock).add("was added second");
14
15 // B. Multiple mocks that must be used in a particular order
16 List firstMock = mock(List.class);
17 List secondMock = mock(List.class);
18
19 //using mocks
20 firstMock.add("was called first");
21 secondMock.add("was called second");
22
23 //create inOrder object passing any mocks that need to be verified in order
24 InOrder inOrder = inOrder(firstMock, secondMock);
25
26 //following will make sure that firstMock was called before secondMock
27 inOrder.verify(firstMock).add("was called first");
28 inOrder.verify(secondMock).add("was called second");
29
30 // Oh, and A + B can be mixed together at will
```

# MOCKITO - KONTROLA KROTNOŚCI WYWOŁAŃ METOD

```
01 //using mocks - only mockOne is interacted
02 mockOne.add("one");
03
04 //ordinary verification
05 verify(mockOne).add("one");
06
07 //verify that method was never called on a mock
08 verify(mockOne, never()).add("two");
09
10 //verify that other mocks were not interacted
11 verifyZeroInteractions(mockTwo, mockThree);
```

```
1 //using mocks
2 mockedList.add("one");
3 mockedList.add("two");
4
5 verify(mockedList).add("one");
6
7 //following verification will fail
8 verifyNoMoreInteractions(mockedList);
```

## MOCKITO - WYJĄTKI

```
1 doThrow(new RuntimeException()).when(mockedList).clear();  
2  
3 //following throws RuntimeException:  
4 mockedList.clear();
```

## MOCKITO - ZWRACANIE RÓŻNYCH WARTOŚCI

```
01 when(mock.someMethod("some arg"))
02     .thenThrow(new RuntimeException())
03     .thenReturn("foo");
04
05 //First call: throws runtime exception:
06 mock.someMethod("some arg");
07
08 //Second call: prints "foo"
09 System.out.println(mock.someMethod("some arg"));
10
11 //Any consecutive call: prints "foo" as well (last stubbing wins).
12 System.out.println(mock.someMethod("some arg"));

1 when(mock.someMethod("some arg"))
2     .thenReturn("one", "two", "three");
```

## MOCKITO - OBIEKTY SZPIEGUJĄCE

```
01 List list = new LinkedList();
02 List spy = spy(list);
03
04 //optionally, you can stub out some methods:
05 when(spy.size()).thenReturn(100);
06
07 //using the spy calls *real* methods
08 spy.add("one");
09 spy.add("two");
10
11 //prints "one" - the first element of a list
12 System.out.println(spy.get(0));
13
14 //size() method was stubbed - 100 is printed
15 System.out.println(spy.size());
16
17 //optionally, you can verify
18 verify(spy).add("one");
19 verify(spy).add("two");
```

# SOLID PRINCIPLES

- ▶ Zbiór podstawowych założeń dotyczących ogólnych zasad programowania obiektowego, wprowadzonych przez Roberta C. Martina

# SOLID PRINCIPLES

- ▶ **Single responsibility principle** - klasy powinny być jak najbardziej wyspecjalizowane (idealnie gdyby były skupione na realizacji jednego zadania)
- ▶ **Open closed principle** - jednostki kodu (klasy, moduły, metody itd.) powinny być otwarte na rozszerzanie, ale zamknięte na zmiany
- ▶ **Liskov substitution principle** - zamiana obiektów na instancje typu pochodnego nie powinna skutkować koniecznością wprowadzania zmian w programie
- ▶ **Interface segregation principle** - interfejsy powinny być mocno wyspecjalizowane (dużo specyficznych interfejsów jest lepszych niż jeden ogólny)
- ▶ **Dependency inversion principle** - moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych (użycie abstrakcji). Abstrakcje nie powinny być zależne od implementacji



# OGÓLNE ZASADY PROJEKTOWE

- ▶ Keep it Simple Stupid (KISS)
- ▶ Don't Repeat Yourself (DRY)
- ▶ Tell Don't Ask
- ▶ You Ani't Gonna Need It (YAGNI)
- ▶ Separation of Concerns

# SPRZĘŻENIE (ANG. COUPLING)

- ▶ Jest miarą określającą wzajemną zależność klas
- ▶ Należy dążyć do jak najmniejszego sprzężenia - obiekty powinny być od siebie zależne tylko w zakresie niezbędnym do realizacji założonego zadania

# SPÓJNOŚĆ (ANG. COHESION)

- ▶ Jest miarą jak bardzo klasa lub grupa klas przyczynia się do realizacji określonego celu
- ▶ Należy dążyć do jak największej spójności

# WZORCE PROJEKTOWE

---

# WZORZEC PROJEKTOWY

- ▶ Odkrywany na bazie doświadczenia, szablon sprawdzonego, a zarazem najlepszego rozwiązania powszechnie występującego problemu
- ▶ Koncepcja pochodzi od architekta (Christopher Alexander), który zaproponował wykorzystanie języka wzorców. Podejście to zostało przeniesione na grunt innych dziedzin m.in. inżynierii oprogramowania
- ▶ Wzorce związane z programowaniem obiektowym zostały zebrane w „Design Patterns Elements of Reusable Object-Oriented Software”, autorstwa: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides tzw. bandy czworga (GOF)

# KLASYFIKACJA

- ▶ Ze względu na rodzaj i rolę
  - ▶ kreacyjne - opisują proces tworzenia i konfigurowania nowych obiektów
  - ▶ strukturalne - opisują struktury obiektów / klas
  - ▶ czynnościowe - opisują zachowanie i odpowiedzialność współpracujących obiektów / powiązanych klas
- ▶ Ze względu na zakres
  - ▶ klasowe - opisują statyczne związki między klasami
  - ▶ obiektowe - opisują dynamiczne zależności między obiektami

# BUILDER, OBIEKTOWY, KONSTRUKCYJNY

## ▶ Przeznaczenie

- ▶ Oddziela sposób tworzenia złożonego obiektu od jego reprezentacji dzięki czemu jeden proces konstrukcji może skutkować powstawaniem różnych reprezentacji

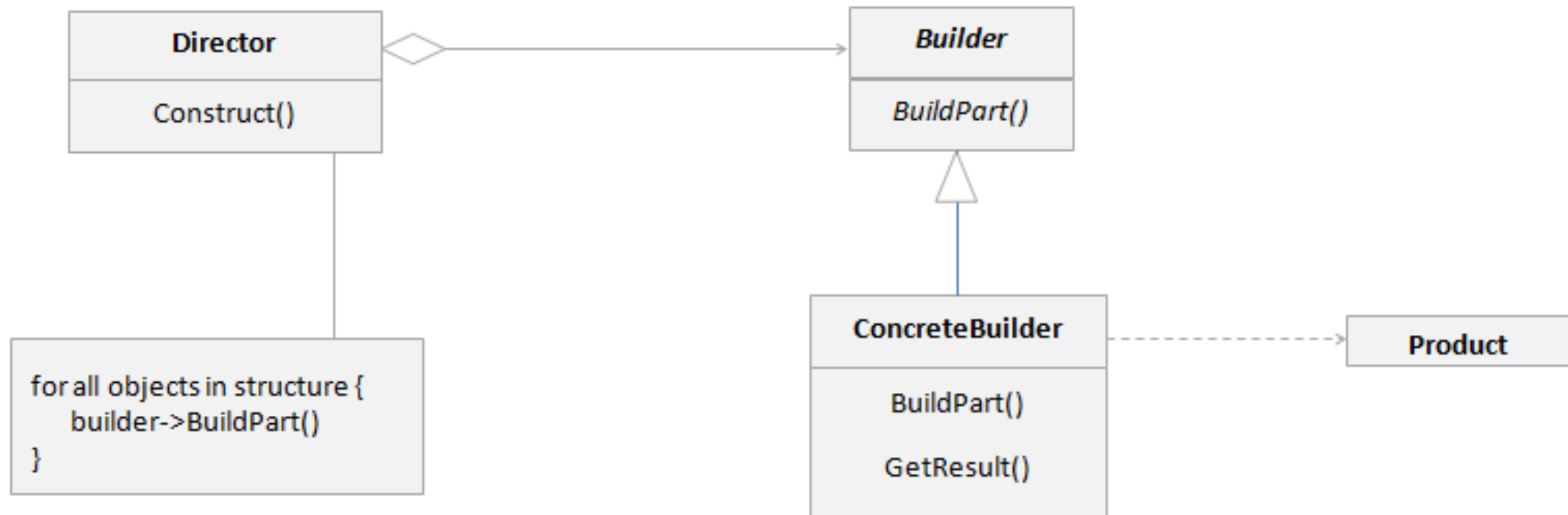
## ▶ Zastosowanie

- ▶ Kiedy algorytm tworzenia obiektu powinien być niezależny od składników tego obiektu, a także sposobu ich łączenia
- ▶ Kiedy proces konstrukcji obiektu musi gwarantować możliwość uzyskania obiektów w różnej konfiguracji

## ▶ Konsekwencje

- ▶ Elastyczność podczas zmian wewnętrznej konfiguracji
- ▶ Kontrola nad procesem wytwórczym i odizolowanie go od ostatecznego sposobu reprezentacji

# BUILDER, OBIKTOWY, KONSTRUKCYJNY





# FACTORY METHOD, KLASOWY, KONSTRUKCYJNY

### ▶ Przeznaczenie

- ▶ Definiuje interfejs umożliwiający tworzenie obiektów bez konieczności specyfikowania ich konkretnej implementacji - typ tworzonego obiektu jest określany w podklasach implementujących metodę wytwórczą

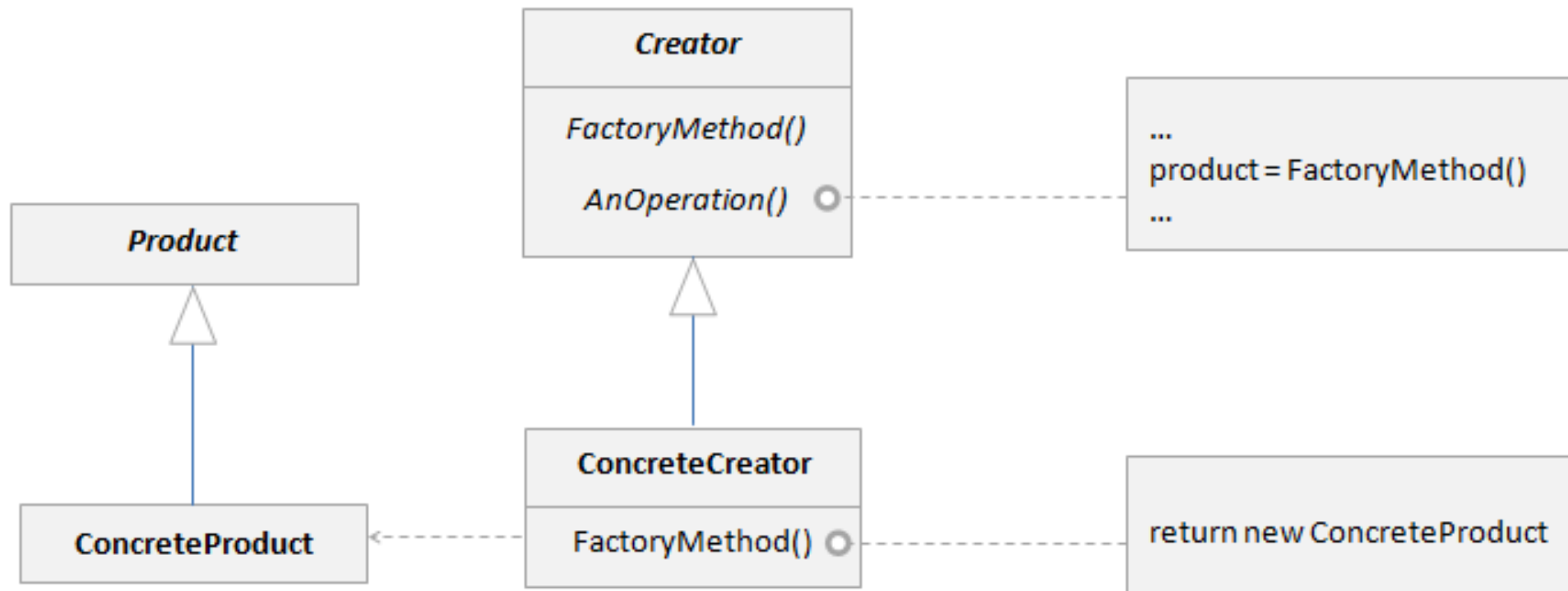
### ▶ Zastosowanie

- ▶ Kiedy nie wiadomo jaka konkretna implementacja klasy będzie wykorzystywana i zachodzi potrzeba jej elastycznej podmiany

### ▶ Konsekwencje

- ▶ Duża niezależności od konkretnej implementacji
- ▶ Możliwość łączenia równoległych hierarchii klas

# FACTORY METHOD, KLASOWY, KONSTRUKCYJNY



# ABSTRACT FACTORY, OBIEKTOWY, KONSTRUKCYJNY

### ▶ Przeznaczenie

- ▶ Dostarcza interfejs określający sposób tworzenia rodzin zależnych i powiązanych wzajemnie obiektów bez konieczności określania ich konkretnych klas

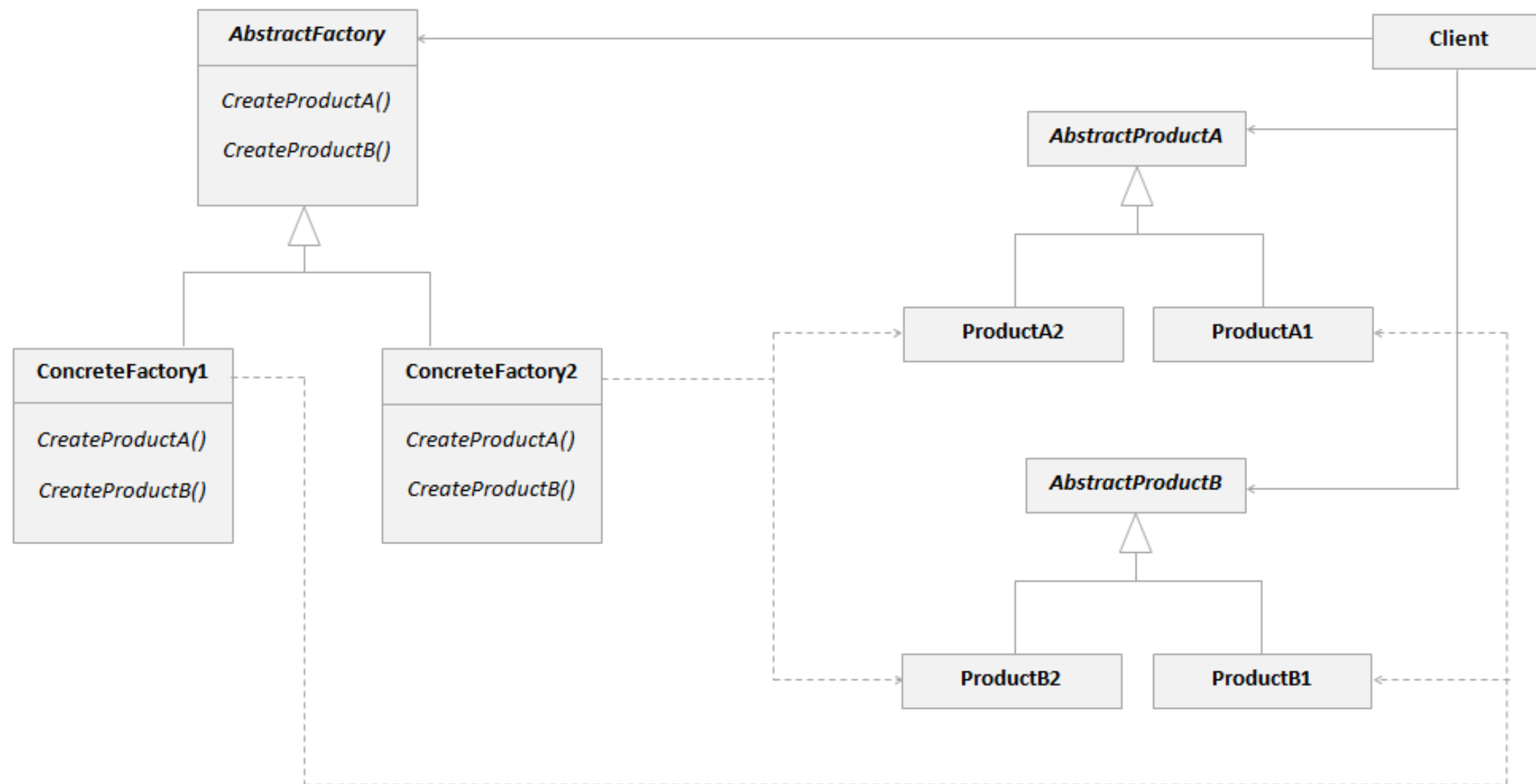
### ▶ Zastosowanie

- ▶ Kiedy system powinien być niezależny od tego w jaki sposób tworzone, komponowane lub reprezentowane są jego produkty/elementy
- ▶ Kiedy system musi być konfigurowany tak, aby mógł korzystać z jednej lub wielu rodzin produktów
- ▶ Kiedy rodzina produktów została zaprojektowana, aby używać jej w całości i należy wymusić spełnienie tego warunku

### ▶ Konsekwencje

- ▶ Izolacja klientów od konkretnych implementacji klas produktów
- ▶ Prosta podmiana rodziny produktów (promowanie spójności)
- ▶ Łatwe dodawanie nowych rodzin produktów
- ▶ Trudne dodawanie obsługi nowych produktów

# ABSTRACT FACTORY, OBIEKTOWY, KONSTRUKCYJNY



# PROTOTYPE, OBIEKTOWY, KONSTRUKCYJNY

### ▶ Przeznaczenie

- ▶ Określa rodzaj obiektu przy użyciu instancji prototypu i tworzy nowe obiekty poprzez jego klonowanie

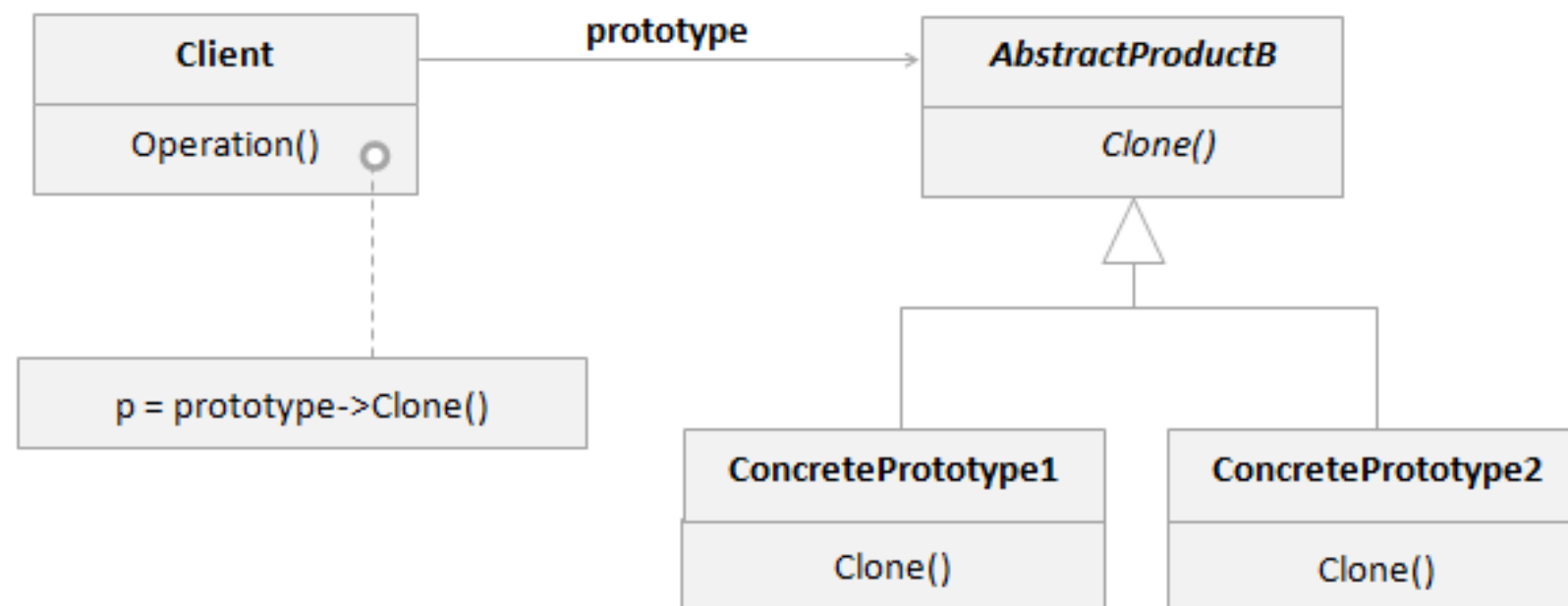
### ▶ Zastosowanie

- ▶ Kiedy klasy na bazie których mają być tworzone obiekty są dostarczane dynamicznie podczas działania aplikacji
- ▶ Kiedy instancja klasy może mieć tylko kilka różnych kombinacji stanu
- ▶ Kiedy chce się uniknąć tworzenia hierarchii klas fabryk odpowiadających hierarchii klas produktów

### ▶ Konsekwencje

- ▶ Możliwość dodawania i usuwania produktów w czasie wykonywania programu
- ▶ Możliwość określania nowych obiektów poprzez zmianę wartości lub struktury
- ▶ Redukcja liczby klas, która wynika z użycia wzorca Factory Method

# PROTOTYPE, OBIKTOWY, KONSTRUKCYJNY



# SINGLETON, OBIEKTOWY, KONSTRUKCYJNY

### ▶ Przeznaczenie

- ▶ Zapewnia istnienie tylko jednej instancji klasy i dostarcza globalny punkt dostępu do niej

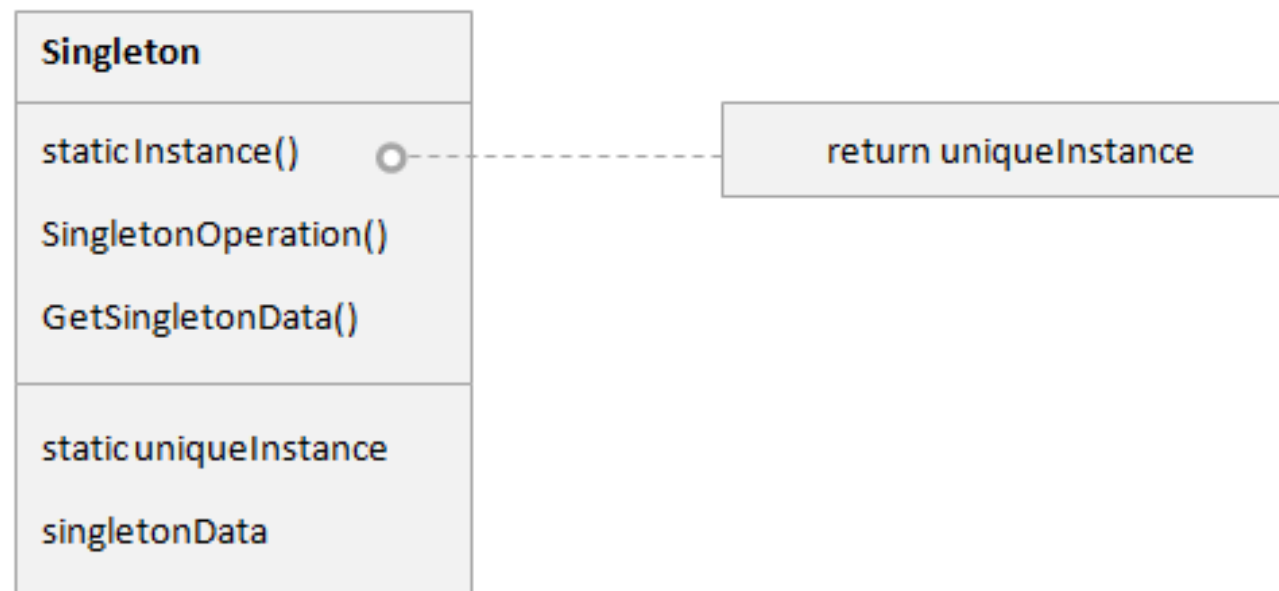
### ▶ Zastosowanie

- ▶ Kiedy musi istnieć dokładnie jedna instancja klasy i musi być ona dostępna w sposób dobrze znany klientom
- ▶ Kiedy istnieje potrzeba rozszerzania jedyne go egzemplarza klasy przez tworzenie podklas, bez konieczności modyfikacji kodu klienckiego

### ▶ Konsekwencje

- ▶ Kontrola dostępu do jedyne go egzemplarza
- ▶ Możliwość tworzenia przestrzeni nazewniczych (redukcja zmiennych globalnych)
- ▶ Wymuszenie maksymalnej liczby instancji danego typu
- ▶ Dużo większa elastyczność niż w przypadku operacji statycznych

# SINGLETON, OBIKTOWY, KONSTRUKCYJNY





# ADAPTER, KLASOWY I OBIEKTOWY, STRUKTURALNY

### ▶ Przeznaczenie

- ▶ Dokonuje konwersji jednego interfejsu w drugi (na taki jakiego spodziewa się klient)
- ▶ Pozwala na współpracę klas niekompatybilnych pod względem interfejsów

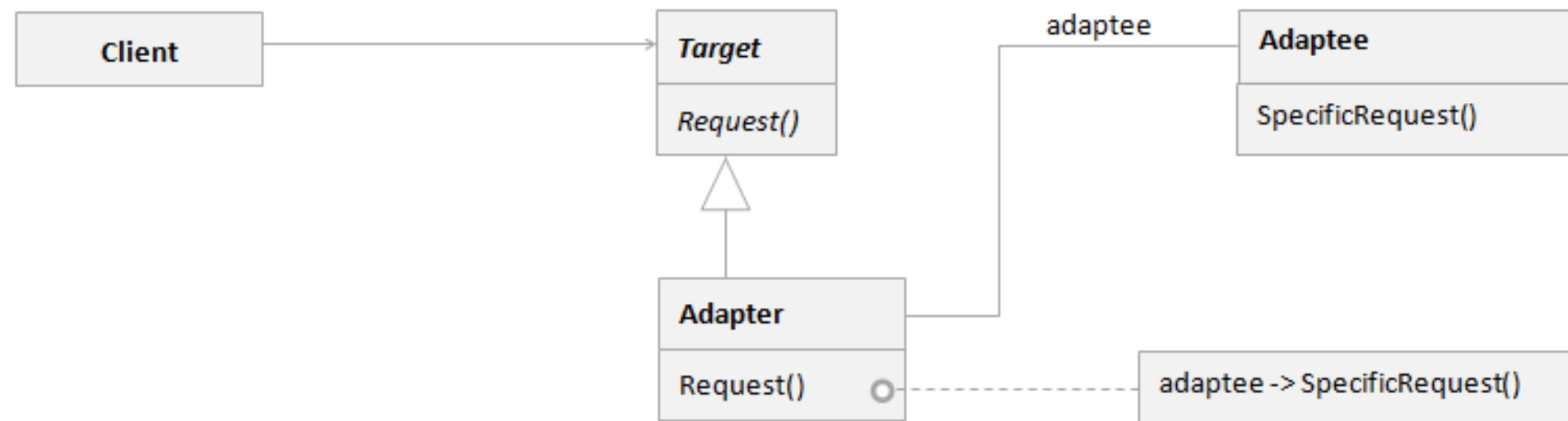
### ▶ Zastosowanie

- ▶ Kiedy musimy skorzystać z istniejącej klasy, ale jej interfejs nie odpowiada założonym wymaganiom
- ▶ Kiedy potrzeba stworzyć reużywalną klasę, która będzie kooperowała z niepowiązanymi lub niekompatybilnymi klasami
- ▶ Kiedy należy użyć kilku istniejących klas, ale dostosowywanie ich interfejsów poprzez tworzenie odpowiednich podklas jest niepraktyczne

### ▶ Konsekwencje

- ▶ Możliwość użycia niekompatybilnych klas/komponentów
- ▶ Możliwość opakowania istniejącego interfejsu w nowy
- ▶ Utrudnione przesłanianie zachowań adoptowanej klasy

# ADAPTER, KLASOWY I OBIEKTOWY, STRUKTURALNY



# BRIDGE, OBIEKTOWY, STRUKTURALNY

### ▶ Przeznaczenie

- ▶ Dokonuje separacji abstrakcji od implementacji tak, aby mogły one być niezależnie modyfikowane

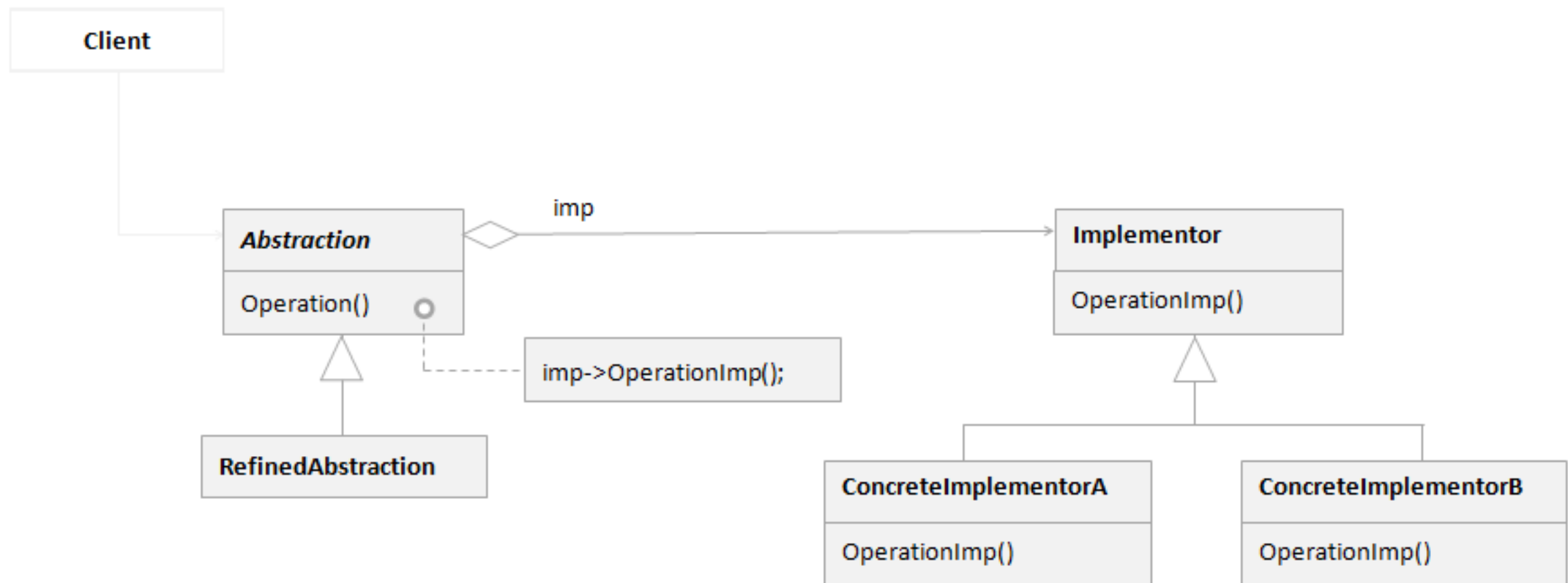
### ▶ Zastosowanie

- ▶ Kiedy zależy nam na separacji abstrakcji od jej implementacji np. w celu jej zmiany/wyboru podczas wykonywania kodu
- ▶ Kiedy zarówno abstrakcja jak i implementacja powinny być niezależnie rozszerzalne przez dziedziczenie
- ▶ Kiedy zmiany w implementacji abstrakcji nie powinny mieć wpływu na kod klientów

### ▶ Konsekwencje

- ▶ Rozdzielenie interfejsu abstrakcji od jej implementacji
- ▶ Poprawiona elastyczność i rozszerzalność
- ▶ Izolacja kodu klientów od konkretnej implementacji

# BRIDGE, OBIKTOWY, STRUKTURALNY



# COMPOSITE, OBIEKTOWY, STRUKTURALNY

## ▶ Przeznaczenie

- ▶ Dokonuje kompozycji obiektów w strukturę o charakterze drzewiastym
- ▶ Pozwala traktować obiekty indywidualne oraz złożone w jednakowy sposób

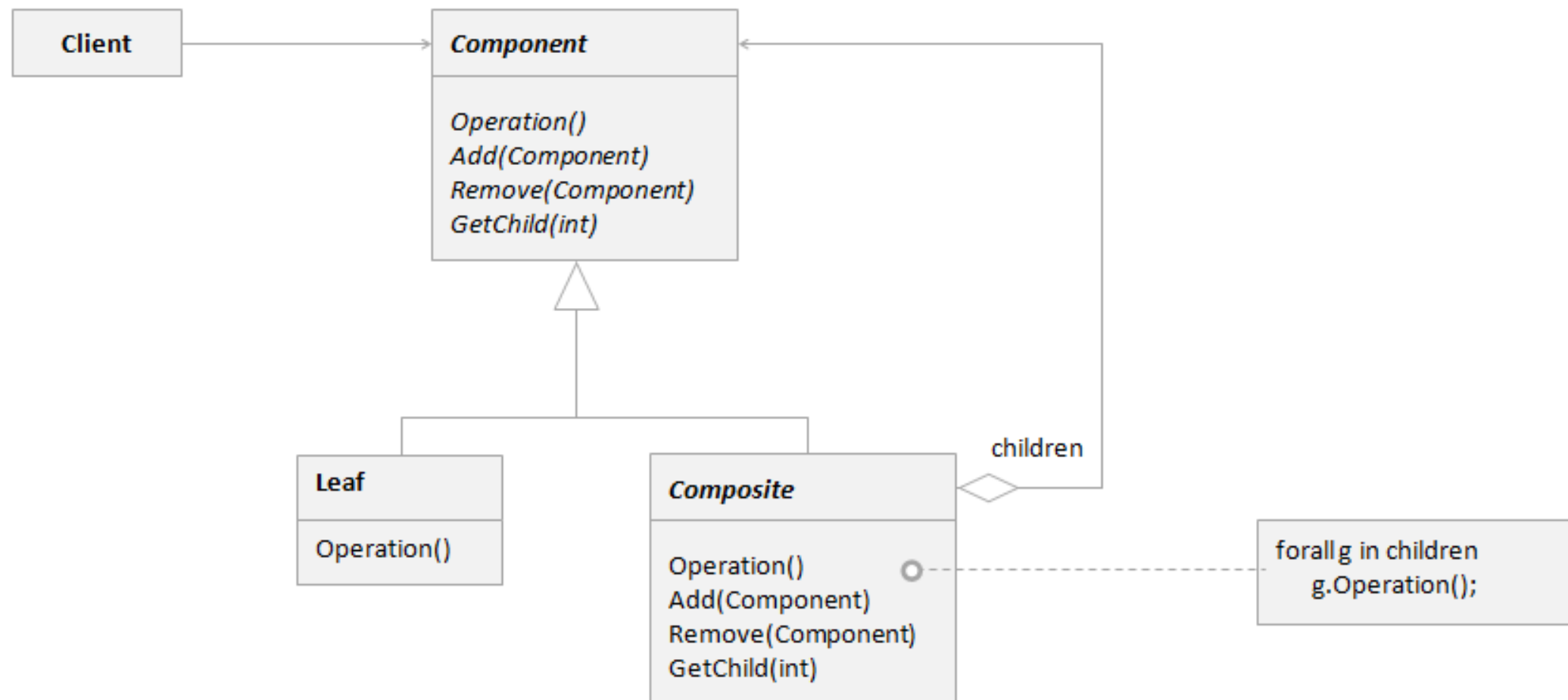
## ▶ Zastosowanie

- ▶ Kiedy zachodzi potrzeba reprezentacji zależnych obiektów w formie struktury drzewiastej - wzajemnie zagnieżdżonych elementów którymi mogą być liście lub kompozyty
- ▶ Kiedy wszystkie elementy struktury powinny być traktowane w jednakowy sposób mimo występujących między nimi różnic

## ▶ Konsekwencje

- ▶ Możliwość definiowania i zarządzania hierarchiami zawierającymi obiekty pierwotne i złożone
- ▶ Łatwe wykonywanie operacji na wybranych fragmentach hierarchii
- ▶ Uproszczony kod

# COMPOSITE, OBIEKTOWY, STRUKTURALNY



# DECORATOR, OBIEKTOWY, STRUKTURALNY

### ▶ Przeznaczenie

- ▶ Dodawanie odpowiedzialności / zachowań do obiektu w dynamiczny sposób
- ▶ Dostarczenie elastycznej alternatywy dla mechanizmu dziedziczenia

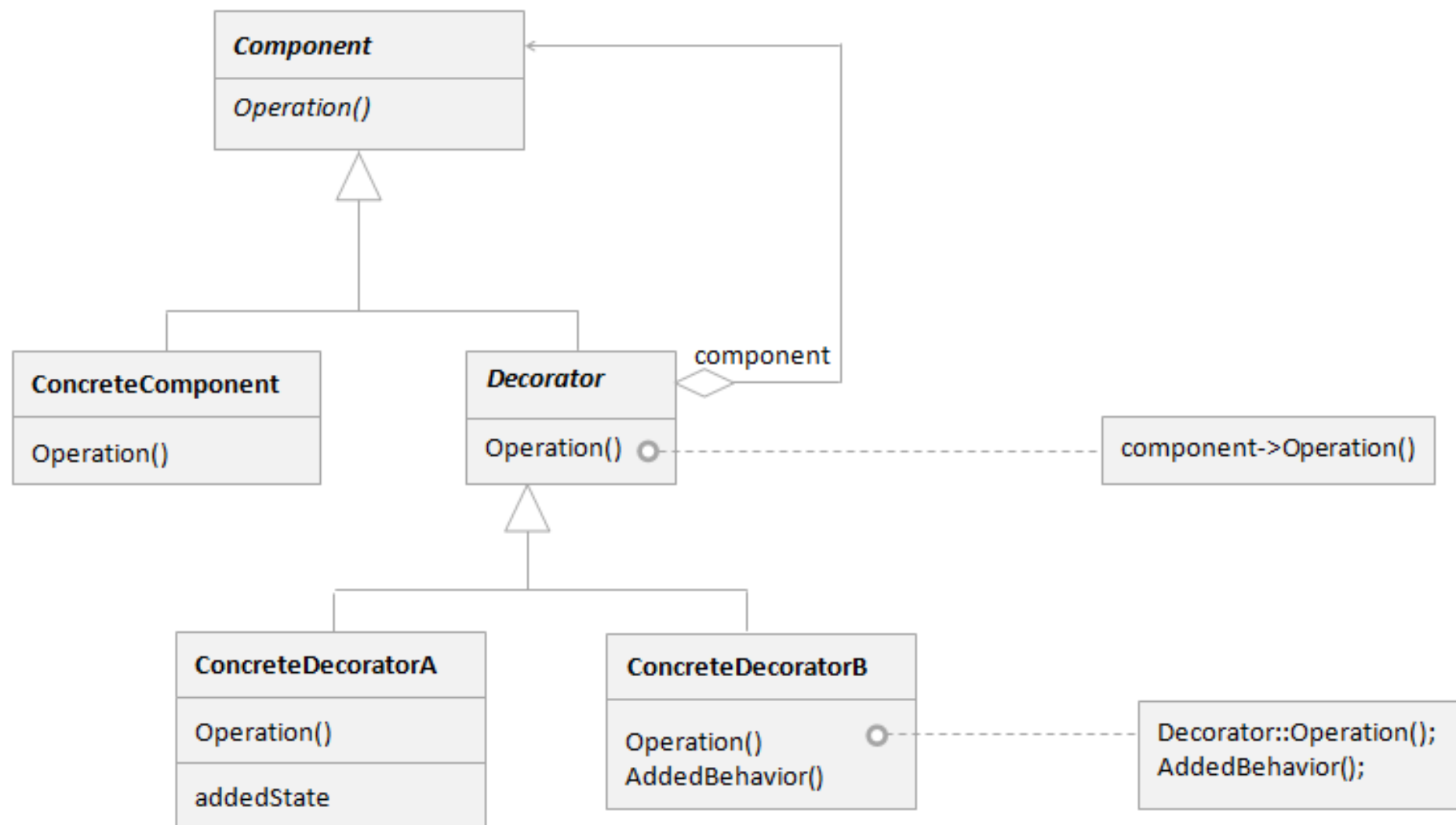
### ▶ Zastosowanie

- ▶ Kiedy zachodzi potrzeba dodania nowych odpowiedzialności do obiektu w dynamiczny i przeźroczysty sposób - bez wymuszania zmian w pozostałym kodzie
- ▶ Kiedy rozszerzanie funkcjonalności przez dziedziczenie jest niepraktyczne np. z powodu konieczności utworzenia ogromnej ilości podklas

### ▶ Konsekwencje

- ▶ Większa elastyczność niż przy zwykłym dziedziczeniu
- ▶ Możliwość dynamicznej modyfikacji zachowań obiektów
- ▶ Uproszczenie kodu (brak przeładowanych klas)

# DECORATOR, OBIEKTOWY, STRUKTURALNY





# FACADE, OBIEKTOWY, STRUKTURALNY

### ▶ Przeznaczenie

- ▶ Dostarczanie jednego, zunifikowanego interfejsu dla zbioru interfejsów poszczególnych podsystemów
- ▶ Zdefiniowanie interfejsu wyższego poziomu ułatwiającego wykorzystanie z systemu

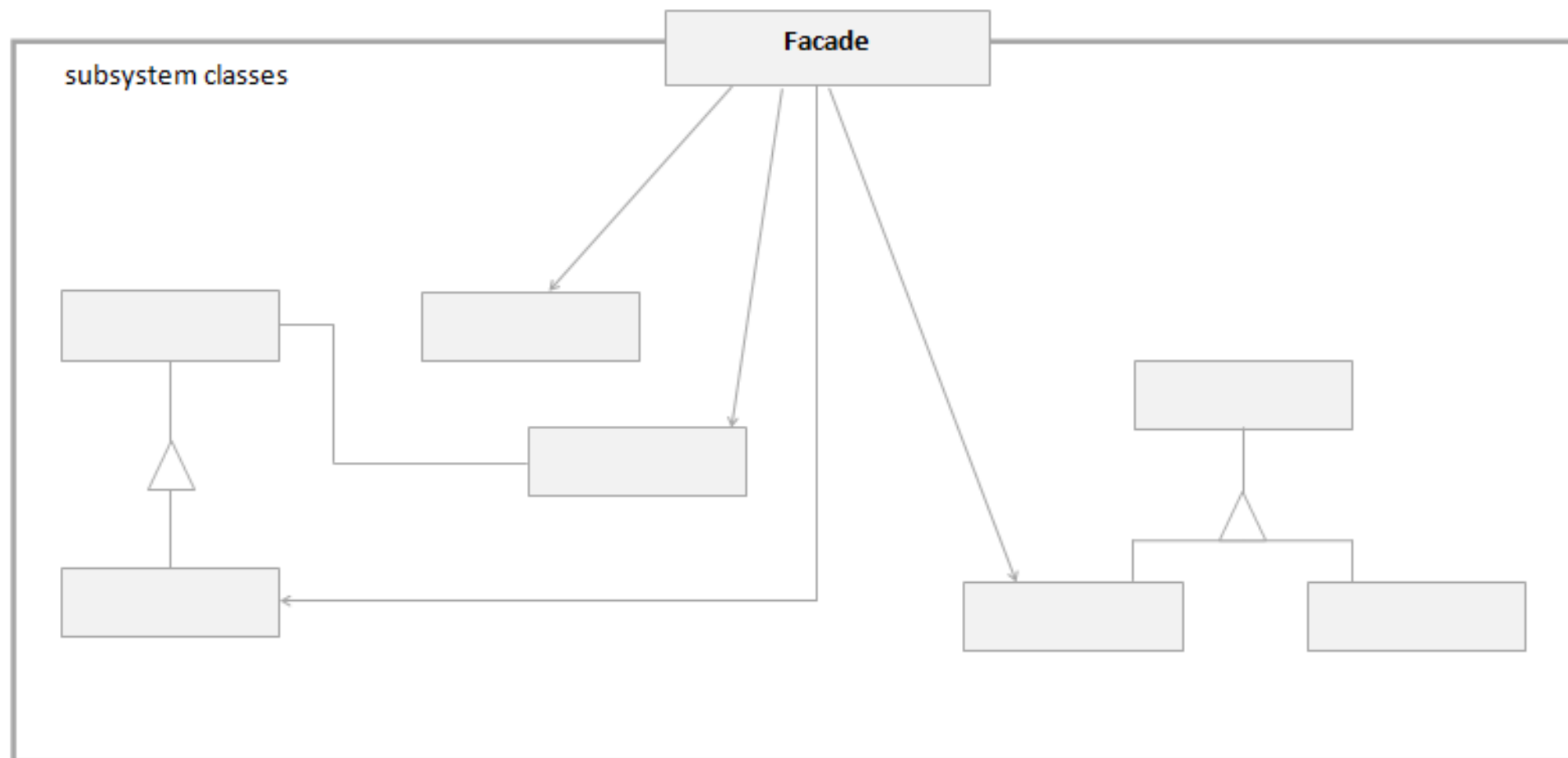
### ▶ Zastosowanie

- ▶ Kiedy chcemy dostarczyć prosty interfejs dla złożonego podsystemu/podsystemów
- ▶ Kiedy istnieje wiele zależności między klientami oraz używanymi przez nich podsystemami i należy je zredukować
- ▶ Kiedy trzeba ułatwić komunikację z wieloma systemem i zdefiniować centralny punkt wejścia

### ▶ Konsekwencje

- ▶ Izolacja klientów od podsystemów i szczegółów niskiego poziomu
- ▶ Niskie sprzężenie między systemami i ich podsystemami

# FACADE, OBIEKTOWY, STRUKTURALNY



# PROXY, OBIEKTOWY, STRUKTURALNY

## ▶ Przeznaczenie

- ▶ Dostarczenie pośrednika dla obiektu tak aby kontrolować do niego dostęp

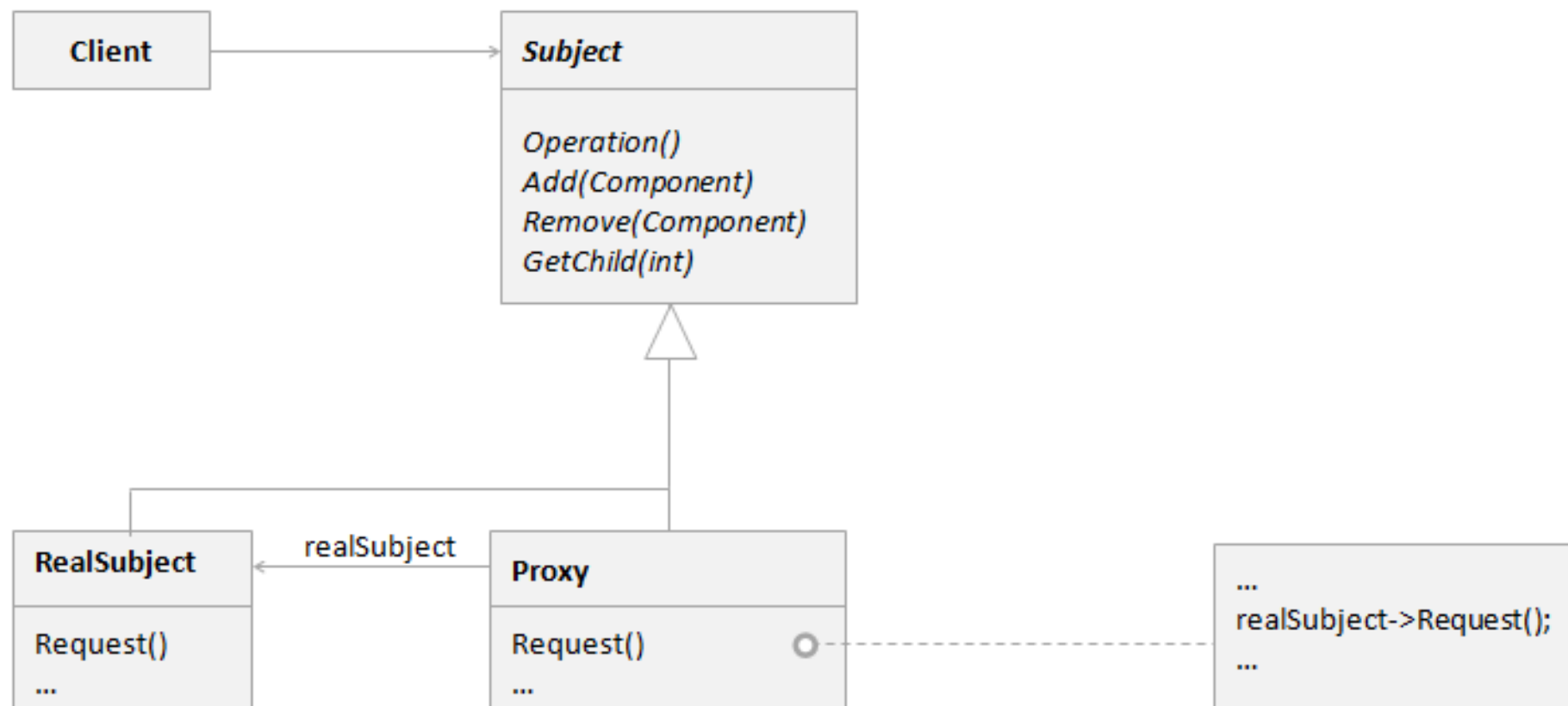
## ▶ Zastosowanie

- ▶ Kiedy potrzebne jest wprowadzenie dodatkowej warstwy (poziomu niezależności) pozwalającego na zdalny, kontrolowany albo inteligentny dostęp do obiektu
- ▶ Kiedy chcemy odizolować klienta od niepotrzebnej złożoności

## ▶ Konsekwencje

- ▶ Możliwość optymalizacji
- ▶ Możliwość ukrycia złożoności pewnych mechanizmów
- ▶ Możliwość kontroli dostępu
- ▶ Możliwość wzbogacania funkcjonalności w dynamiczny sposób

# PROXY, OBIEKTOWY, STRUKTURALNY



# FLYWEIGHT, OBIEKTOWY, STRUKTURALNY

## ▶ Przeznaczenie

- ▶ Konieczna jest wydajna obsługa dużej liczby małych obiektów

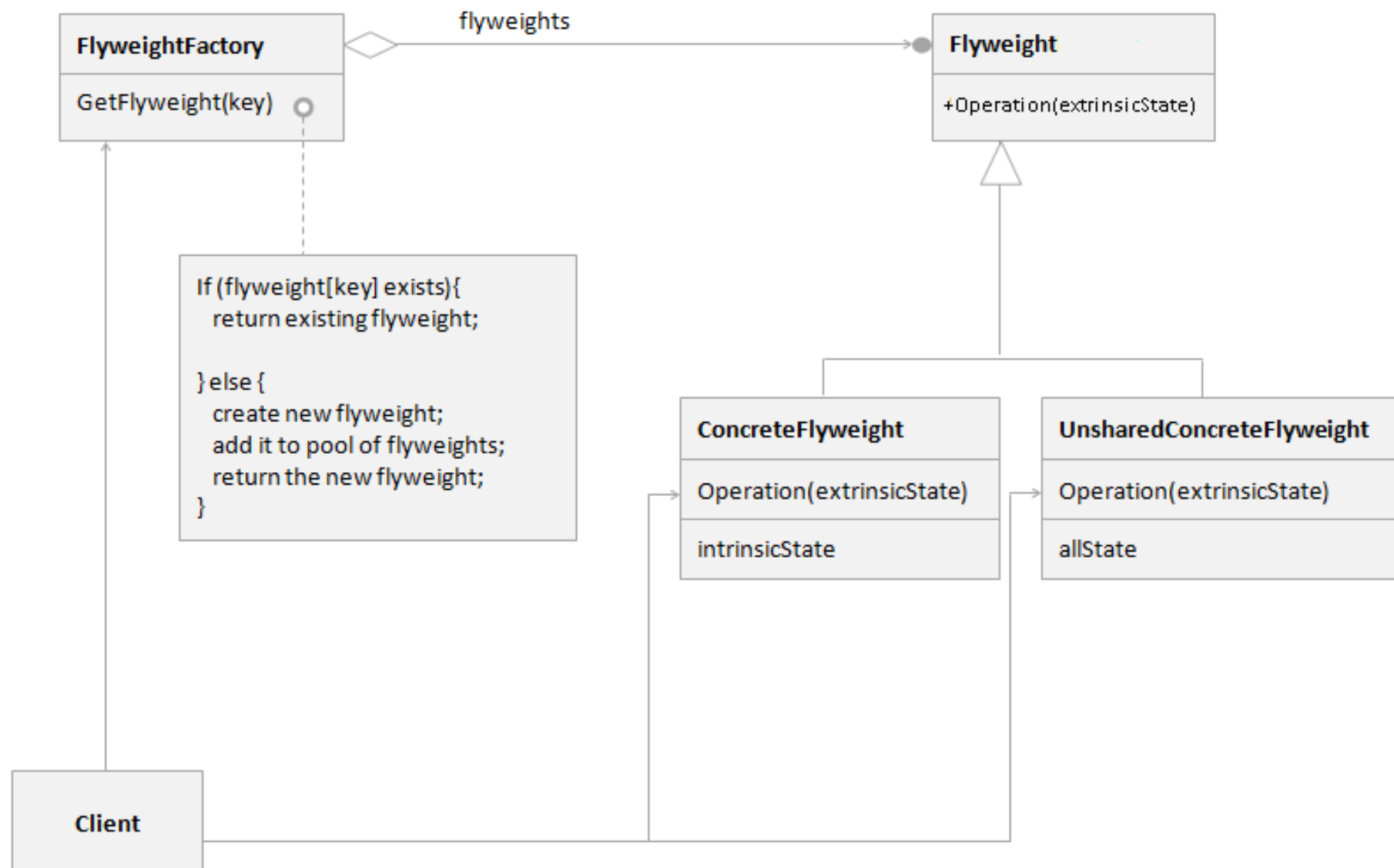
## ▶ Zastosowanie

- ▶ Kiedy aplikacja korzysta z dużej ilości obiektów, a ich koszty przechowywania/zarządzania są wysokie
- ▶ Kiedy większość stanu obiektów można wyodrębnić i zapisać poza nimi (zastąpić nielicznymi obiektami współużytkowanymi)
- ▶ Aplikacja nie zależy od tożsamości obiektów

## ▶ Konsekwencje

- ▶ Zmniejszenie zużywanych zasobów zależne od ilości obiektów, wielkości współdzielonego stanu i sposobu jego przechowywania

# FLYWEIGHT, OBIKTOWY, STRUKTURALNY



# CHAIN OF RESPONSIBILITY, OBIEKTOWY, BEHAWIORALNY

### ▶ Przeznaczenie

- ▶ Odseparowuje nadawcę i odbiorcę komunikatu
- ▶ Pozwala na obsłużenie komunikatu przez wielu odbiorców

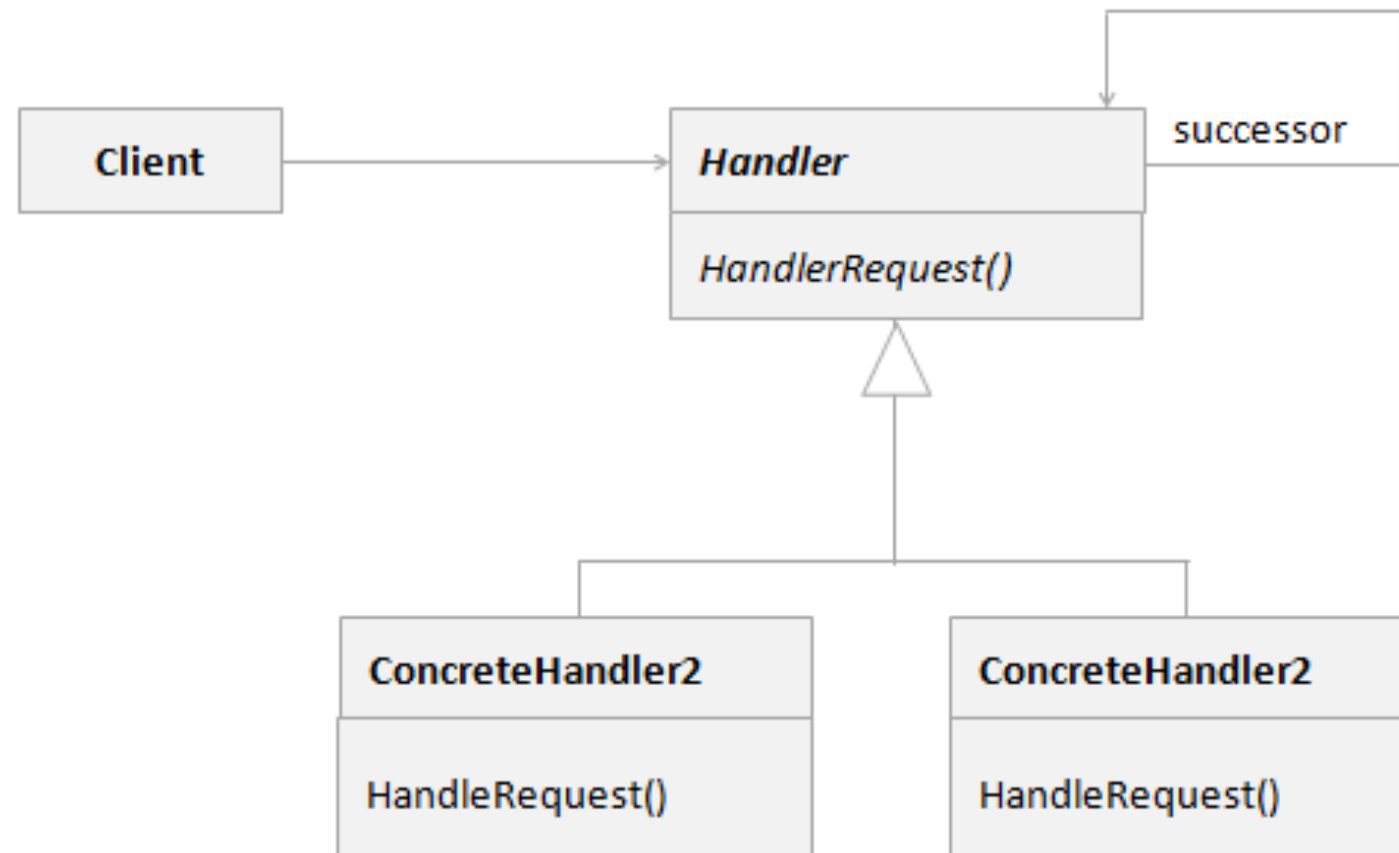
### ▶ Zastosowanie

- ▶ Kiedy więcej niż jeden odbiorca może potencjalnie obsłużyć żądanie, a nie wiadomo który z nich to zrobi
- ▶ Kiedy obiekt obsługujący żądanie powinien być ustalany automatycznie
- ▶ Kiedy należy przesłać żądanie do jednego lub kilku odbiorców bez ich konkretnego wskazywania
- ▶ Kiedy zbiór odbiorców żądania może być ustalany dynamicznie w czasie działania aplikacji

### ▶ Konsekwencje

- ▶ Niskie sprzężenie nadawcy i odbiorcy
- ▶ Elastyczność w zakresie przydzielania zadań
- ▶ Brak gwarancji odbioru żądania

# CHAIN OF RESPONSIBILITY, OBIKTOWY, BEHAWIORALNY





# COMMAND, OBIEKTOWY, BEHAWIORALNY

### ▶ Przeznaczenie

- ▶ Hermetyzuje żądanie w formie obiektu
- ▶ Pozwala na wykonywanie różnych operacji w taki sam sposób
- ▶ Umożliwia cofanie wykonywanych kroków

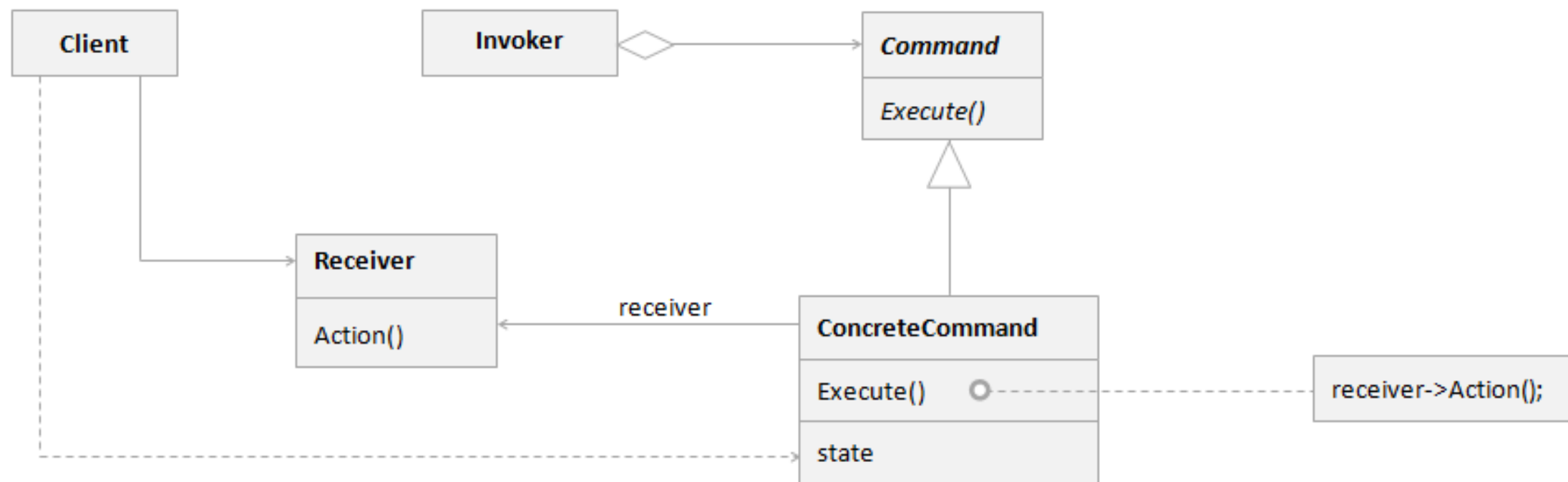
### ▶ Zastosowanie

- ▶ Kiedy potrzebny jest mechanizm typu callback
- ▶ Kiedy zachodzi konieczność zarządzania zadaniami - kolejkowanie i wywoływanie ich w odpowiednim momencie
- ▶ Kiedy konieczna jest możliwość rejestrowania, powtarzania i cofania zmian

### ▶ Konsekwencje

- ▶ Separacja obiektu wykonującego operację od obiektu który wie jak ją wykonać
- ▶ Możliwość przekazywania komend
- ▶ Łatwe dodawanie nowych komend
- ▶ Podniesienie poziomu abstrakcji (makra)

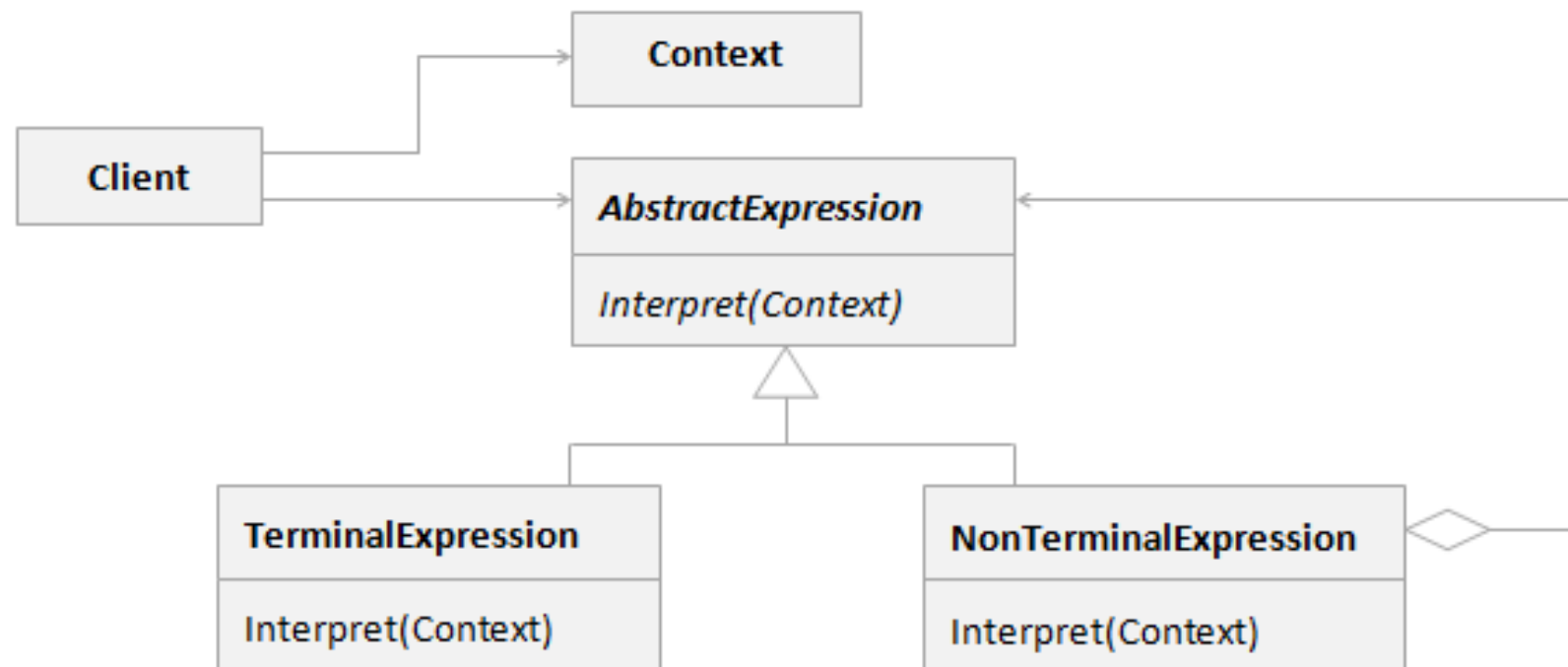
# COMMAND, OBIEKTOWY, BEHAWIORALNY



# INTERPRETER, KLASOWY, BEHAWIORALNY

- ▶ Przeznaczenie
  - ▶ Określa reprezentację gramatyki języka oraz interpreter, wykorzystujący tę reprezentację do interpretacji zdań danego języka
- ▶ Zastosowanie
  - ▶ Kiedy gramatyka języka jest prosta
  - ▶ Kiedy wydajność nie jest najważniejsza
- ▶ Konsekwencje
  - ▶ Modyfikacja i rozszerzanie gramatyki języka jest proste
  - ▶ Trudno zarządzanie złożoną gramatyką

# INTERPRETER, KLASOWY, BEHAWIORALNY



# ITERATOR, OBIEKTOWY, BEHAWIORALNY

- ▶ Przeznaczenie
  - ▶ Zapewnia sekwencyjny dostęp do elementów złożonego obiektu niezależnie od jego implementacji i wewnętrznej struktury
- ▶ Zastosowanie
  - ▶ Kiedy potrzebny jest dostęp do elementów agregatu bez ujawniania jego wewnętrznej struktury lub sposobu implementacji
  - ▶ Kiedy potrzebny jest jednolity interfejs, który umożliwi poruszanie się po różnych strukturach agregujących
- ▶ Konsekwencje
  - ▶ Spójny sposób poruszania się po obiektach agregujących

# ITERATOR, OBIEKTOWY, BEHAWIORALNY



# MEDIATOR, OBIEKTOWY, BEHAWIORALNY

### ▶ Przeznaczenie

- ▶ Zapewnia luźne powiązanie między współpracującymi obiektami biorąc na siebie komunikację między nimi
- ▶ Zapewnia możliwość niezależnej modyfikacji interakcji między współdziałającymi elementami

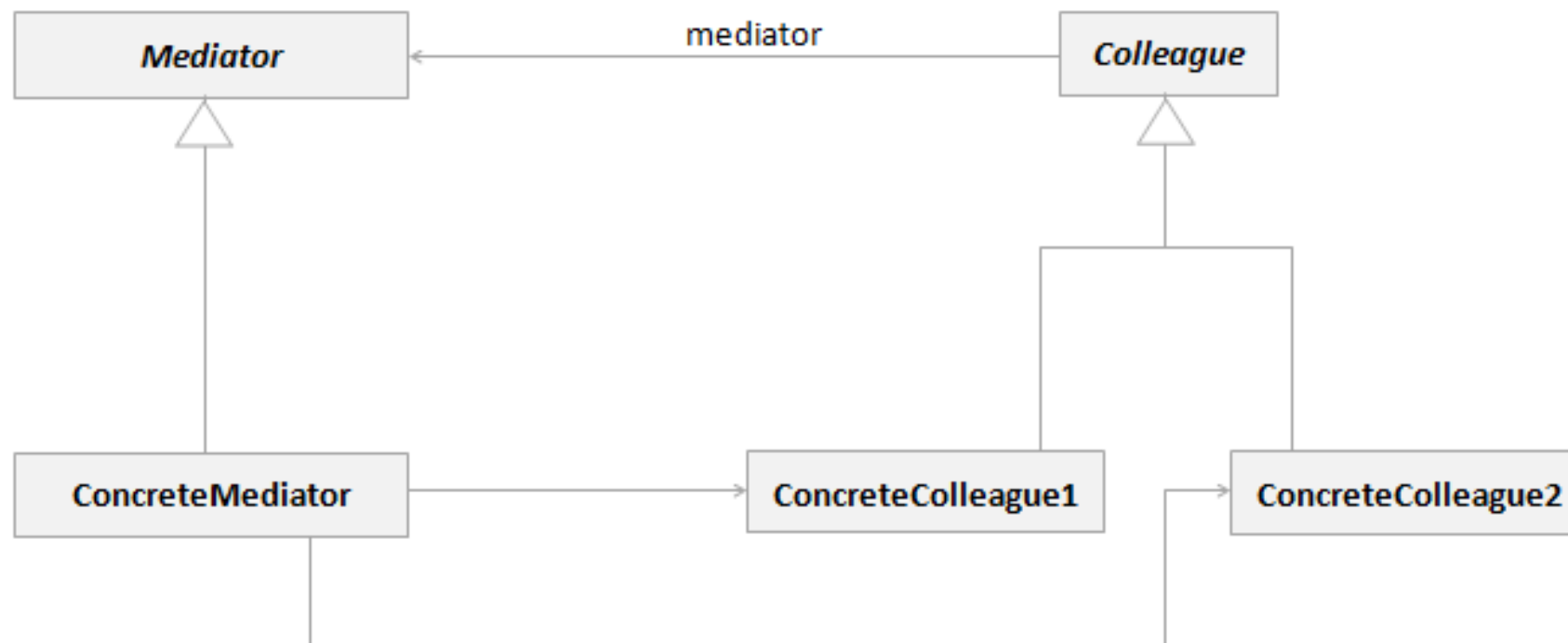
### ▶ Zastosowanie

- ▶ Kiedy zestaw obiektów komunikuje się w dobrze zdefiniowany, ale skomplikowany sposób
- ▶ Kiedy powiązania między obiektami uniemożliwiają/utrudniają ich modyfikację
- ▶ Kiedy powtarzające się zachowania skutkują utworzeniem wielu podklas

### ▶ Konsekwencje

- ▶ Rozdzielenie współpracujących obiektów
- ▶ Uproszczony protokół komunikacji
- ▶ Centralizacja sterowania
- ▶ Hermetyzacja komunikacji w obiekcie

# MEDIATOR, OBIEKTOWY, BEHAWIORALNY





# MEMENTO, OBIEKTOWY, BEHAWIORALNY

### ▶ Przeznaczenie

- ▶ Rejestruje i zapisuje wewnętrzny stan obiektu w zewnętrznym obszarze pamięci bez naruszenia kapsułkowania
- ▶ Umożliwia przywracanie określonego stanu obiektu

### ▶ Zastosowanie

- ▶ Kiedy trzeba zachować migawkę stanu obiektu w celu późniejszego odtworzenia, a nie może on być udostępniony przez standardowy interfejs aby nie naruszyć kapsułkowania

### ▶ Konsekwencje

- ▶ Zachowanie hermetyzacji
- ▶ Możliwość przywracania stanu
- ▶ Potencjalnie duże zużycie zasobów

# MEMENTO, OBIEKTOWY, BEHAWIORALNY



# OBSERVER, OBIEKTOWY, BEHAWIORALNY

### ▶ Przeznaczenie

- ▶ Określa zależność jeden do wielu między obiektami - kiedy zmienia się stan jednego obiektu wszystkie obiekty zależne są o tym informowane w ustandaryzowany sposób

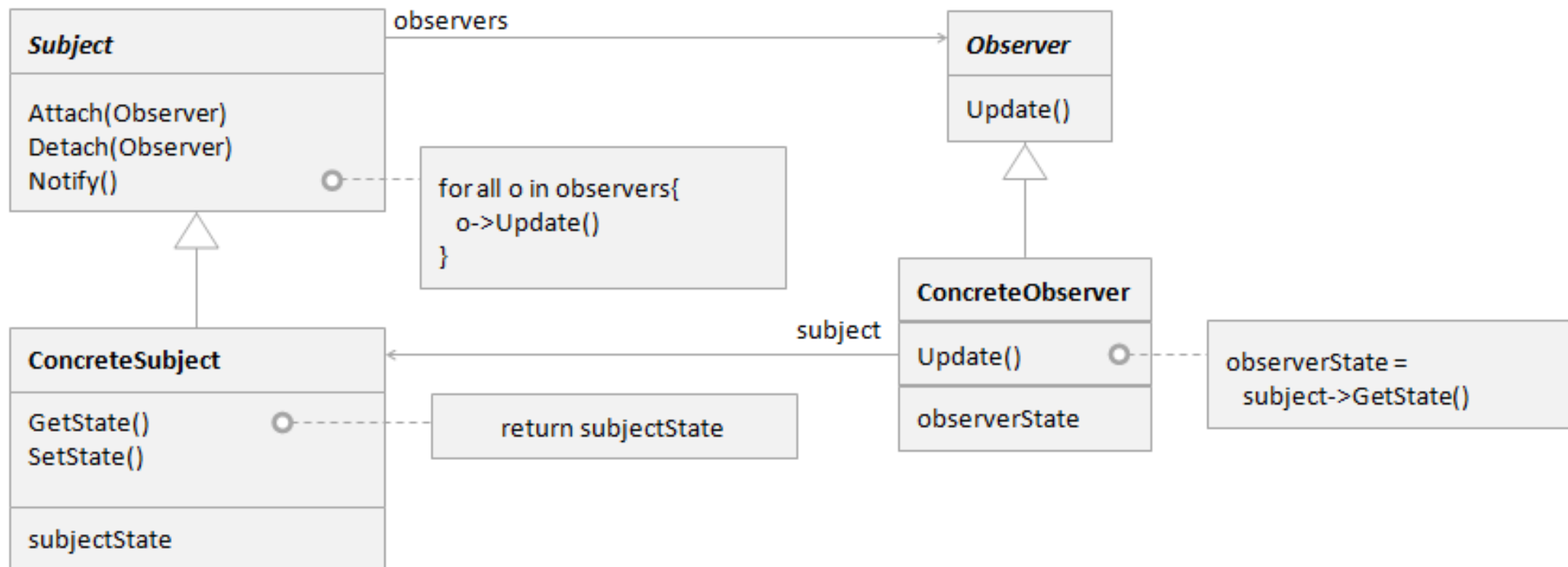
### ▶ Zastosowanie

- ▶ Kiedy zmiana stanu jednego obiektu może powodować zmianę stanu lub podjęcia działania w innych obiektach
- ▶ Kiedy obiekt powinien mieć możliwość powiadamiania innych obiektów o zmianie stanu (nie wiadomo ilu i jakich)
- ▶ Kiedy zamknięcie dwóch aspektów tej samej abstrakcji w oddzielnych obiektach ułatwia ich modyfikację

### ▶ Konsekwencje

- ▶ Obsługa grupowego dostarczania/rozsyłania komunikatów
- ▶ Luźne powiązanie obiektów

# OBSERVER, OBIEKTOWY, BEHAWIORALNY



# STATE, OBIEKTOWY, BEHAWIORALNY

### ▶ Przeznaczenie

- ▶ Pozwala na zmianę zachowania obiektu w zależności od jego aktualnego stanu, co wygląda jak by obiekt zmienił swoją klasę

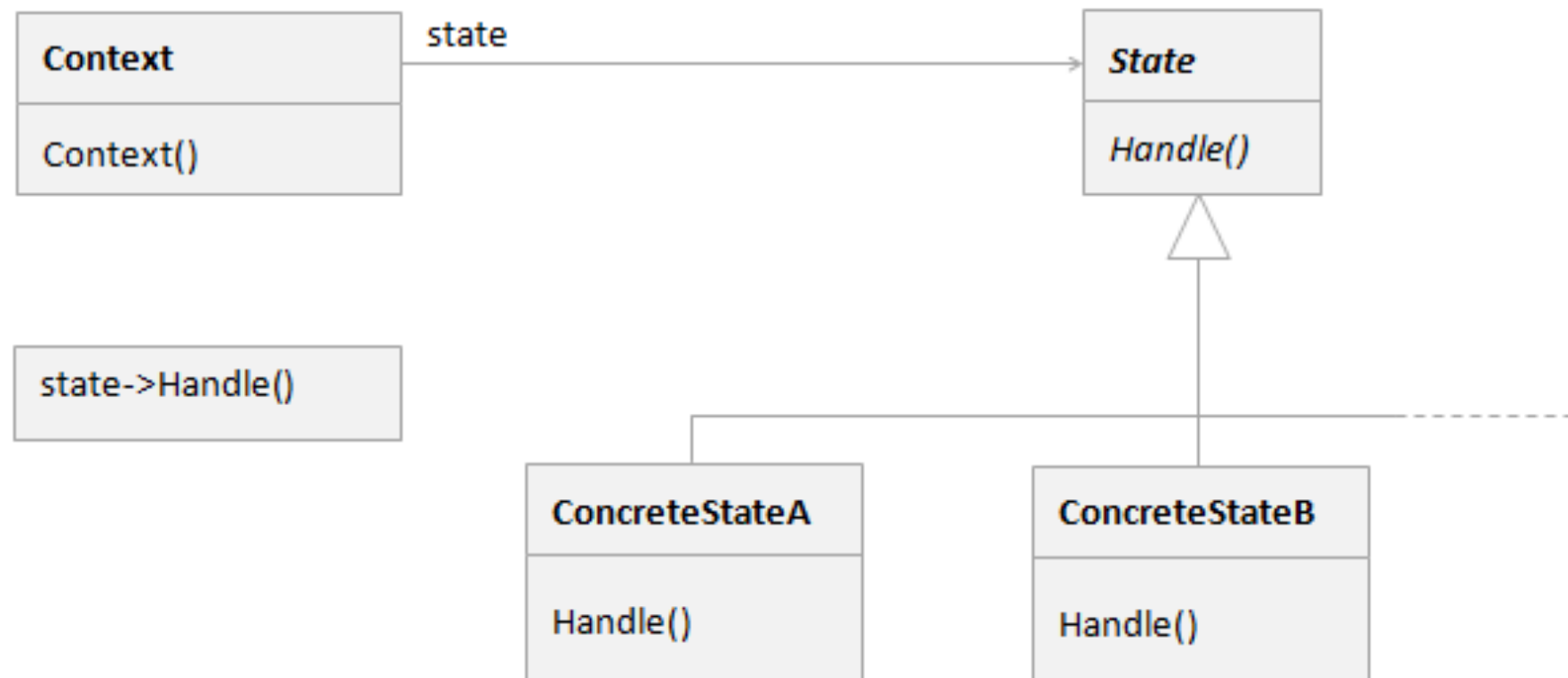
### ▶ Zastosowanie

- ▶ Kiedy zachowanie obiektu powinno zależeć od jego stanu
- ▶ Kiedy wykonywane operacje są długie i składają się z wielu instrukcji warunkowych

### ▶ Konsekwencje

- ▶ Separacja zachowania i stanu
- ▶ Możliwość współużytkowania stanu obiektu
- ▶ Zmiana stanu jest jawna

# STATE, OBIEKTOWY, BEHAWIORALNY



# STRATEGY, OBIEKTOWY, BEHAWIORALNY

### ▶ Przeznaczenie

- ▶ Pozwala zidentyfikować wiele algorytmów rozwiązania tego samego problemu i w zależności od zachodzącej potrzeby je podmieniać
- ▶ Pozwala stosować różne algorytmy dla różnych klientów

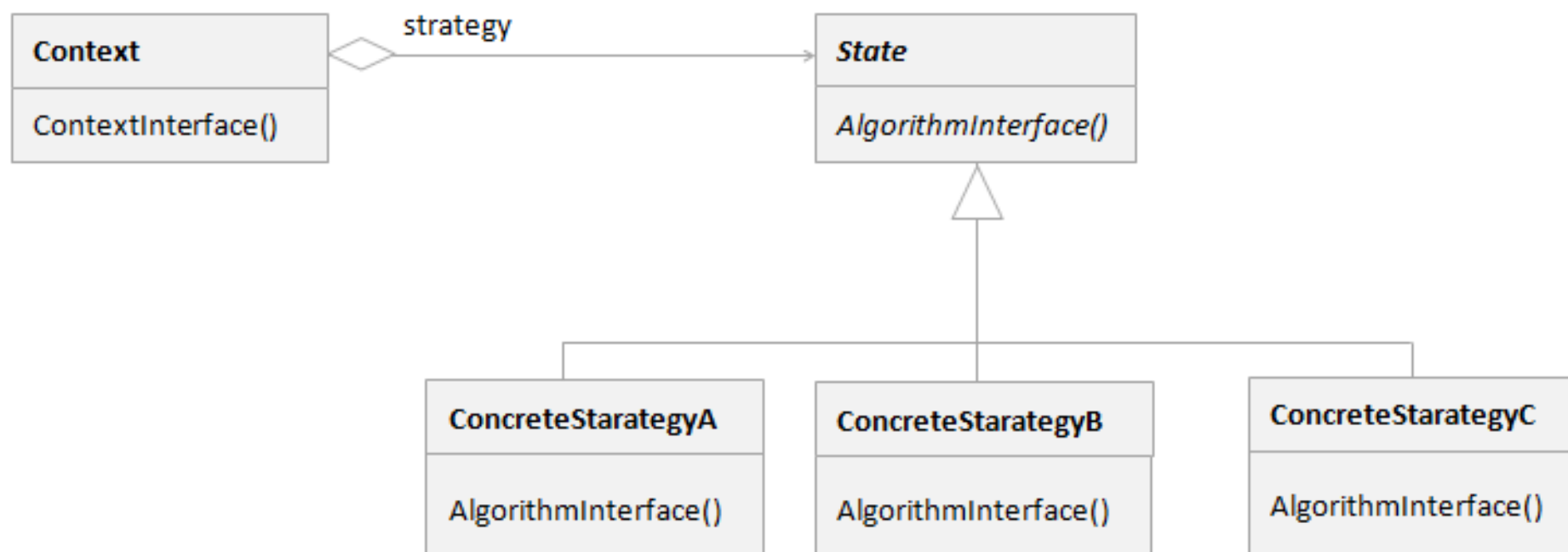
### ▶ Zastosowanie

- ▶ Kiedy wiele powiązanych klas różni się tylko zachowaniem
- ▶ Kiedy trzeba zastosować różne implementacje jednego algorytmu
- ▶ Kiedy zależy na ukryciu złożoności algorytmu lub danych na których on działa
- ▶ Kiedy klasa definiuje wiele zachowań w postaci złożonych instrukcji warunkowych

### ▶ Konsekwencje

- ▶ Algorytmy zgrupowane w rodziny
- ▶ Łatwość podmiany algorytmu
- ▶ Niskie sprzężenie z konkretną implementacją algorytmu
- ▶ Eliminacja instrukcji warunkowych

# STRATEGY, OBIKTOWY, BEHAWIORALNY





# TEMPLATE METHOD, KLASOWY, BEHAWIORALNY

## ▶ Przeznaczenie

- ▶ Definiuje szkielet algorytmu delegując jego kroki do metod w podklasach (możliwa jest zmiana poszczególnych kroków ale nie ich sekwencji)

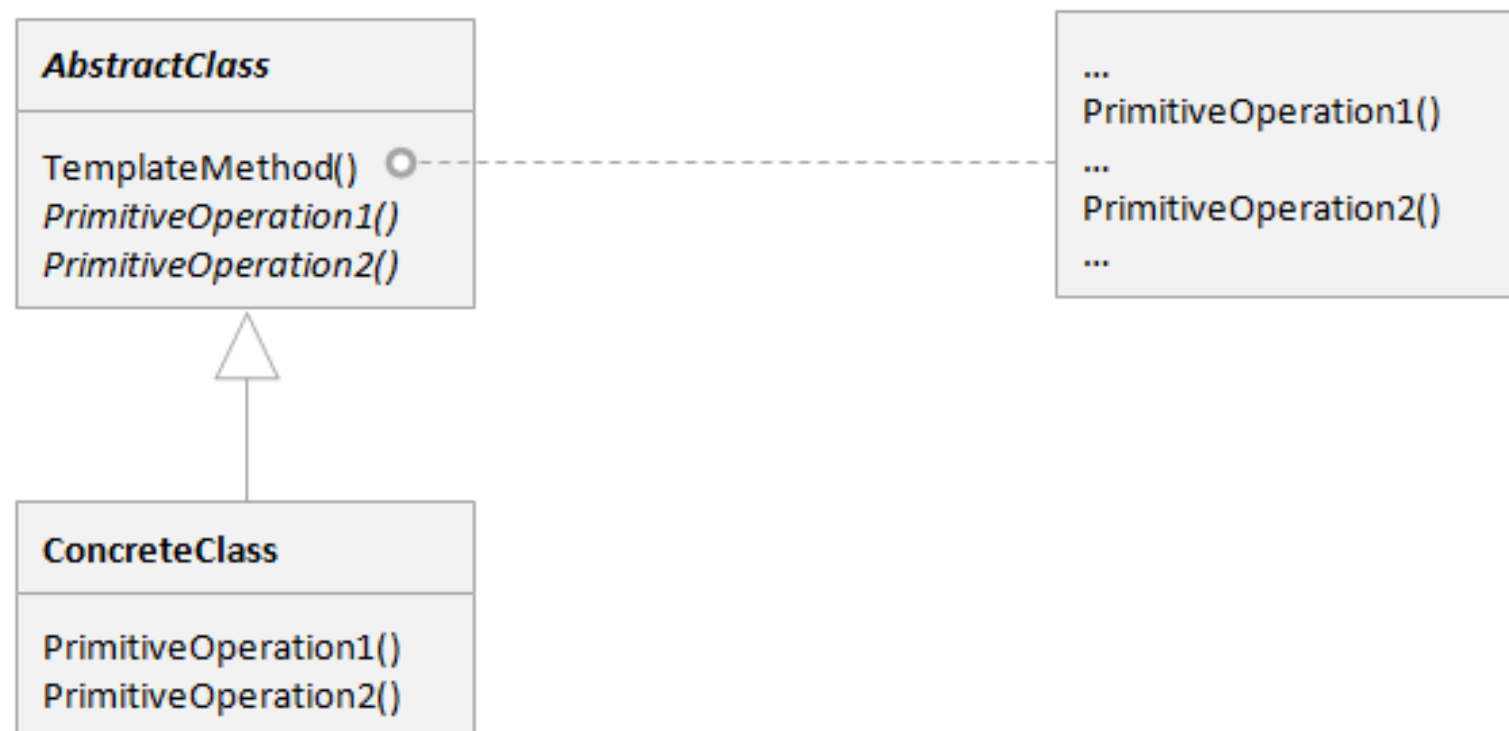
## ▶ Zastosowanie

- ▶ Kiedy trzeba zaimplementować jednorazowo niezmienną część algorytmu i umożliwić implementowanie zmiennych części w podklasach
- ▶ Kiedy zachowanie wspólne dla podklas należy wyodrębnić i umieścić w jednej klasie aby uniknąć duplikacji kodu
- ▶ Kiedy należy kontrolować rozszerzalność klas

## ▶ Konsekwencje

- ▶ Redukcja duplikacji kodu
- ▶ Odwrócenie struktury sterowania
- ▶ Niezmiennność kroków algorytmu

# TEMPLATE METHOD, KLASOWY, BEHAWIORALNY



# VISITOR, OBIEKTOWY, BEHAWIORALNY

### ▶ Przeznaczenie

- ▶ Reprezentuje operacje wykonywane na elementach struktury obiektów
- ▶ Umożliwia definiowanie nowych operacji bez konieczności zmian w klasach elementów na których działa

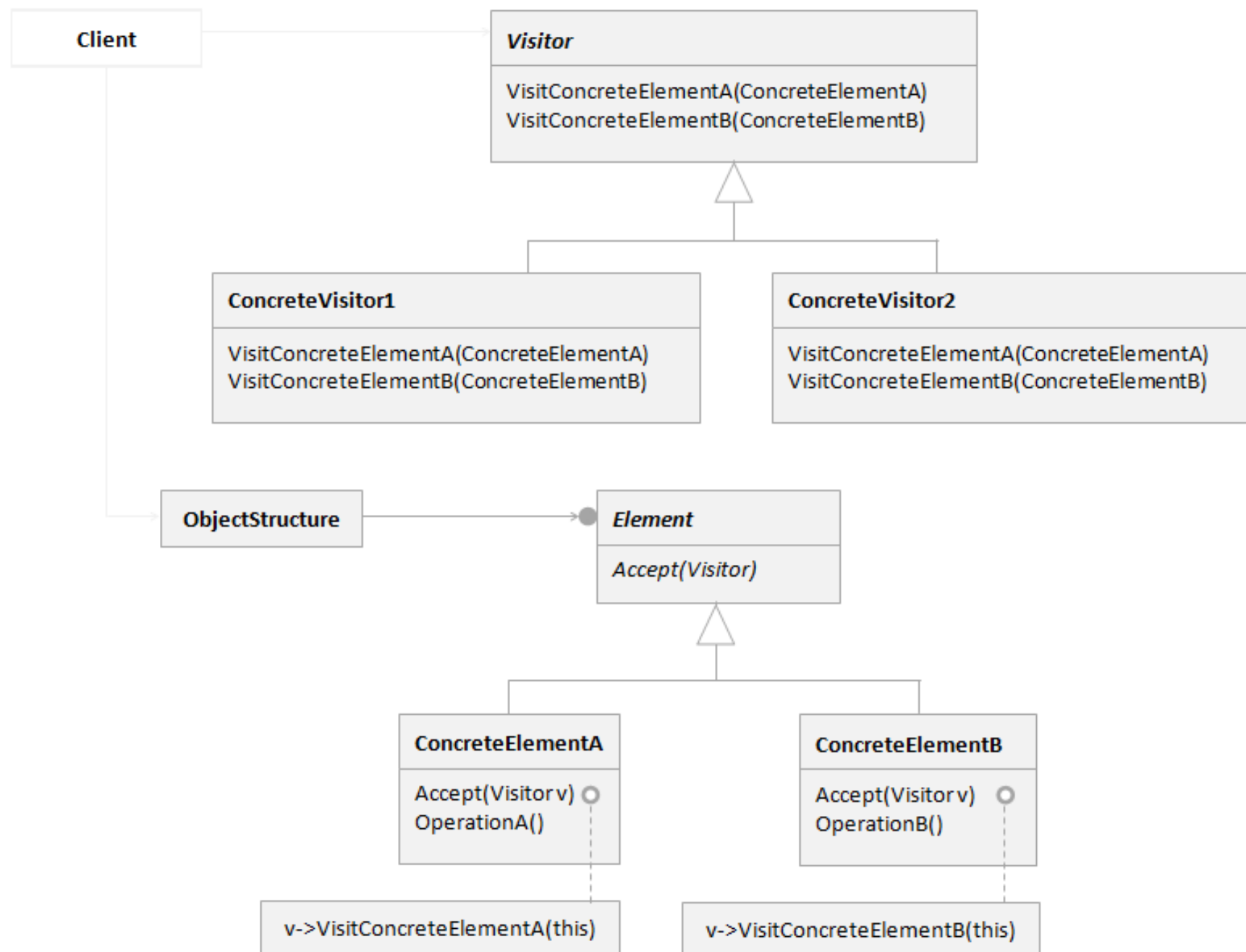
### ▶ Zastosowanie

- ▶ Kiedy struktura obiektów obejmuje wiele klas i trzeba na nich wykonać operacje zależne od typu elementu
- ▶ Kiedy na klasach struktury trzeba wykonać wiele niepowiązanych z nimi operacji bez ich zaśmiecania
- ▶ Kiedy klasy struktury rzadko się zmieniają, ale często definiowane są operacje na nich wykonywane

### ▶ Konsekwencje

- ▶ Łatwe dodawanie nowych operacji
- ▶ Trudne dodawanie nowych elementów struktury
- ▶ Możliwość odwiedzania różnych hierarchii tym samym gościem
- ▶ Możliwość grupowania powiązanych operacji

## VISITOR, OBIKTOWY, BEHAWIORALNY



# TEST DRIVEN DEVELOPMENT

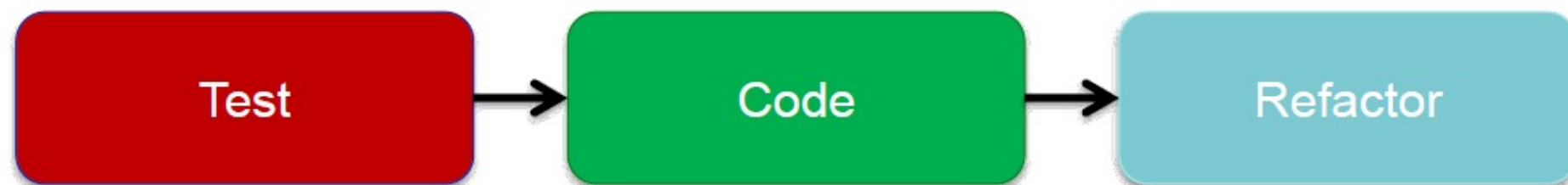
---

# TEST DRIVEN DEVELOPMENT

- ▶ Ewolucyjne podejście do tworzenia oprogramowania wywodzące się z eXtreme Programming
- ▶ Technika programistyczna prowadząca do prostego kodu wysokiej jakości

# TDD W PRAKTYCE

- ▶ Programista implementuje kolejne funkcjonalności pracując iteracyjnie zgodnie z poniższym cyklem
  - ▶ Dodanie / uruchomienie testu
  - ▶ Stworzenie implementacji spełniającej test
  - ▶ Budowanie jakości przez refaktoryzację



# TEST JEDNOSTKOWY W PODEJŚCIU TDD

- ▶ Determinuje API
- ▶ Określa kontekst rozwiązania
- ▶ Weryfikuje poprawność
- ▶ Umożliwia bezpieczną refaktoryzację
- ▶ Wymusza tworzenie prostego i potrzebnego kodu
- ▶ Pokazuje postęp prac
- ▶ Dokumentuje sposób działania systemu



# CECHY DOBRYCH TESTÓW

- ▶ Szybkie
- ▶ Spójne
- ▶ Izolowane
- ▶ Powtarzalne
- ▶ Celowe

# TECHNIKI TWORZENIA TESTOWALNEGO KODU

- ▶ Stosowanie dobrych praktyk związanych z programowaniem np.:
  - ▶ Programowanie przez interfejsy
  - ▶ Single Responsibility Principle
  - ▶ Inversion of control / Dependency injection
  - ▶ Unikanie duplikacji
  - ▶ Wzorce projektowe
  - ▶ Programowanie przez zdarzenia

# IMPLEMENTACJA W PODEJŚCIU TDD

- ▶ Kod spełniający wymagania określone testami

# REFAKTORYZACJA W PODEJŚCIU TDD

- ▶ Ulepszanie kodu / projektu bez zmiany jego funkcjonalności
- ▶ Umożliwia systematyczne poprawianie jakości rozwiązania
- ▶ Przykłady
  - ▶ Usuwanie duplikacji
  - ▶ Zmniejszanie sprzężenia
  - ▶ Poprawianie podziału odpowiedzialności
  - ▶ Wprowadzanie wzorców projektowych

# CODE SMELLS

- ▶ Sygnalizują konieczność rewizji kodu / refaktoryzacji
- ▶ Przykłady
  - ▶ Długie metody
  - ▶ Duże klasy
  - ▶ Wiele argumentów metody/konstruktor
  - ▶ Nadmierna złożoność
  - ▶ Duplikacja

# KATALOG REFAKTORYZACJI

- ▶ Klasyfikuje typowe problemy związane z jakością kodu oraz sposoby ich naprawy
- ▶ „Refactoring: Improving the Design of Existing Code” M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts

# ISTOTNE ELEMENTY TDD

- ▶ Wykonywanie małych kroków (podejście iteracyjne)
- ▶ Ewolucyjne budowanie rozwiązania
- ▶ Bezpieczeństwo
- ▶ Nieustanne poprawianie jakości

# TDD W PRAKTYCE - PARSER WYRAŻEŃ

- ▶ Wyrażenia umieszczane w tekście mają postać  $\${nazwa}$
- ▶ W czasie ewaluacji wyrażenia powinny zostać zmienione na konkretne wartości
- ▶ Ewaluacja wyrażenia bez podania wartości powinna kończyć się wyjątkiem
- ▶ Wartości nadmiarowe powinny być ignorowane



# TDD AS IF YOU MEANT IT

- ▶ Tworzone testy powinny być jak najmniejsze i zbliżające do rozwiązania założonego problemu
- ▶ Kod produkcyjny może powstawać wyłącznie w celu spełnienia testu (jak najprostsza implementacja, początkowo w metodzie testu)
- ▶ W miarę możliwości powinna być przeprowadzana refaktoryzacja prowadząca do redukcji duplikacji i poprawy rozwiązania
- ▶ Tworzenie nowych metod (nie testowych) może się odbywać wyłącznie w wyniku refaktoryzacji - ekstrakcji metody do klasy testu lub innych, istniejących klas
- ▶ Tworzenie nowych klas może się odbywać wyłącznie w wyniku refaktoryzacji - przeniesienia metody / grupowania metod z klasy testowej lub innych, istniejących klas

# SILNIK GRY KÓŁKO I KRZYŻYK

- ▶ Plansza ma rozmiar 3 x 3 pola
- ▶ Gracze zajmują naprzemian wolne pola stawiając na nich swój znak (kółko lub krzyżyk)
- ▶ Gra kończy się kiedy wszystkie pola zostaną zajęte lub jeden z graczy zajmie sekwencję wygrywającą (kolumna, wiersz lub przekątna)

# SPECIFICATION BY EXAMPLE

- ▶ TDD umożliwia tworzenie jakościowych rozwiązań na poziomie kodu jednak nie chroni przed implementacją niewłaściwych / zbędnych funkcjonalności
- ▶ Programowanie behawioralne polega na tłumaczeniu zebranych wymagań w zbiór wykonywalnych testów, które następnie mogą zostać zaimplementowane z użyciem TDD
- ▶ Wspierane przez narzędzia: Cucumber, xBehave, Concordion, Fit/Fitnesse oraz inne

## SPECIFICATION BY EXAMPLE

### 1. Write story

Plain  
text

Scenario: A trader is alerted of status

Given a stock and a threshold of 15.0

When stock is traded at 5.0

Then the alert status should be OFF

When stock is traded at 16.0

Then the alert status should be ON

### 2. Map steps to Java

POJO

```
public class TraderSteps {  
    private TradingService service; // Injected  
    private Stock stock; // Created  
  
    @Given("a stock and a threshold of $threshold")  
    public void aStock(double threshold) {  
        stock = service.newStock("STK", threshold);  
    }  
    @When("the stock is traded at price $price")  
    public void theStockIsTraded(double price) {  
        stock.tradeAt(price);  
    }  
    @Then("the alert status is $status")  
    public void theAlertStatusIs(String status) {  
        assertThat(stock.getStatus().name(), equalTo(status));  
    }  
}
```

## SPECIFICATION BY EXAMPLE

### 3. Configure Stories

```
public class TraderStories extends JUnitStories {  
  
    public Configuration configuration() {  
        return new MostUsefulConfiguration()  
            .useStoryLoader(new LoadFromClasspath(this.getClass()))  
            .useStoryReporterBuilder(new StoryReporterBuilder()  
                .withCodeLocation(codeLocationFromClass(this.getClass()))  
                .withFormats(CONSOLE, TXT, HTML, XML));  
    }  
  
    public List<CandidateSteps> candidateSteps() {  
        return new InstanceStepsFactory(configuration(),  
            new TraderSteps(new TradingService())).createCandidateSteps();  
    }  
  
    protected List<String> storyPaths() {  
        return new StoryFinder().findPaths(codeLocationFromClass(this.getClass()),  
            "**/*.story");  
    }  
}
```

Only  
once

### 4. Run Stories



With  
any of

### 5. View Reports

HTML

#### Scenario: A trader is alerted of status

Given a stock and a threshold of 15.0  
When stock is traded at 5.0  
Then the alert status is OFF  
When stock is traded at 16.0  
Then the alert status is ON

# HISTORIE I SCENARIUSZE

- ▶ Mają wysoki poziom abstrakcji
- ▶ Są wykonywalne (testy jako specyfikacja)
- ▶ Stanowią „żywą” dokumentację systemu
- ▶ Powinny być niezależne od technologii
- ▶ Muszą być rozumiane przez wszystkich udziałowców
- ▶ Ukazują postęp dotychczasowych prac oraz określają funkcjonalność

# HISTORIA

- ▶ Opisuje wymagania dotyczące funkcjonalności systemu
- ▶ Powstaje przy współpracy różnych udziałowców
- ▶ Składa się z
  - ▶ Tytułu
  - ▶ Narracji
  - ▶ Kryteriów akceptacji (scenariuszy)

# PRZYKŁADOWY SZABLON HISTORII

Title - tytuł krótko streszcza historyjkę

Narration - rola, opis funkcjonalności, cele

As a [ROLE]

I want [FEATURE]

So that [BENEFIT]

Kryteria akceptacji zapisane jako warianty scenariuszy, określają kiedy historia jest kompletna

Scenario 1: Title

Given [context] And [some more context]...

When [event]

Then [outcome] And [another outcome]...



# PRZYKŁADOWA HISTORIA

Wypłata gotówki z bankomatu

Narracja:

Aby mieć dostęp do środków, kiedy bank jest nieczynny  
jako właściciel konta  
chcę mieć możliwość wypłaty pieniędzy z bankomatu

# SCENARIUSZ 1

Scenariusz: Na koncie są wystarczające środki

Jeśli bilans konta wynosi 100 zł  
oraz w bankomacie jest 1000 zł  
kiedy właściciel wypłaci 50 zł  
wtedy bilans konta wyniesie 50 zł  
oraz w bankomacie zostanie 950 zł

# SCENARIUSZ 2

Scenariusz: Na koncie nie ma wystarczających środków

Jeśli bilans konta wynosi 100 zł  
oraz w bankomacie jest 1000 zł  
kiedy właściciel spróbuje wypłacić 150 zł  
wtedy bilans konta wyniesie 100 zł  
oraz w bankomacie zostanie 1000 zł

# KORZYŚCI

- ▶ Spójna komunikacja między osobami zaangażowanymi w projekt
- ▶ „Żywa”, zawsze aktualna dokumentacja
- ▶ Możliwość monitorowania postępów prac oraz poprawności zaimplementowanych części systemu
- ▶ Nacisk na to co, a nie jak ma to być zrobione