
REFAKTORYZACJA DO WZORCÓW PROJEKTOWYCH

REFAKTORYZACJA

- ▶ Poprawianie / ulepszanie istniejącego kodu bez zmiany jego zachowania / istniejącej funkcjonalności
- ▶ Ciągłe podnoszenie jakości kodu przekładające się na jego lepsze zrozumienie, mniejszą ilość potencjalnych błędów, promowanie dobrych praktyk
- ▶ Powinno być realizowane systematycznie, przez wszystkich członków zespołu, w oparciu o testy, w rozsądnie określonym zakresie

PROCES

- ▶ Estymacja czasu
- ▶ Szacowanie korzyści i kosztów
- ▶ Dbłość o bezpieczeństwo zmian
- ▶ Wymuszanie / budowanie kultury jakości

PODSTAWOWE DEFINICJE ZWIĄZANE LOGIKĄ

- ▶ **Reguły biznesowe** definiują pojęcia i polityki niezbędne dla działania biznesu
- ▶ **Proces biznesowy** to seria powtarzalnych kroków, wykonywanych przez organizację, w celu uzyskania pożądanego efektu (celu biznesowego)
- ▶ **Logika biznesowa** to część aplikacji, odpowiedzialna za realizację przyjętych reguł biznesowych

POPRAWNA IMPLEMENTACJA LOGIKI BIZNESOWEJ

- ▶ Opiera się o stosowanie praktyk, prowadzących do czystego kodu oraz czystej architektury
- ▶ Większość z nich daje się uogólnić do działań zapewniających **niskie sprzężenie (low coupling)** i **wysoką spójność (high cohesion)**

KOD NISKIEJ JAKOŚCI - TYPOWE PRZYCZYNY

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

KOD NISKIEJ JAKOŚCI - KONSEKWENCJE

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

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 - ANTYWZORCE

- ▶ Nazwy niezgodne z rzeczywistością
- ▶ Nazwy nieznane, wymyślane, przekręcone
- ▶ Nazwy zawierające liczby
- ▶ Nazwy jednoliterowe

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

- ▶ Znacząco poprawia czytelność
- ▶ Powinno być spójne na poziomie projektu (style guide, użycie narzędzi)
- ▶ Dotyczy między innymi:
 - ▶ stosowania wcięć i separatorów w postaci pustej linii
 - ▶ odpowiedniego łamania linii / instrukcji
 - ▶ długości linii
 - ▶ konwencji nazewniczych
 - ▶ kolejności deklaracji

KOMENTARZE

- ▶ Komentarze powinny:
 - ▶ być wyjątkiem od reguły - 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 czy prawa autorskie. Powinny one być ograniczone do niezbędnego minimum

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 określonego zadania
 - ▶ zachowywać się jak czarne skrzynki - powinny ukrywać dane i dostarczać operacje, które pozwalają na nich bezpiecznie operować (enkapsulacja i ukrywanie danych)
 - ▶ być skonfigurowane zależnościami podawanymi z zewnątrz

OBSŁUGA BŁĘDÓW / SYTUACJI WYJĄTKOWYCH

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

SPRZĘŻENIE (ANG. COUPLING)

- ▶ Określa siłę zależności między współpracującymi jednostkami (metody, klasy, komponenty, moduły...)
- ▶ Należy dążyć do małego sprzężenia - zależność tylko w zakresie niezbędnym do realizacji założonego celu

SPÓJNOŚĆ (ANG. COHESION)

- ▶ Określa jak bardzo dana jednostka (metoda, klasa, komponent, moduł...) skupia się na realizacji określonego celu
- ▶ Należy dążyć do wysokiej spójności / specjalizacji

INTERFEJS / API

- ▶ Abstrakcja definiująca kontrakt między współpracującymi elementami (obiektami, komponentami, modułami, usługami, mikroserwisami)
- ▶ Określa możliwe sposoby interakcji
- ▶ Ukrywa szczegóły implementacyjne

SOLID PRINCIPLES

- ▶ Zbiór 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

INNE POPULARNE ZASADY

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

INWERSJA KONTROLI (IOC)

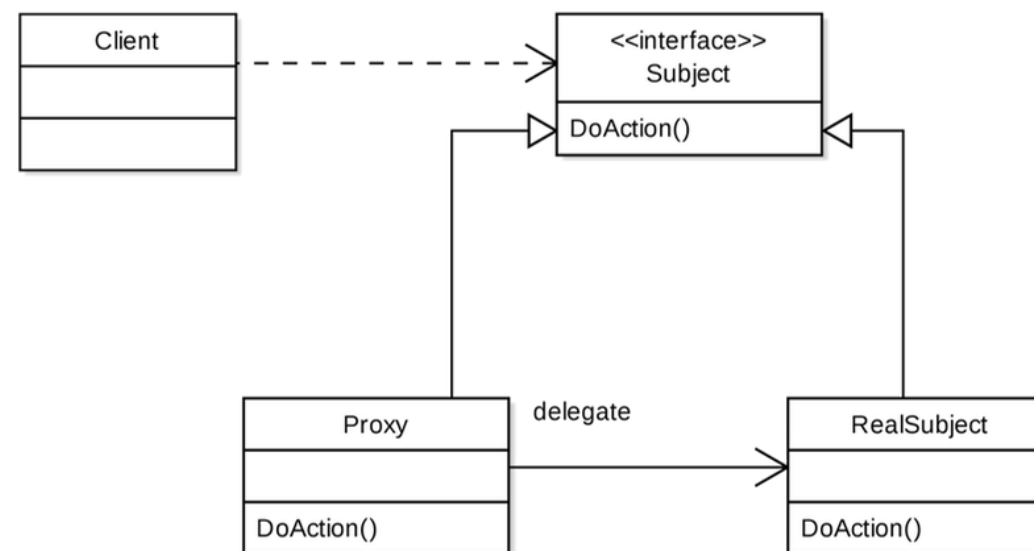
- ▶ Odwrócenie sterowania wykonania programu / przeniesienie na zewnątrz odpowiedzialności za kontrolę wykonania

WSTRZYKIWANIE ZALEŻNOŚCI

- ▶ Tworzenie, konfigurowanie i podawanie zależności „z zewnątrz”
- ▶ Zmniejsza sprzężenie między elementami aplikacji, co m.in. daje swobodę wymiany implementacji i ułatwia testowanie
- ▶ Przykłady realizacji:
 - ▶ Ręczne podawanie zależności np. przez konstruktor, metody
 - ▶ Wzorce projektowe np. **Factory Method**
 - ▶ Kontener zarządzający cyklem życia komponentów (**Spring, CDI**)

SEPARACJA Z UŻYCIEM PROXY

- ▶ Umożliwia utworzenie obiektu zastępczego / pośredniego dla innego obiektu
- ▶ Pozwala kontrolować dostęp do oryginalnego obiektu oraz opakowywać wołaną metodę dodatkową logiką



PROGRAMOWANIE ASPEKTOWE

- ▶ Uzupełnia paradygmat programowania obiektowego
- ▶ Umożliwia oddzielenie logiki biznesowej od dodatkowych zadań pobocznych, takich jak: transakcje, logowanie, bezpieczeństwo
- ▶ Często realizowane z wykorzystaniem wzorca **Proxy**

PROGRAMOWANIE PRZEZ ZDARZENIA

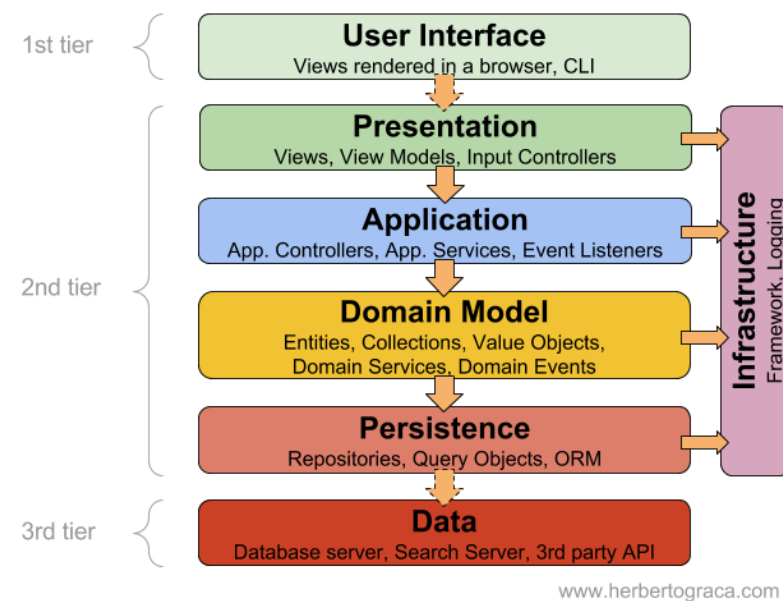
- ▶ Umożliwia rozluźnienie powiązań - komponenty mogą się komunikować mimo, że niewiele o sobie wiedzą
- ▶ Opiera się o wzorzec **Observer**

ARCHITEKTURA

- ▶ Definiuje najważniejsze komponenty, zakres ich odpowiedzialności, a także wzajemne relacje
- ▶ Stanowi szablon rozwiązania
- ▶ Może być definiowana na różnym poziomie np. aplikacja, system

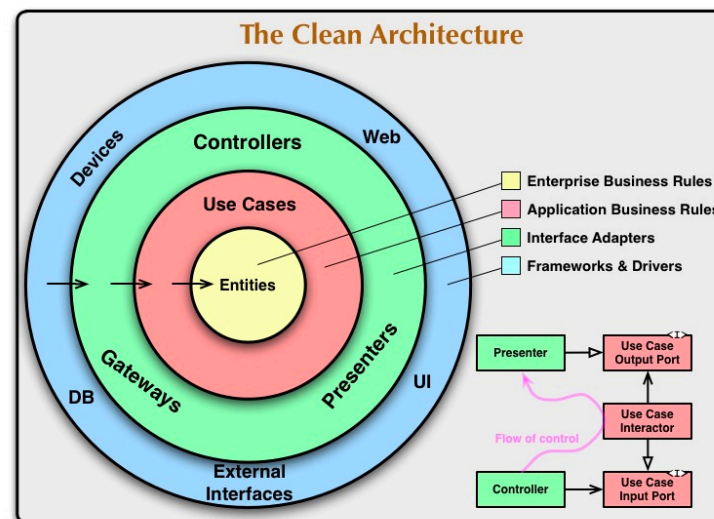
ARCHITEKTURA WARSTWOWA (N-TIER PATTERN)

- ▶ Komponenty aplikacji organizowane są w warstwy pełniące określoną rolę np. prezentacja, logika, utrwalanie danych (podział techniczny)
- ▶ Komunikacja między warstwami odbywa się w jednym kierunku



CZYSTA ARCHITEKTURA

- ▶ Centralnym elementem aplikacji jest logika biznesowa, zaimplementowana w sposób niezależny od bibliotek, frameworków czy użytej infrastruktury
- ▶ Wokół logiki biznesowej tworzone są kolejne, bardziej wysokopoziomowe warstwy m.in. warstwa adapterów umożliwiająca komunikację ze światem zewnętrznym



DOMAIN DRIVEN DESIGN

- ▶ Podejście do tworzenia oprogramowania, które kładzie nacisk na to, aby obiekty i ich zachowania wiernie odzwierciedlały rzeczywistość / domenę problemu

BOUNDED CONTEXT

- ▶ Koncepcja zakładająca podział aplikacji na konteksty, definiujące własny model dziedzinowy, odwzorowujące konkretne potrzeby, warunki i procesy np. płatności, wyszukiwanie produktów, realizacja zamówień
- ▶ Komunikacja między kontekstami odbywa się za pośrednictwem dobrze określonego interfejsu, a zmiany modelu wewnętrznego lub logiki nie powinny mieć na nią wpływu

COMMAND QUERY SEPARATION

- ▶ Zasada przedstawiona przez w 1986 roku przez Bertranda Meyera, mówiąca o tym, że każda metoda powinna być zaklasyfikowana do jednej z grup:
 - ▶ **Command** - metody, które zmieniają stan aplikacji i nic nie zwracają
 - ▶ **Query** - metody, które coś zwracają, ale nie zmieniają stanu aplikacji

MONOLIT

- ▶ Aplikacja rozwijana, testowana i wdrażana jako całość (single artifact)
- ▶ Zalety (do pewnego rozmiaru projektu)
 - ▶ Łatwa implementacja nowych funkcjonalności biznesowych i pobocznych
 - ▶ Mała złożoność infrastrukturalna
- ▶ Wyzwania (od pewnego rozmiaru projektu)
 - ▶ Trudność utrzymania i rozwoju ze względu na rosnącą złożoność, zakres funkcjonalności i rozmiar aplikacji
 - ▶ Ograniczona skalowalność
 - ▶ Przywiązanie do określonych rozwiązań i technologii

MODULARNY MONOLIT

- ▶ Aplikacja rozwijana, testowana i wdrażana jako zbiór modułów, które:
 - ▶ są od siebie niezależne
 - ▶ mają dobrze zdefiniowaną odpowiedzialność
 - ▶ posiadają publiczny interfejs / kontrakt
 - ▶ mogą być reużywane w innych aplikacjach

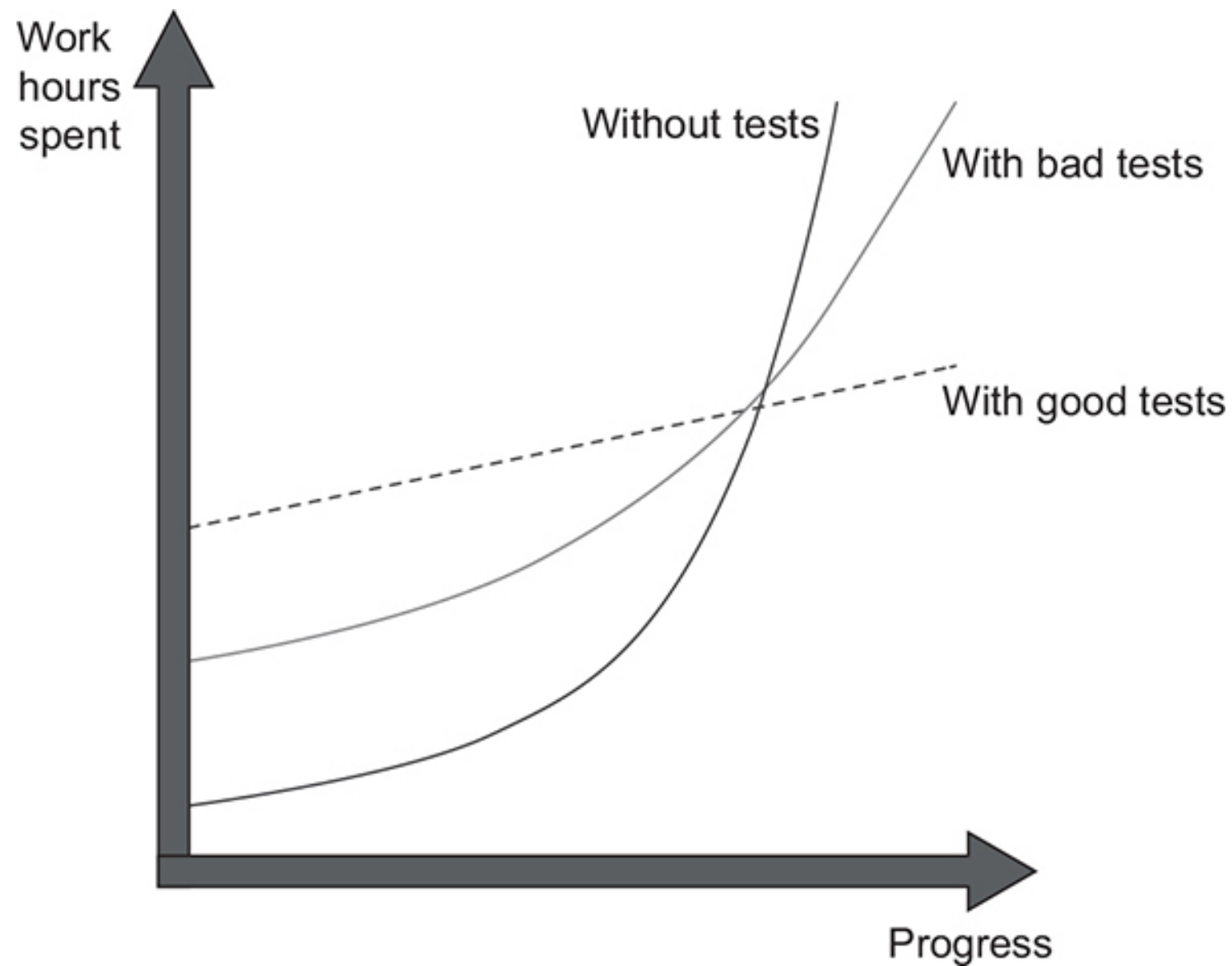
KONTEKST MA ZNACZENIE

- ▶ Każdy problem, proces, firma jest inna, dlatego za każdym razem należy do dostosować podejście - jeśli coś działa w jednym przypadku, nie oznacza to, że zadziała w innym

DLACZEGO TESTOWANIE JEST WAŻNE?

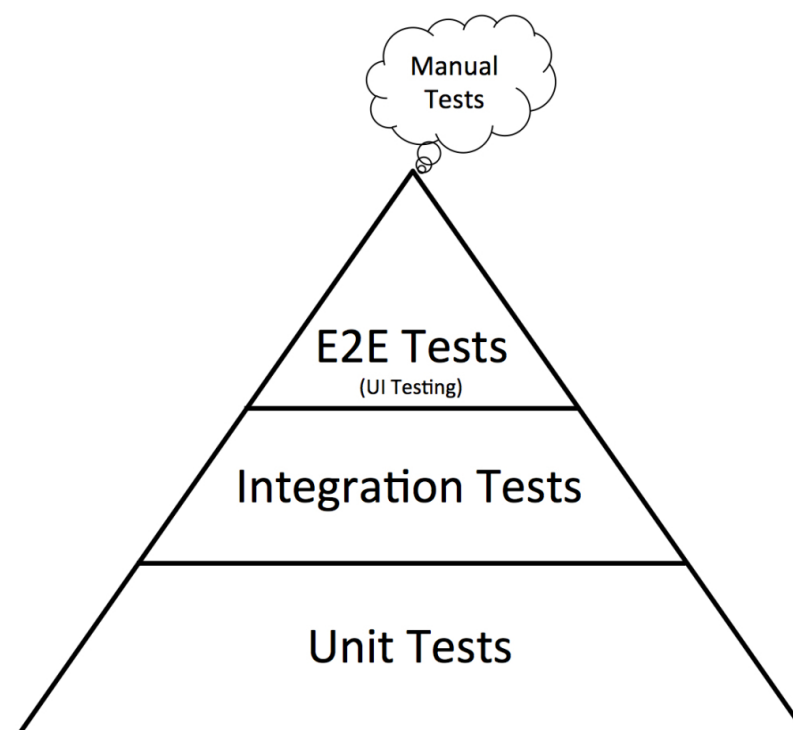
- ▶ Zmiany na poziomie kodu prowadzą do wzrostu jego entropii
- ▶ Bez odpowiedniej opieki (systematyczne sprzątanie, refaktoryzacja) system staje się coraz bardziej złożony, chaotyczny, podatny na błędy i trudny w utrzymaniu
- ▶ Poprawne testy stanowią ochronę przed regresją (siatka bezpieczeństwa) oraz umożliwiają skalowalność/stabilny rozwój oprogramowania

TESTY TO DŁUGOTERMINOWA INWESTYCJA



PIRAMIDA TESTÓW

- ▶ **Testy jednostkowe** - (udział ~70%) weryfikują działanie izolowanych jednostek kodu
- ▶ **Testy integracyjne** - (udział ~20%) weryfikują współdziałanie komponentów oraz integrację z zewnętrznymi bibliotekami i/lub systemami
- ▶ **Testy E2E** - (udział ~10%) weryfikują działanie aplikacji z punktu widzenia użytkownika



JAKOŚĆ TESTÓW MA ZNACZENIE

- ▶ Tworzenie testów jest dziś naturalną częścią procesu wytwarzania oprogramowania
- ▶ Nie wszystkie testy mają taką samą wartość, a ilość testów nie zawsze przekłada się większą jakością
- ▶ Kod testów także wymaga ciągłego utrzymania i dbałości o jakość (zwykle stosunek kodu produkcyjnego do testów to 1:1 - 1:3)

TESTY A JAKOŚĆ KODU

- ▶ Trudności związane z napisaniem testów często wynikają ze złej jakości kodu i/lub architektury (są dobrym wskaźnikiem negatywnym)
- ▶ Możliwość napisania testów, przetestowania kodu nie zawsze oznacza jego dobrą jakość

ZASADY TWORZENIA CZYSTEGO I TESTOWALNEGO KODU

- ▶ Sprowadza się do stosowania praktyk związanych z tworzeniem czystego kodu (innymi słowy czysty kod jest dużo bardziej testowalny)
 - ▶ Pilnowanie dobrego podziału odpowiedzialności
 - ▶ Unikanie silnego sprzężenia
 - ▶ Stosowanie wzorców projektowych
 - ▶ Pisanie czytelnego kodu
 - ▶ ...

ATRYBUTY DOBRYCH TESTÓW

- ▶ Zapewniają ochronę przed regresją
- ▶ Są odporne na zmiany i refaktoryzację
- ▶ Są zrozumiałe i łatwe w utrzymaniu
- ▶ Dają szybki i godny zaufania feedback zwrotny

TEST COVERAGE

- ▶ Dostarcza ogólną informację jak duża część kodu jest pokryta testami
- ▶ Pozwala odnaleźć nietestowane fragmenty kodu
- ▶ Małe pokrycie testami jest sygnałem, że coś jest nie tak (nie działa w drugą stronę)

TEST COVERAGE

- ▶ Przykładowe sposoby pomiarów:

$$\text{Code coverage (test coverage)} = \frac{\text{Lines of code executed}}{\text{Total number of lines}}$$

$$\text{Branch coverage} = \frac{\text{Branches traversed}}{\text{Total number of branches}}$$

- ▶ Wysokie wskaźniki pokrycia nie gwarantują tego, że kod został poprawnie przetestowany, a jedynie, że w pewnym momencie został wykonany

TESTY JEDNOSTKOWE

- ▶ Weryfikują poprawność izolowanych jednostek kodu
- ▶ Szkoła classic (Detroit)
 - ▶ Jednostka kodu to zachowanie/feature - w ramach testu może być weryfikowanych kilka „prawdziwych” klas jednocześnie
 - ▶ Tylko zależności współdzielone między testami np. baza danych, system plików powinny być zastąpione mockami
- ▶ Szkoła mockist (London)
 - ▶ Jednostka kodu to wyizolowana klasa
 - ▶ Zależności z wyjątkiem tych niezmiennych (value objects) powinny być zastąpione mockami

CECHY DOBRYCH TESTÓW JEDNOSTKOWYCH

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

FRAMEWORK xUNIT

- ▶ Odpowiedzialny za:
 - ▶ przygotowanie testów
 - ▶ wykonywanie testów
 - ▶ posprzątanie po testach
 - ▶ raportowanie o błędach i statystykach wykonania

KONSTRUKCJA TESTU

- ▶ Zwykle test przyjmuje formę Given-When-Then lub Arrange-Act-Assert (3A) co można rozumieć jako:
 - ▶ Przygotowanie kontekstu
 - ▶ Wykonanie operacji/działania
 - ▶ Weryfikację wyników
- ▶ Biblioteka udostępnia zbiór predefiniowanych asercji w postaci statycznych metod np. `assertEquals`, `assertArrayEquals`, `assertTrue`, `assertFalse`, `assertNull`
Niespełnienie asercji kończy się wyrzuceniem wyjątku, a tym samym obłaniem testu

MATCHERS

- ▶ Umożliwia 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>

OBIEKTY ZASTĘPCZE (TEST DOUBLES)

- ▶ Zastępują zewnętrzne zależności i umożliwiają testowanie w pełnej izolacji
- ▶ Implementowane ręcznie lub generowanie z użyciem zewnętrznych bibliotek np. [Mockito](#)

KLASYFIKACJA OBIEKTÓW ZASTĘPCZYCH

- ▶ **Dummy** - wartość wymagana do przeprowadzenia testu, najczęściej nie ma wpływu na jego wynik
- ▶ **Stub** - obiekt / funkcja zwracająca predefiniowaną wartość dla określonych parametrów wejściowych, może służyć do redukcji efektów ubocznych
- ▶ **Fake** - lekka implementacja, naśladująca działanie prawdziwej zależności
- ▶ **Spy** - rejestruje interakcje i przekazywane parametry (działa jak proxy)
- ▶ **Mock** - programowalny obiekt, zachowujący się zgodnie z narzuconymi oczekiwaniami / kontraktem

STYLE TESTÓW JEDNOSTKOWYCH




- ▶ **Output-based testing** - weryfikacja zwracanego rezultatu
- ▶ **State-based testing** - weryfikacja stanu po zakończeniu operacji
- ▶ **Communication-based testing** - weryfikacja interakcji między testowanymi obiektami i ich zależnościami

TESTOWANIE BLACK BOX VS. WHITE BOX

- ▶ **Black box testing** - weryfikacja na poziomie kontraktu, bez znajomości wewnętrznych struktur/szczegółów implementacyjnych (co, a nie jak)
- ▶ **White box testing** - weryfikacja z wykorzystaniem wewnętrznych struktur/szczegółów implementacyjnych

JAK I CO TESTOWAĆ?

- ▶ <https://speakerdeck.com/skmetz/magic-tricks-of-testing-ancientcityruby>

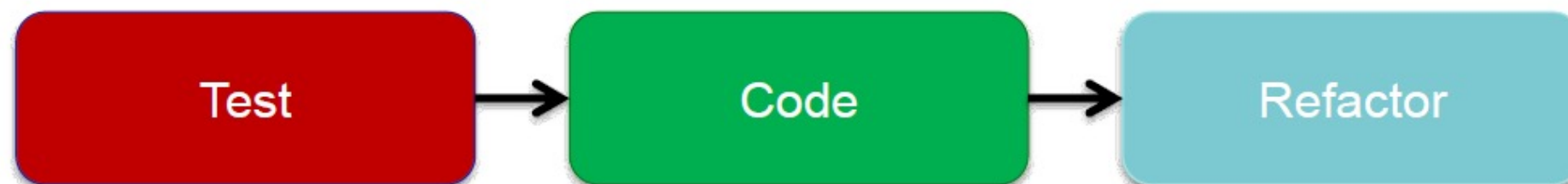
Message	Type	Query	Command
POV 		Assert <i>result</i>	Assert <i>direct public side effects</i>
	Ignore		
			
			Expect <i>to send</i>

TEST DRIVEN DEVELOPMENT

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

TDD W PRAKTYCE

- ▶ Programista implementuje kolejne funkcjonalności pracując iteracyjnie, zgodnie z poniższym cyklem:
 - ▶ dodanie / uruchomienie / oblanie testu
 - ▶ stworzenie implementacji spełniającej test
 - ▶ budowanie jakości poprzez 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

IMPLEMENTACJA W PODEJŚCIU TDD

- ▶ Kod spełniający wymagania określone testami

REFAKTORYZACJA W PODEJŚCIU TDD

- ▶ Ulepszanie kodu / projektu bez zmiany jego funkcjonalności
- ▶ Umożliwia systematyczne budowanie 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

TDD - SUGESTIE

- ▶ Tworząc test spróbuj rozpocząć od asercji / warunku sprawdzającego
- ▶ Zaczynij od najprostszego możliwego testu i niebój się robić dzieciennie małych kroków (np. wartość zwracana wprost)
- ▶ Opisuj problem za pomocą przykładu (give, when, then)
- ▶ Wybieraj testy które poszerzają wiedzę, pokazują problem z różnych perspektyw
- ▶ Testy obejmujące negatywne / niepoprawne scenariusze są bardzo ważne, ale najczęściej mniej istotne z punktu widzenia designu, dlatego mogą być tworzone później

TDD - SUGESTIE

- ▶ Używaj sensownych danych (real data, different data)
- ▶ Nie wyciągaj zbyt szybko wniosków (false positives)
- ▶ Ucz się przez testy (test driven learning :))
- ▶ Zajmuj się tylko jedną rzeczą naraz - nowe pomysły, problemy zapisuj i wróć do nich później
- ▶ Tam gdzie możesz stosuj black box (classical) testing vs. white box (mockist) testing

TDD - SUGESTIE

- ▶ Unikaj zbyt dużej szczegółowości / chirurgicznej precyzji (overspecification)
- ▶ Utrzymuj jakość i czytelność testów - refaktoryzuj (unikanie duplikacji, własne asercje, klasy pomocnicze ...)
- ▶ Mockuj głównie typy abstrakcyjne, ale nie twórz abstrakcji tylko pod testy (abstrakcja powinna mieć sens)
- ▶ Nie mockuj struktur danych / value objects
- ▶ Nie nadużywaj mocków

TESTY INTEGRACYJNE

- ▶ Weryfikują współdziałanie komponentów oraz integrację z zewnętrznymi bibliotekami i/lub systemami w kontekście określonej funkcjonalności
- ▶ Stanowią kolejną linię zabezpieczenia po testach jednostkowych

STRATEGIE IMPLEMENTACJI

- ▶ **Big Bang testing** - wszystkie komponenty lub moduły są integrowane razem, a następnie testowane jako całość
- ▶ **Incremental testing** - odbywa się przez integrację i testowanie kilku, logicznie powiązanych komponentów. Następnie integrowane są przyrostowo kolejne komponenty (Bottom Up lub Top Down), aż do momentu kiedy przetestowana zostanie cała aplikacja

TESTY E2E

- ▶ Weryfikują działanie aplikacji z punktu widzenia użytkownika

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

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

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

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

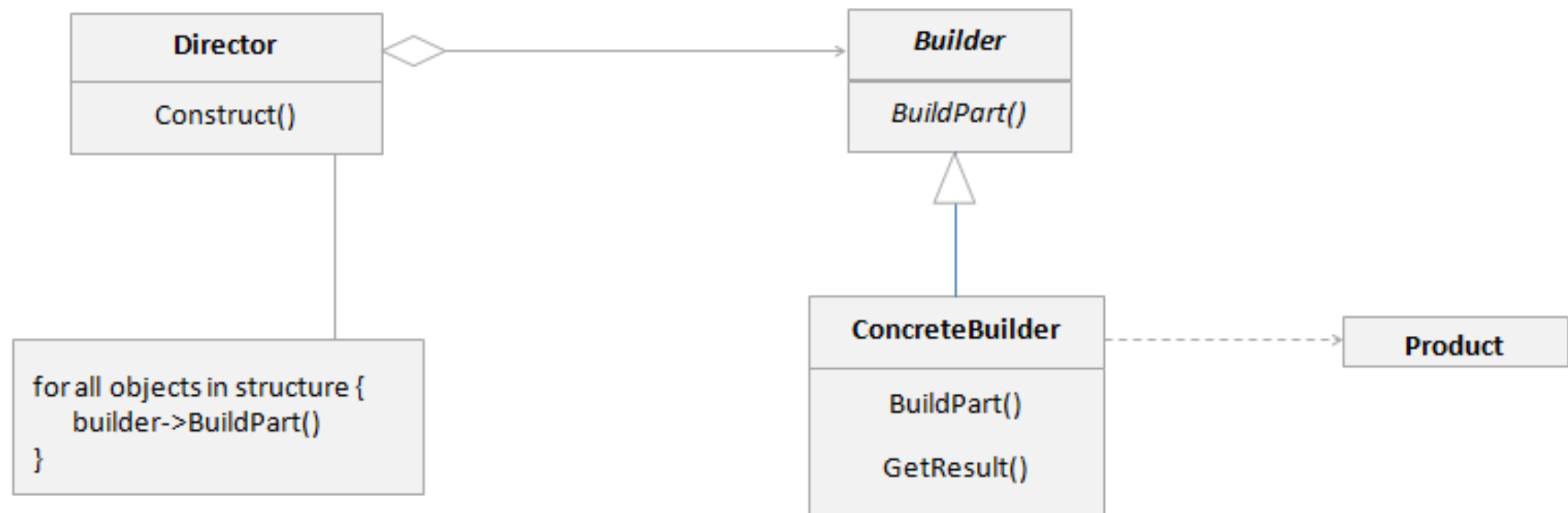
▶ Zastosowanie

- ▶ 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, OBIEKTOWY, 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

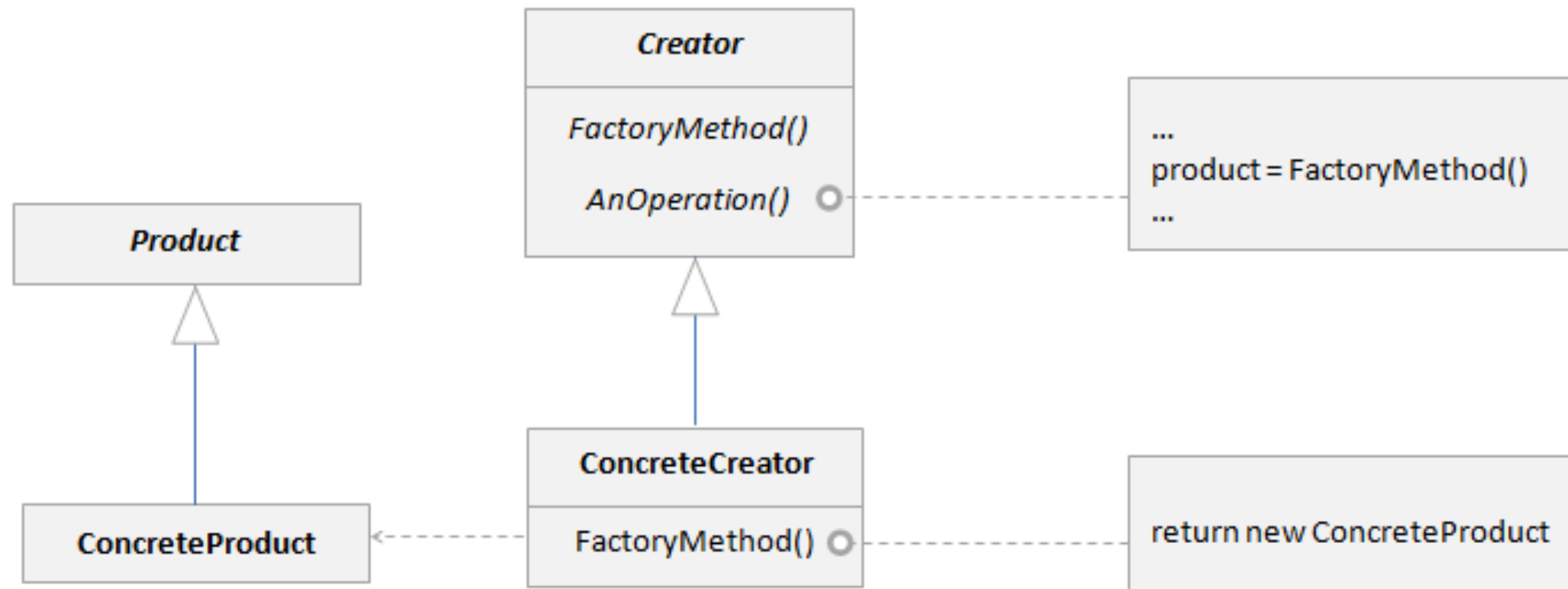
▶ Zastosowanie

- ▶ 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

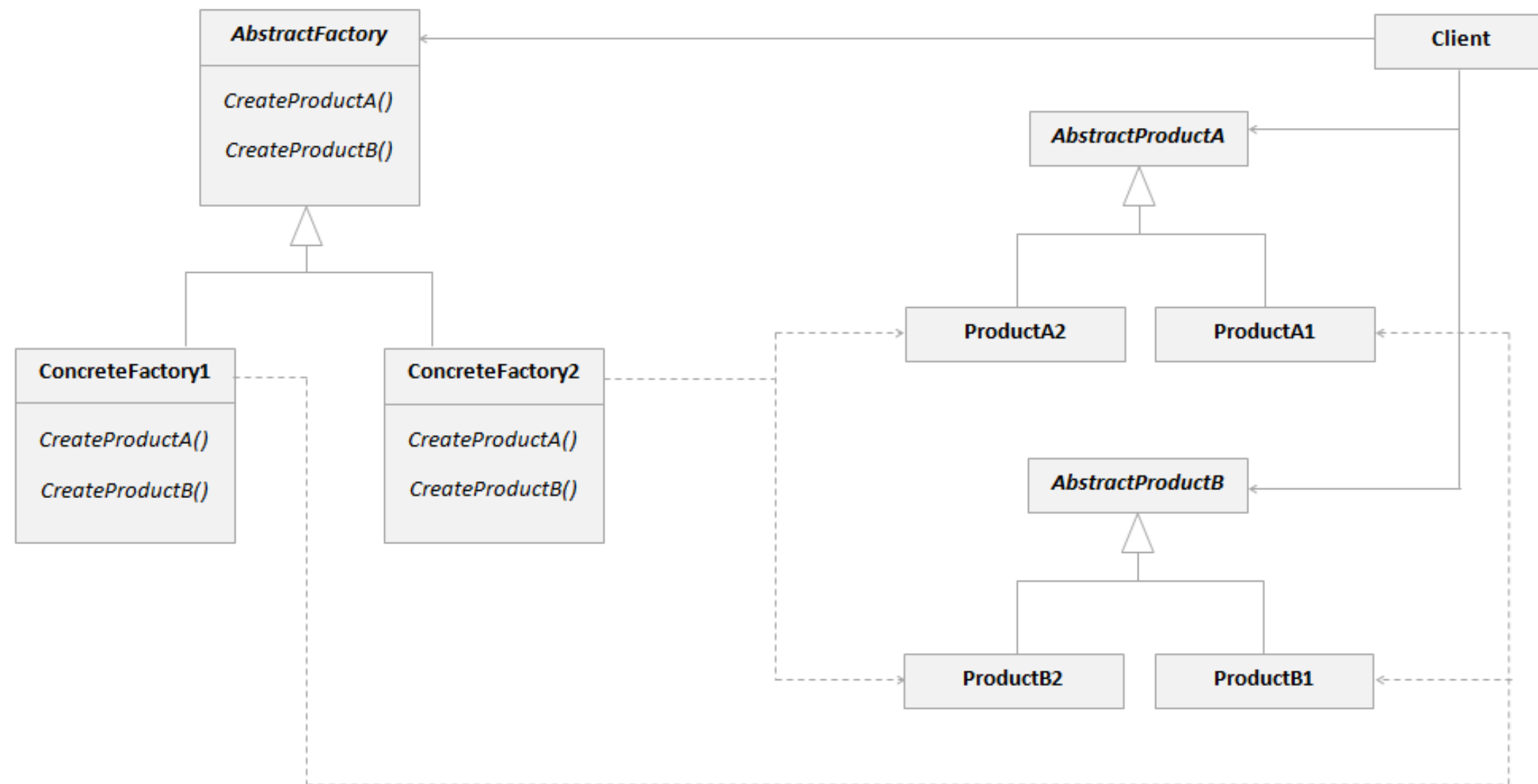
▶ Zastosowanie

- ▶ 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

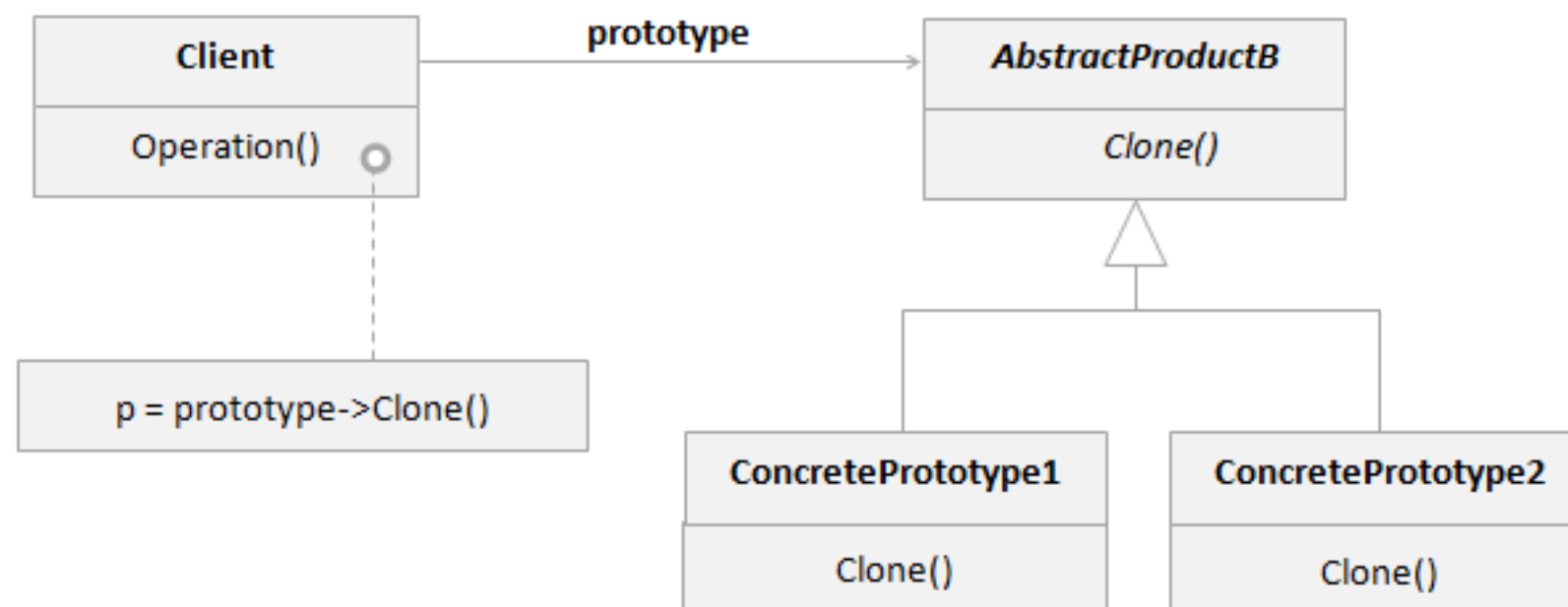
▶ Zastosowanie

- ▶ 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

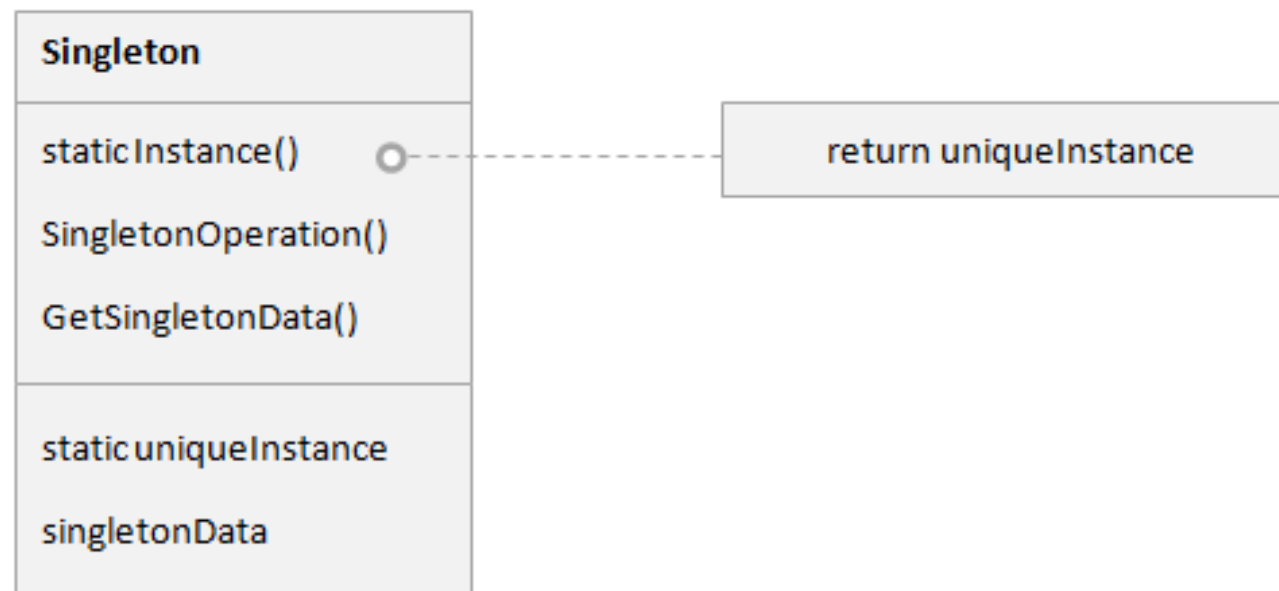
▶ Zastosowanie

- ▶ 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, OBIEKTOWY, 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

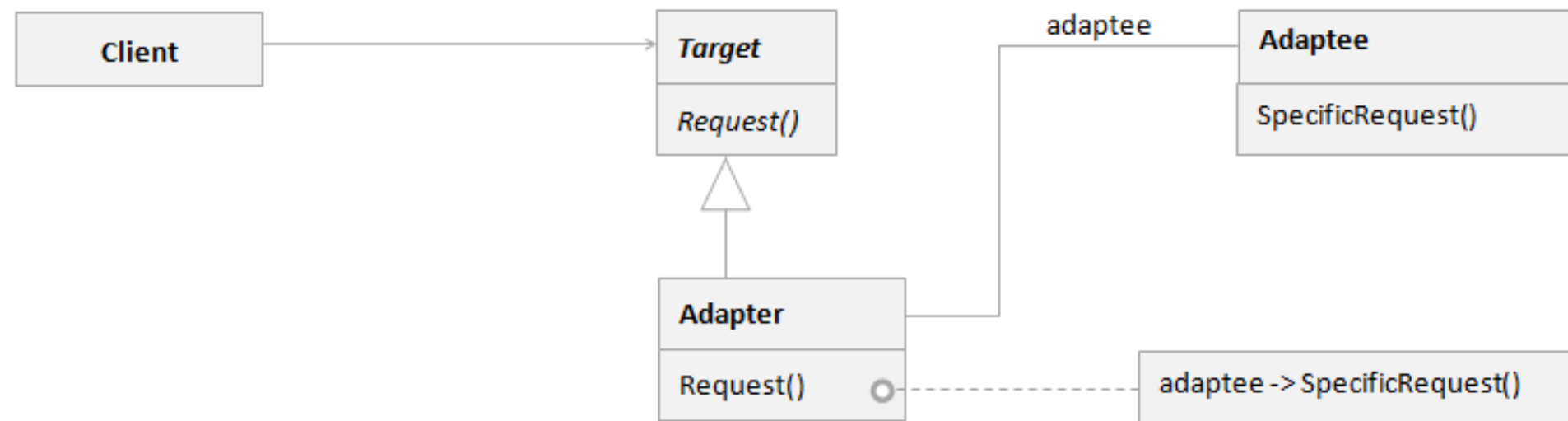
▶ Zastosowanie

- ▶ 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

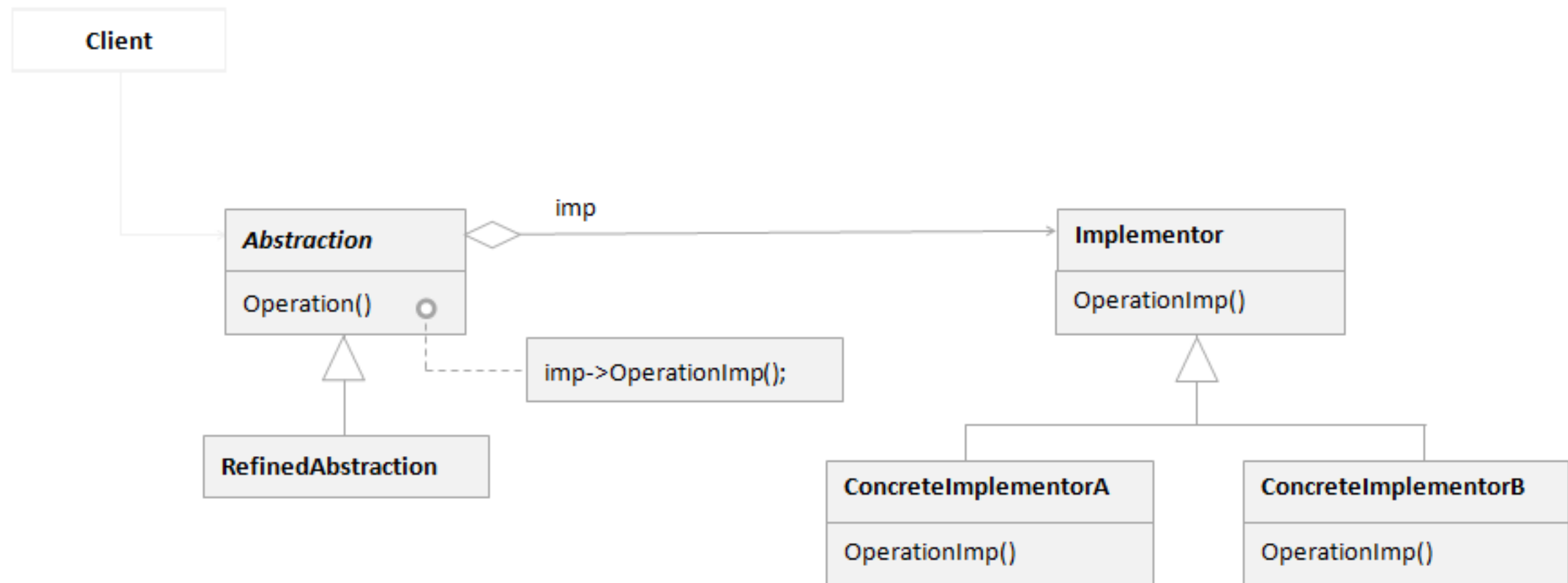
▶ Zastosowanie

- ▶ 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, OBIEKTOWY, STRUKTURALNY



COMPOSITE, OBIEKTOWY, STRUKTURALNY

▶ Przeznaczenie

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

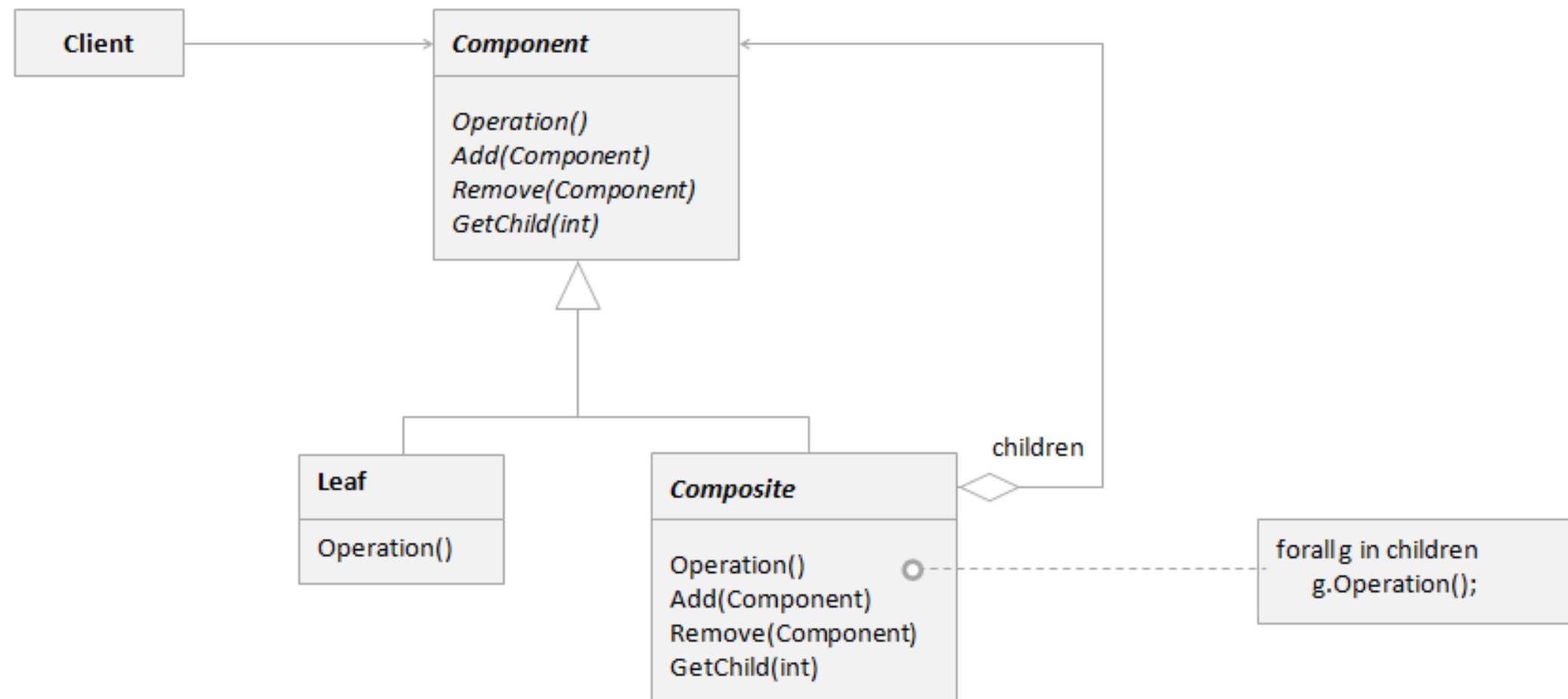
▶ Zastosowanie

- ▶ 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

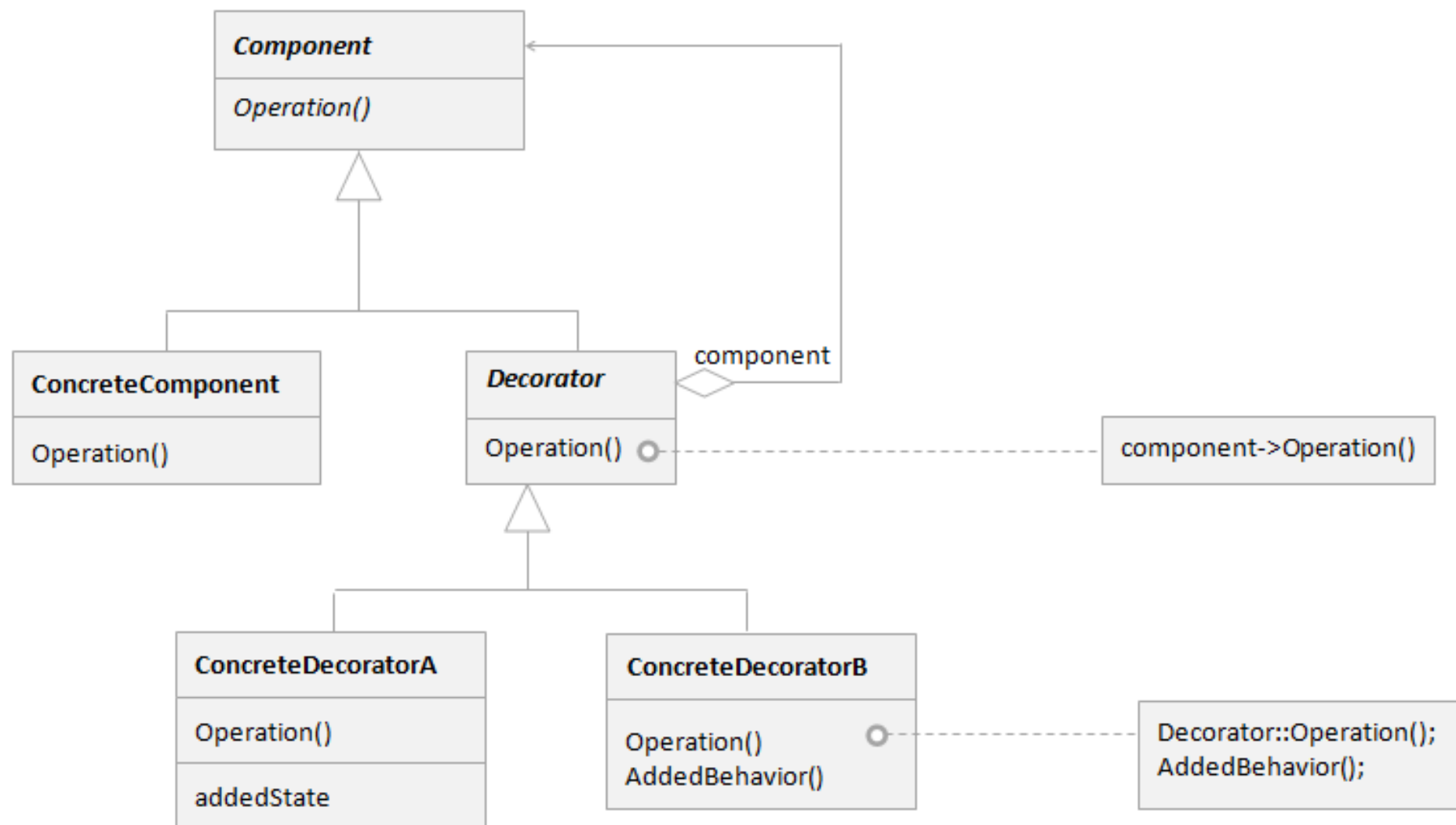
▶ Zastosowanie

- ▶ Kiedy zachodzi potrzeba dodania nowych odpowiedzialności do obiektu w dynamiczny i przejrzysty 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

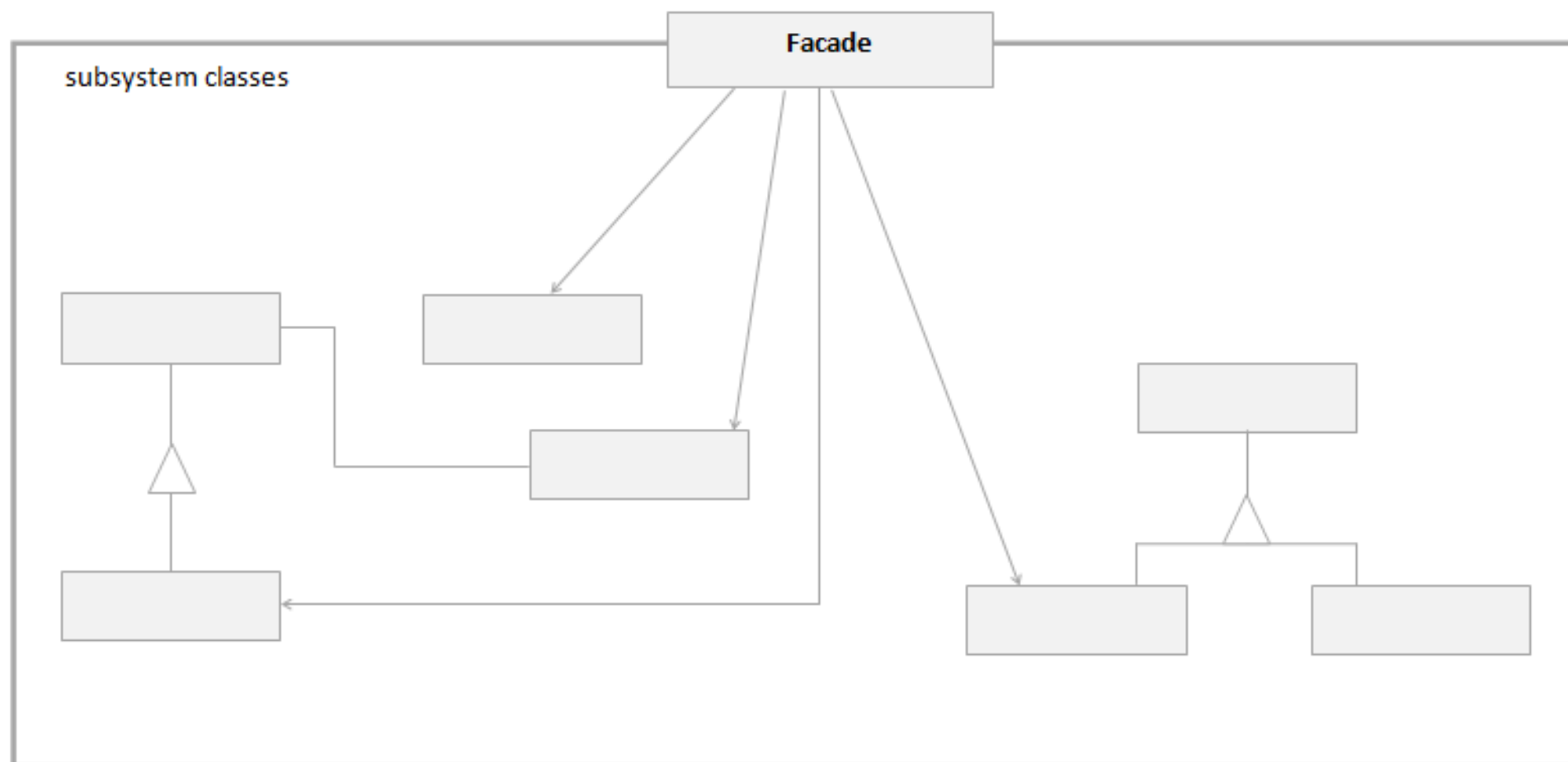
▶ Zastosowanie

- ▶ 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

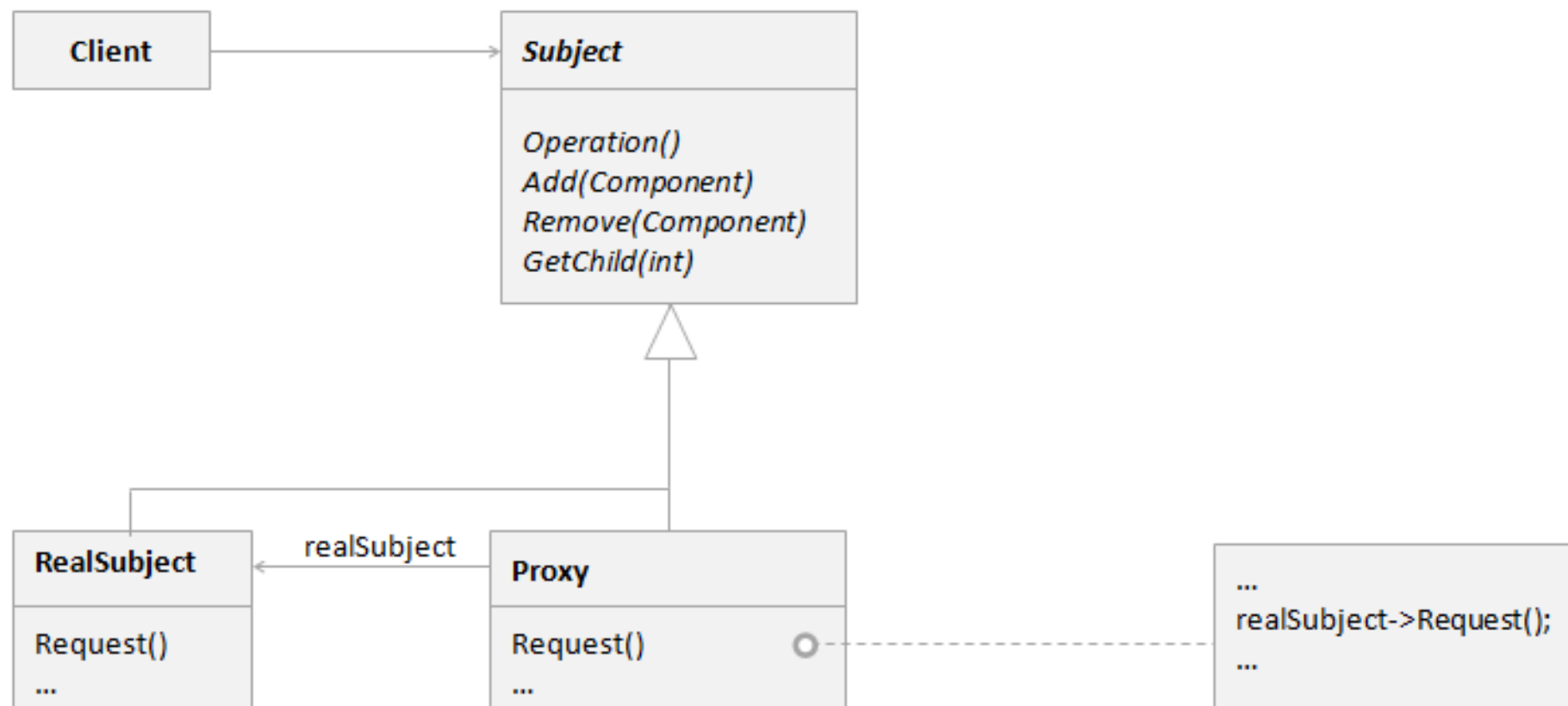
▶ Zastosowanie

- ▶ 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

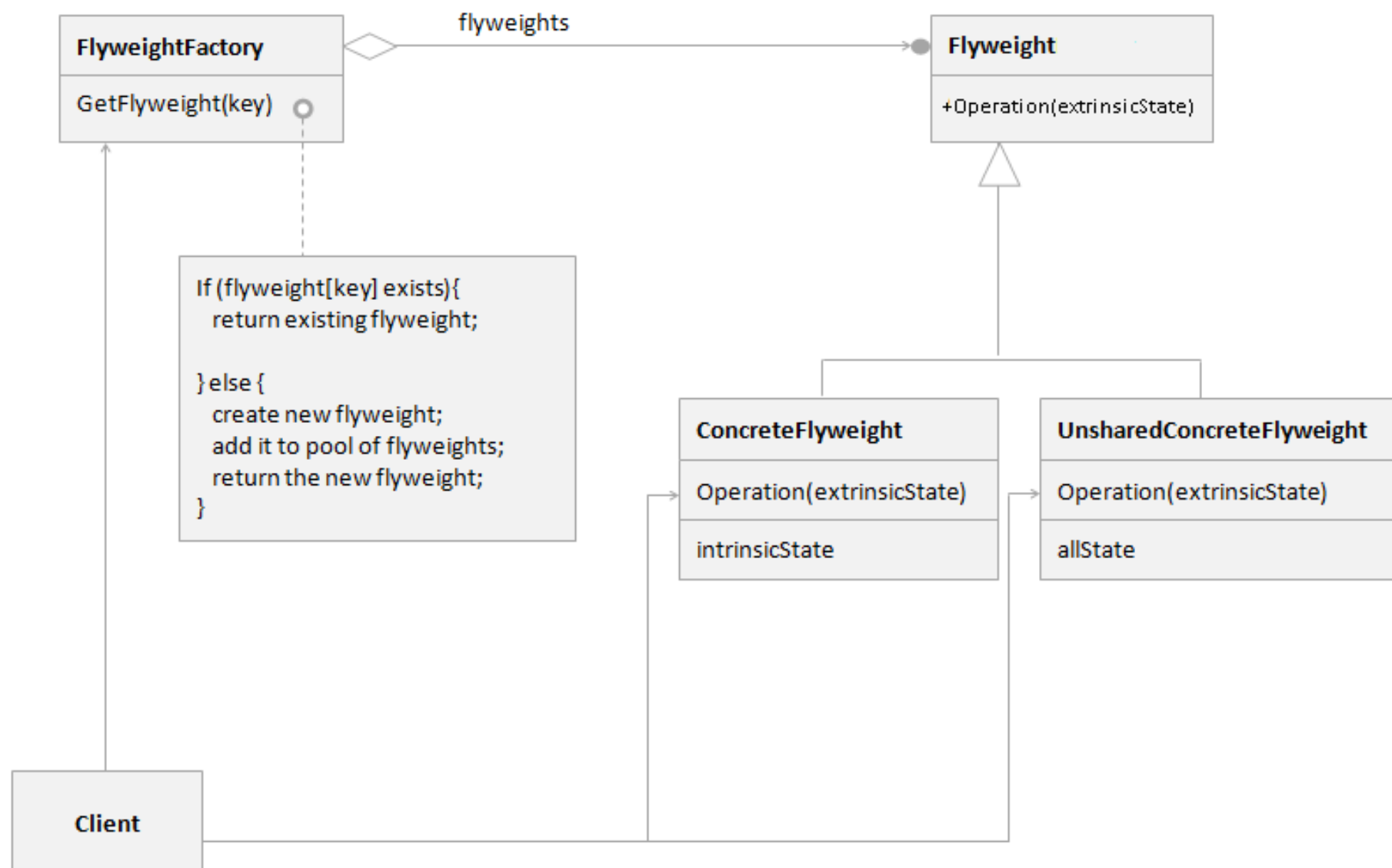
▶ Zastosowanie

- ▶ 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, OBIEKTOWY, STRUKTURALNY



CHAIN OF RESPONSIBILITY, OBIEKTOWY, BEHAWIORALNY

▶ Przeznaczenie

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

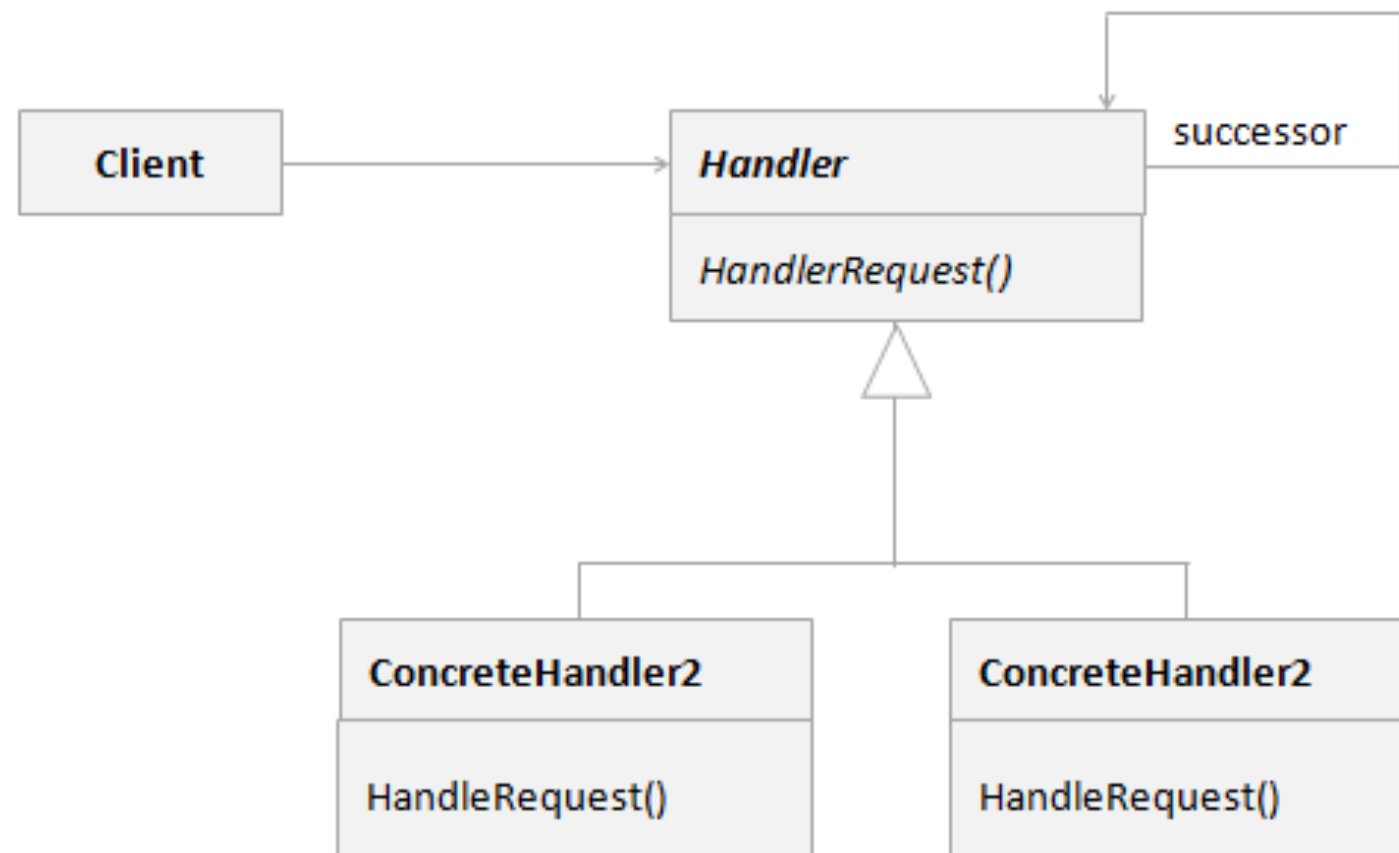
▶ Zastosowanie

- ▶ 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, OBIEKTOWY, BEHAWIORALNY



COMMAND, OBIEKTOWY, BEHAWIORALNY

▶ Przeznaczenie

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

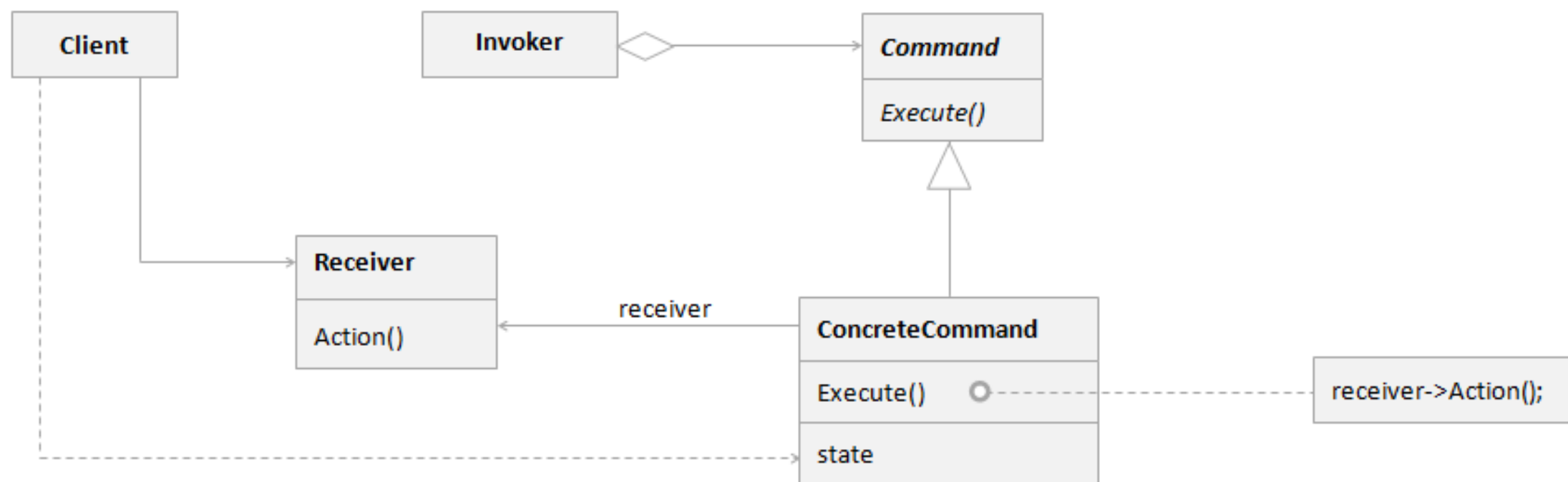
▶ Zastosowanie

- ▶ Kiedy potrzebny jest mechanizm typu callback
- ▶ Kiedy zachodzi konieczność zarządzania zadaniami - kolejkovanie 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)

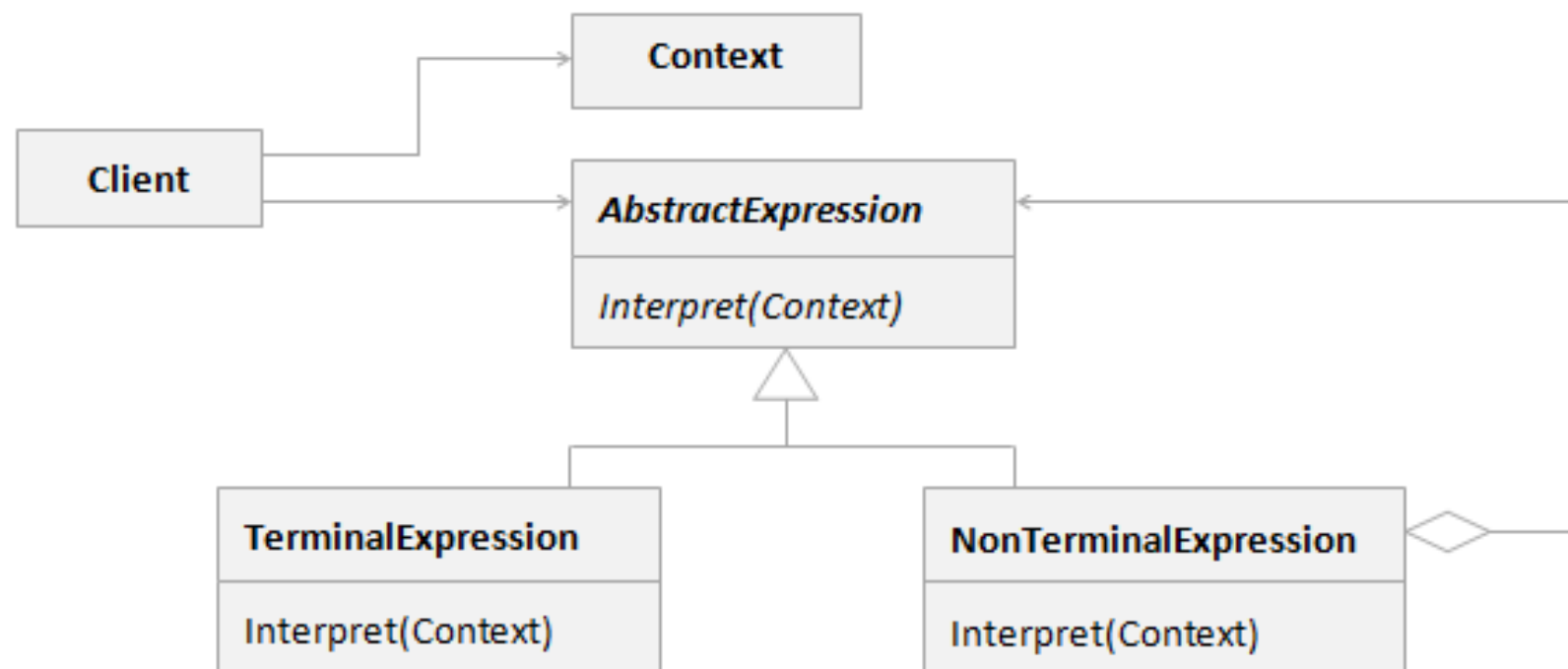
COMMAND, OBIEKTOWY, BEHAWIORALNY



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

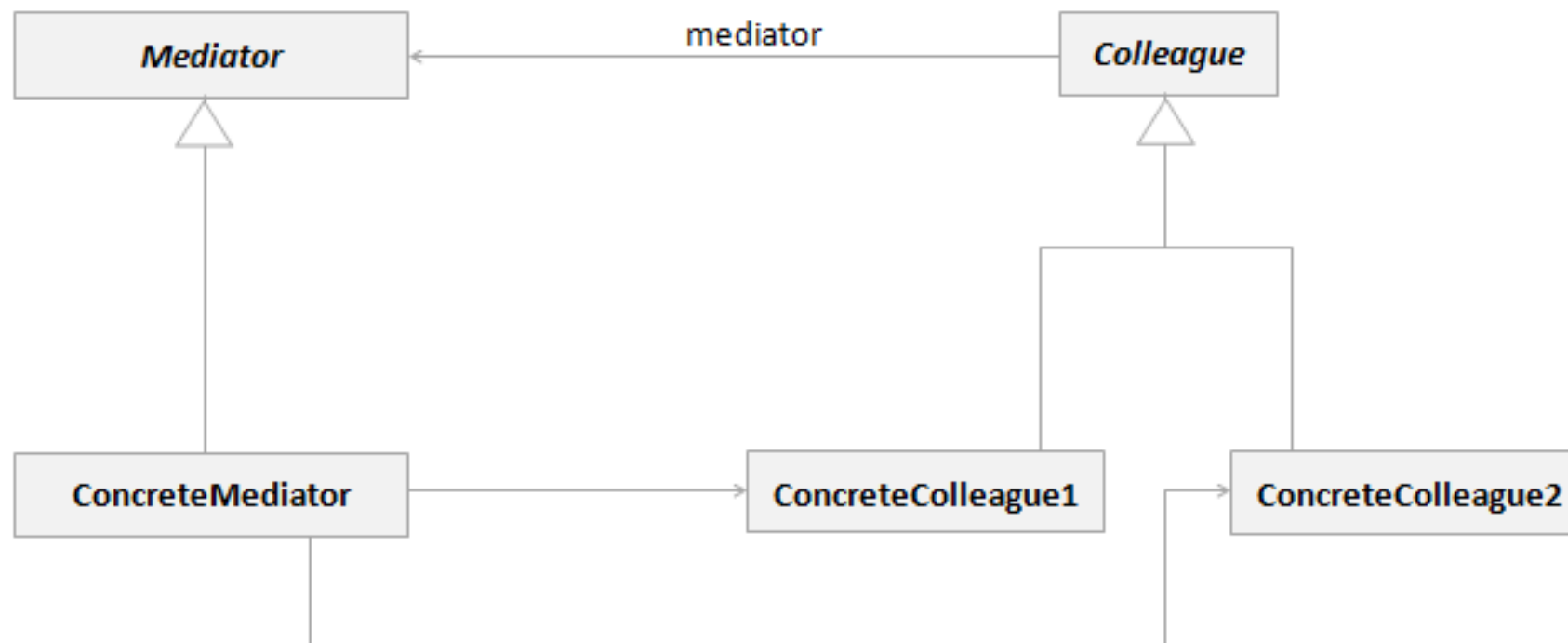
▶ Zastosowanie

- ▶ 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

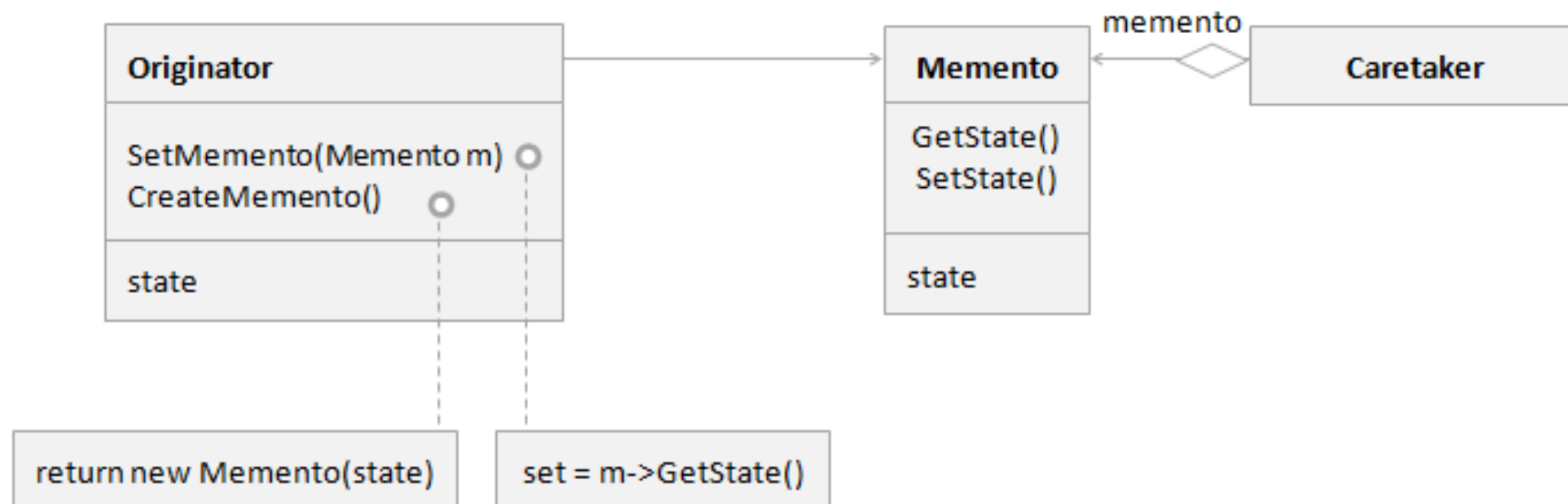
▶ Zastosowanie

- ▶ 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

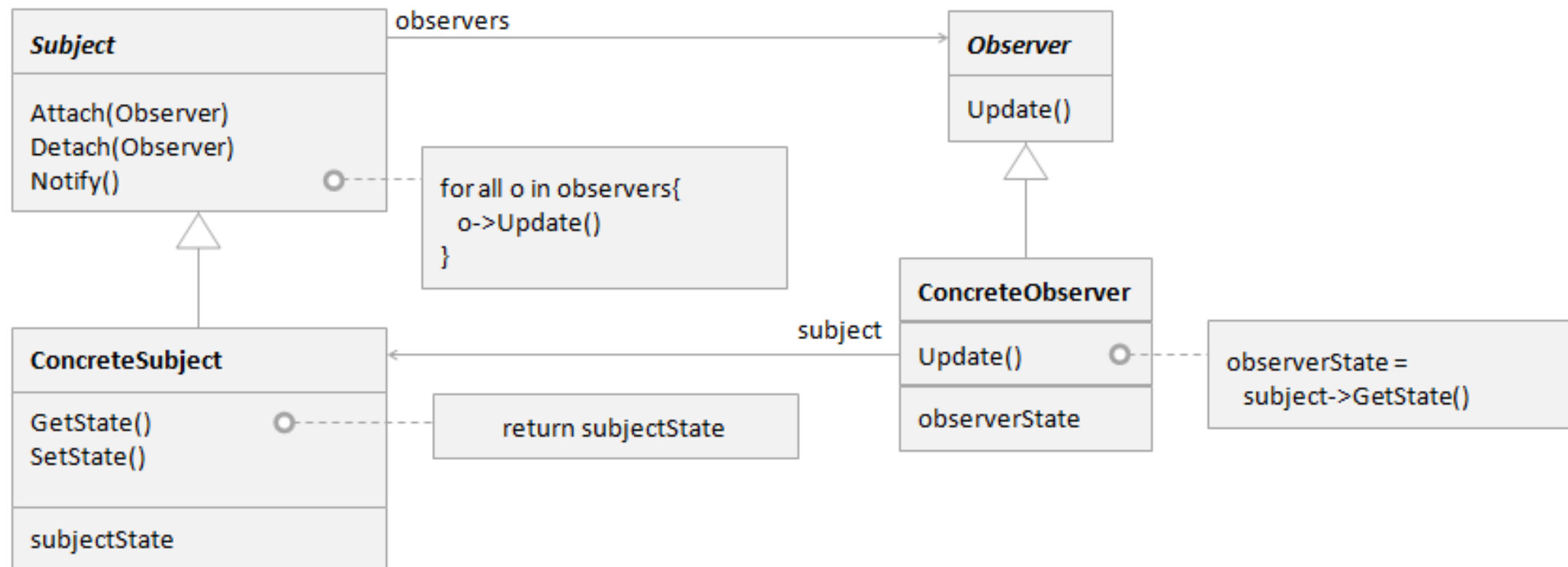
▶ Zastosowanie

- ▶ 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

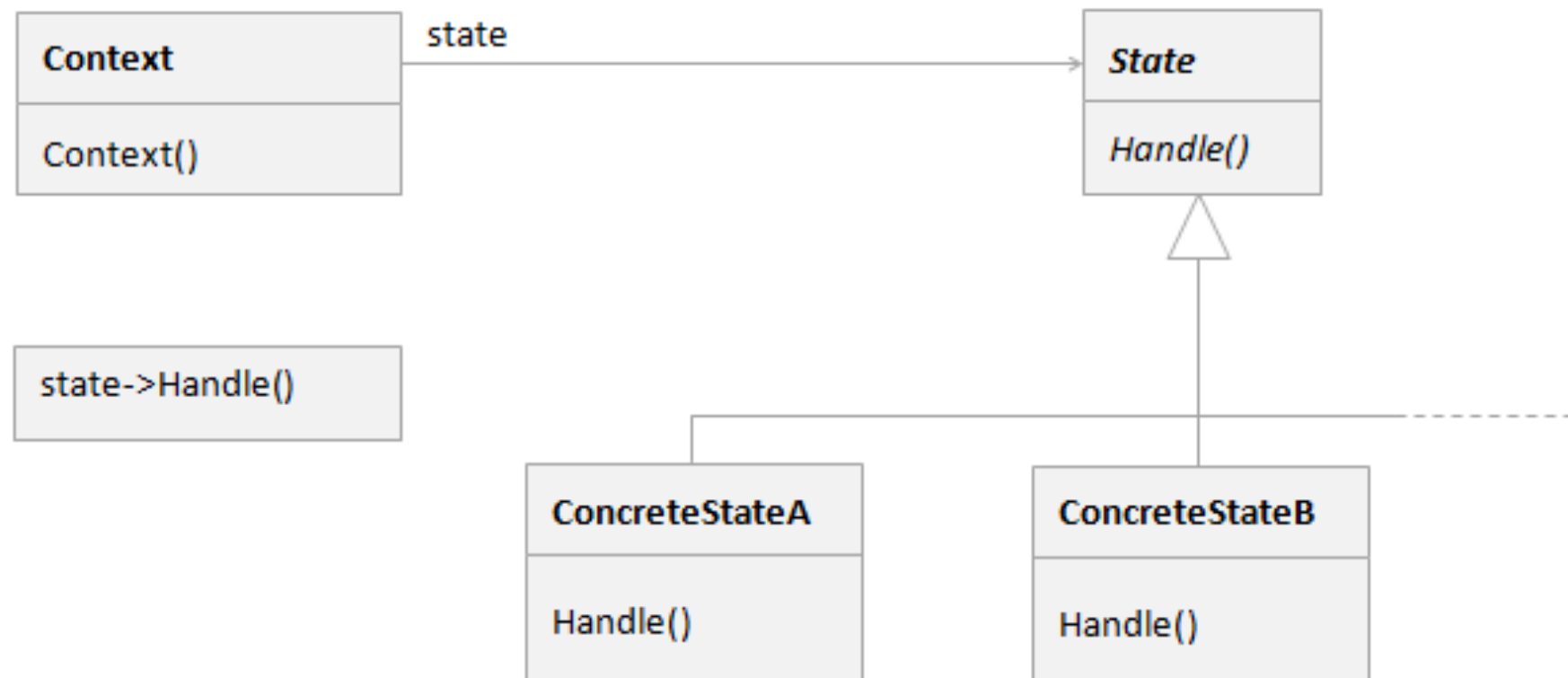
▶ Zastosowanie

- ▶ 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

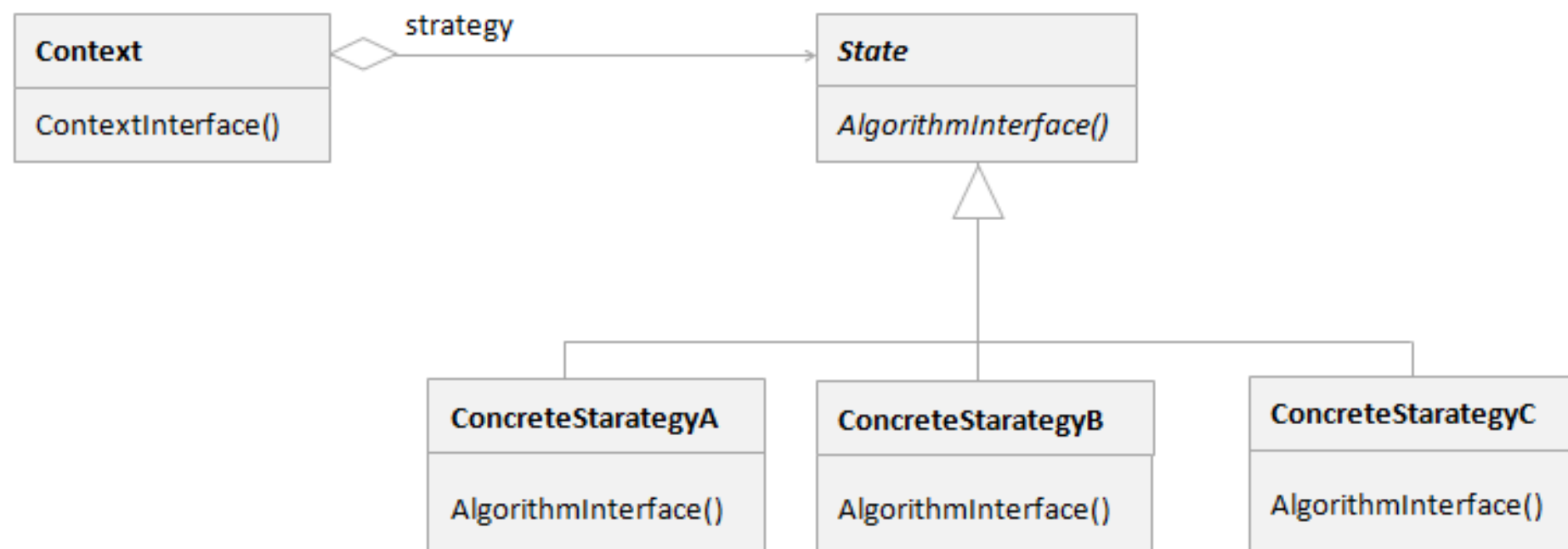
▶ Zastosowanie

- ▶ 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, OBIEKTOWY, 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)

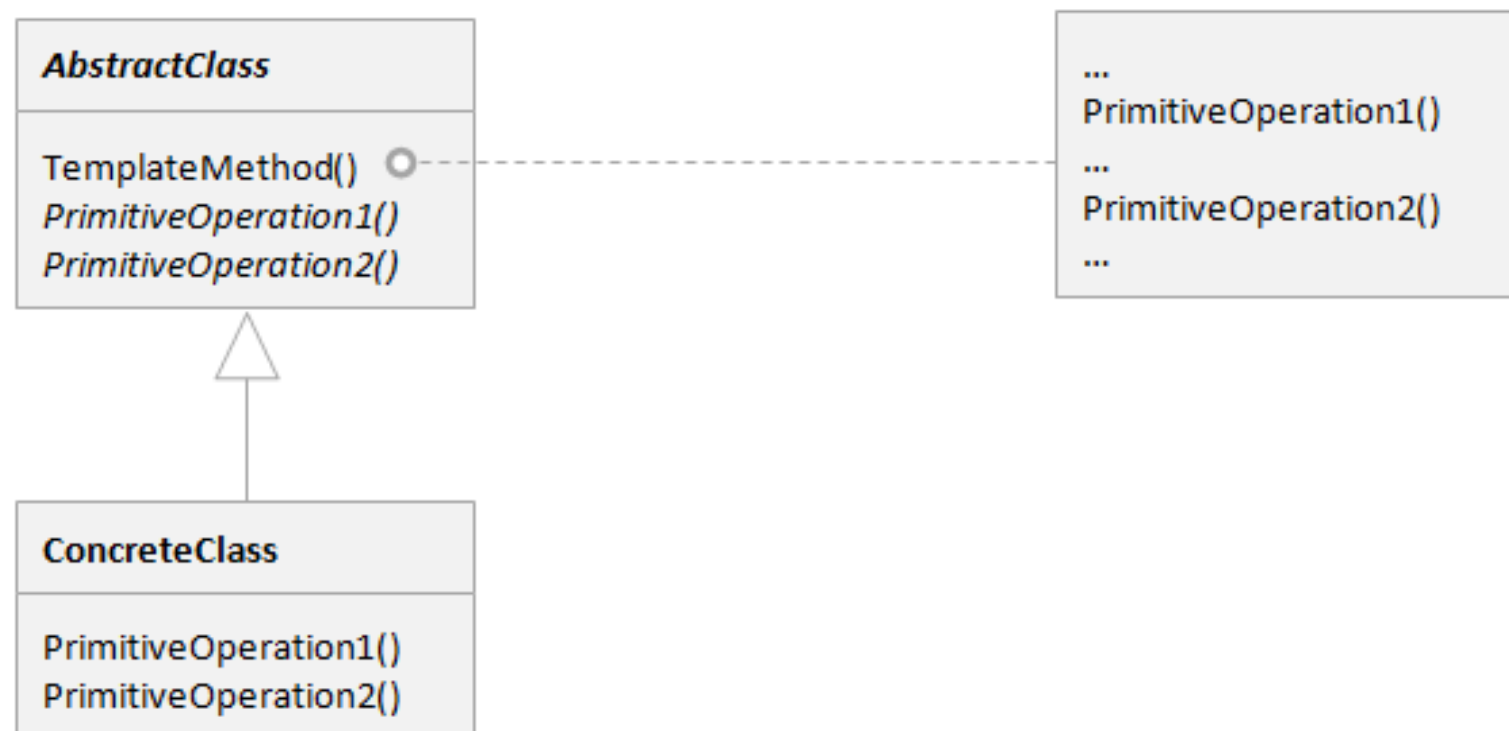
▶ Zastosowanie

- ▶ 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, OBIEKTOWY, BEHAWIORALNY

