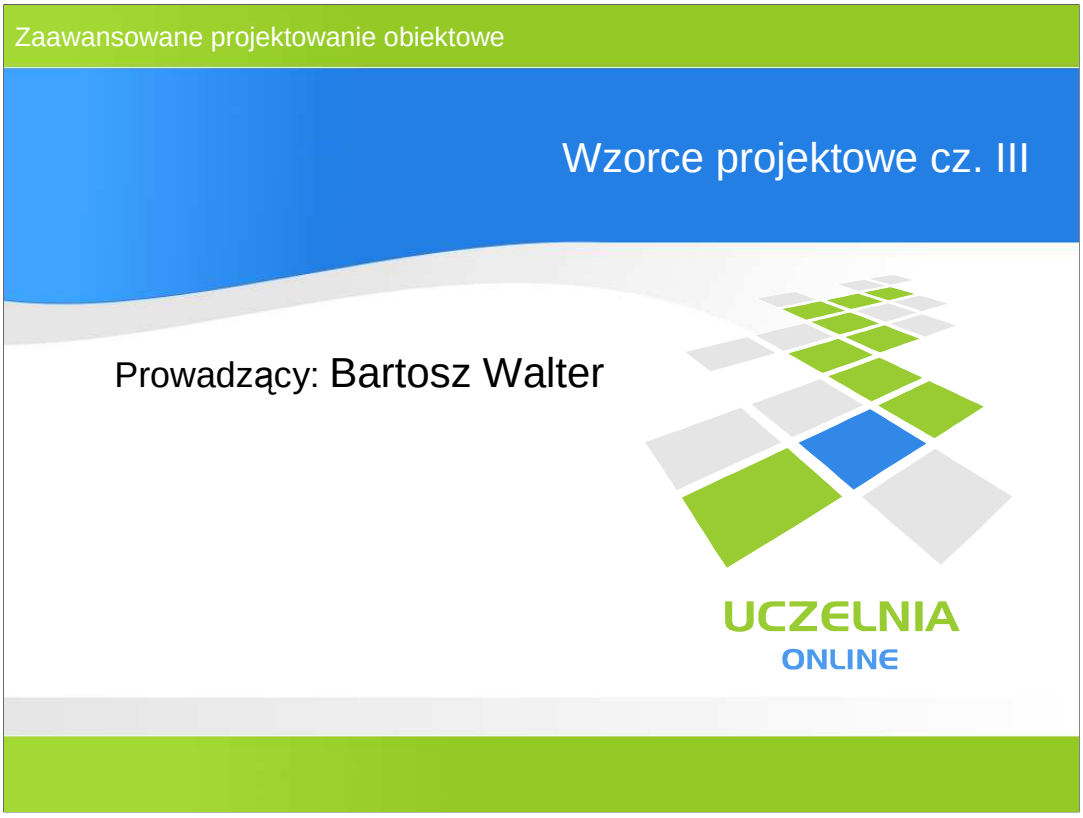

Zaawansowane projektowanie obiektowe
Wzorce projektowe cz. III
Prowadzący: Bartosz Walter
UCZELNIA
ONLINE

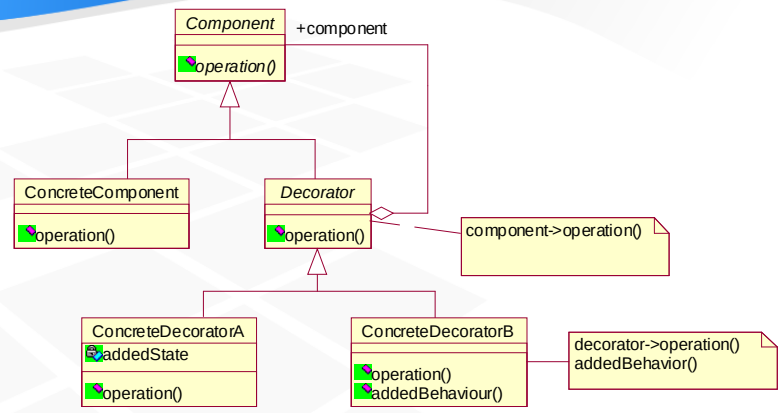
Decorator: cel

- Umożliwienie dynamicznego dodawania funkcjonalności do obiektu
- Stworzenie elastycznej alternatywy dla tworzenia podklas

E. Gamma et al. (1995)

Dekorator jest wzorcem zbliżonym pod względem struktury do wzorców Proxy i Adapter. Celem jego stosowania jest stworzenie możliwości dodawania funkcjonalności do klasy w czasie wykonywania programu. Alternatywnym sposobem realizacji podobnego celu (modyfikacji zachowania wewnątrz grupy klas) jest dziedziczenie, jednak ma ono poważne wady. Jeżeli klasa ma trzy różne właściwości, które mogą wpływać na jej zachowanie i mogą przyjmować wartości binarne (np. klasa Pracownik: wiek – pełnoletni/dziecko, zatrudnienie – pracujący/bezrobotny, stan cywilny – wolny/żonaty), wówczas do reprezentacji wszystkich możliwych przypadków należałoby utworzyć 2^3 = 8 podklas. Liczba ta rośnie wykładniczo wraz ze wzrostem liczby właściwości. Takie rozwiązanie na dłuższą metę jest nieakceptowalne, i dlatego konieczne jest wykorzystanie innego mechanizmu, np. wzorca Decoratora.

Decorator: struktura



Klient wysyła komunikat do obiektu *Decorator*, który przekazuje go obiektowi *ConcreteComponent* oraz wykonuje dodatkowe operacje („dekoracje”)

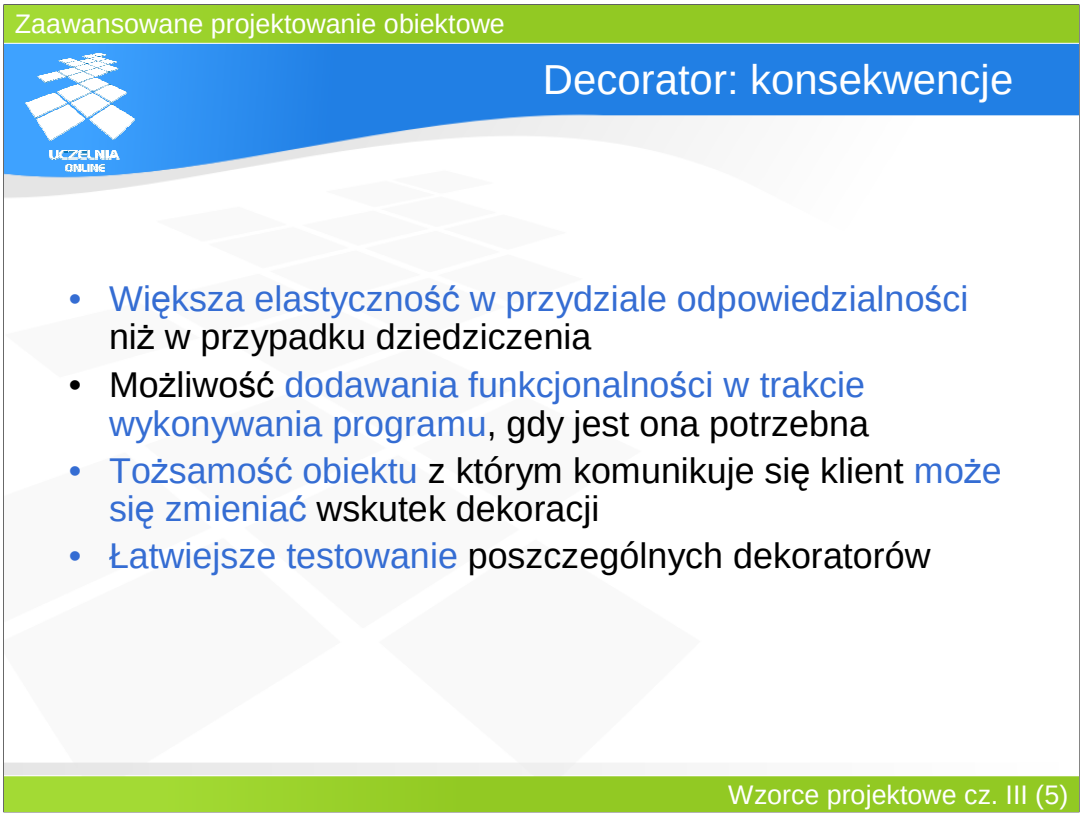
Component jest wspólnym interfejsem dla wszystkich klas, które można dekorować. Implementują go zarówno klasa ConcreteComponent, która jest odpowiedzialna za podstawową funkcjonalność oferowaną klientowi, jak i dekoratory. Każdy dekorator posiada referencję (oznaczoną jako kompozycję, aby zaznaczyć obowiązkowość i siłę tej relacji) do innego obiektu Component, którym może być ponownie dekorator lub ConcreteComponent. Otrzymując żądanie wykonania określonej operacji, dekorator deleguje je do swojego „wewnętrznego” obiektu Component, a następnie wykonuje specyficzną dla siebie dodatkową funkcjonalność. Dzięki temu dodanie do obiektu nowej funkcjonalności polega na utworzeniu dekoratora i przekazaniu mu owego obiektu. W ten sposób dekorator staje się rzeczywistym odbiorcą komunikatów od klienta, a ConcreteComponent – jego podwykonawcą.

Kiedy każdy dekorator (klasy ConcreteDecoratorA i ConcreteDecorator B) dodaje do dekorowanego obiektu tylko jedną funkcję, wówczas dekorując obiekt wielokrotnie uzyskujemy efekt osiągnięcia żądanej sumarycznej funkcjonalności. Pod względem typu udekorowany obiekt nie różni się od obiektu nieudekorowanego (klient widzi go przez interfejs Component), dlatego zastosowanie tego wzorca nie wymaga istotnych zmian w kodzie klienta.

Decorator: uczestnicy

- **Component**
  - definiuje wspólny interfejs obiektów, które można dekorować
- **Concrete Component**
  - realizuje podstawową funkcjonalność obiektu
- **Decorator**
  - posiada referencję typu *Component* i do tego obiektu deleguje komunikaty
  - rozszerza funkcjonalność obiektu *ConcreteComponent*

Warto zwrócić uwagę, że obiekt ConcreteComponent, aby mógł uczestniczyć w tym wzorcu, musi definiować interfejs Component, którego alternatywną implementacją są dekoratory. Ważne jest też, aby dekoratory odpowiednio delegowały swoje metody do wewnętrznych obiektów typu Component.

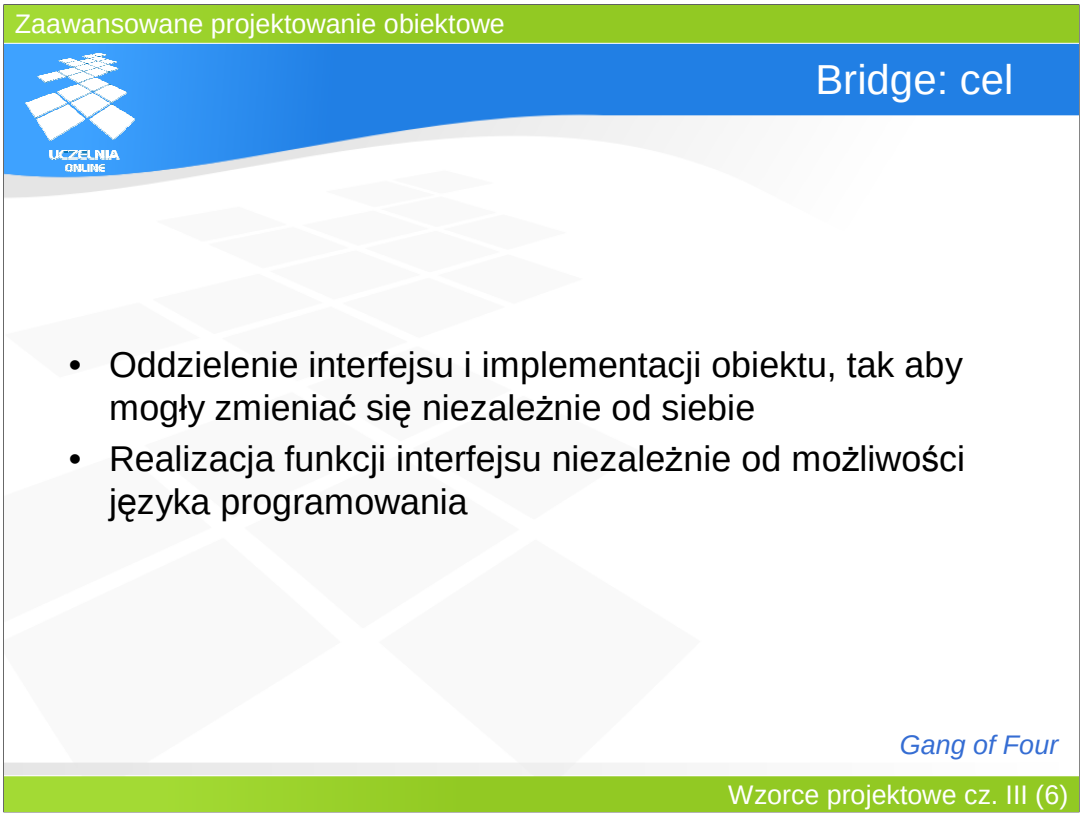


Wykorzystanie dekoratorów w celu rozszerzenia funkcjonalności oferowanej przez klasę ma wiele zalet nad stosowaniem dziedziczenia. Przydział odpowiedzialności do obiektu jest dynamiczny i na dowolnym poziomie ziarnistości, zależnym od implementacji dekoratorów.

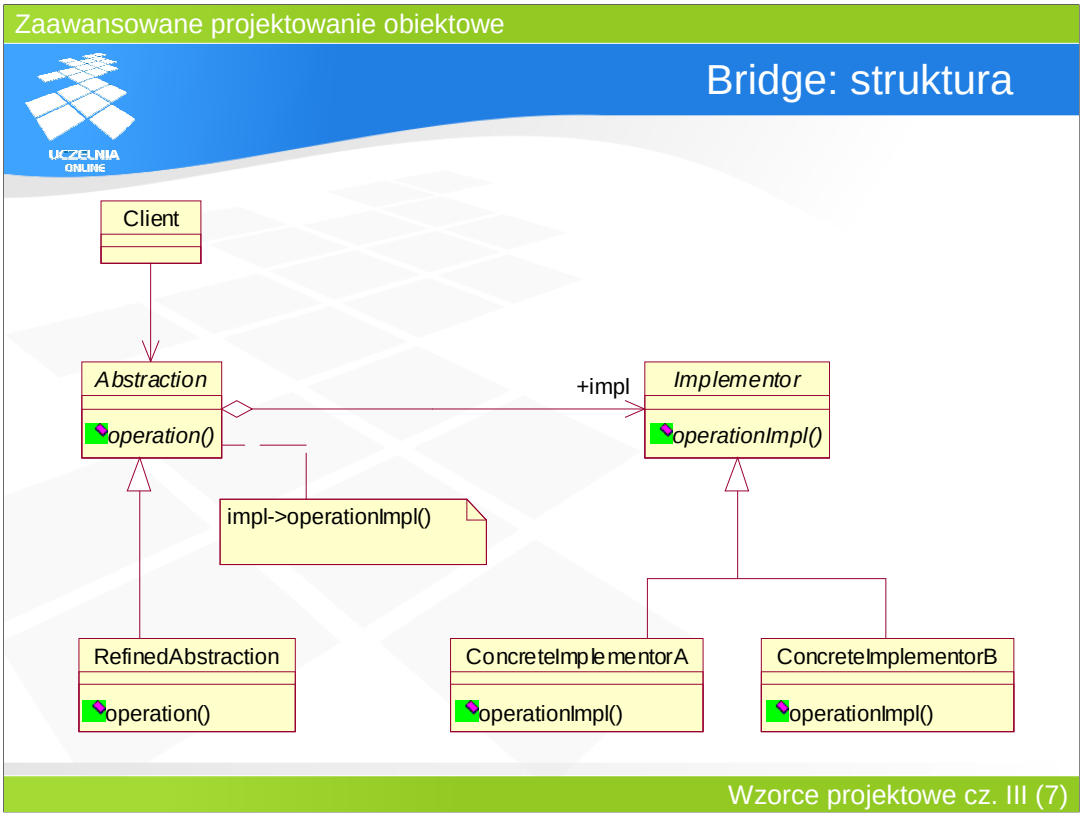
Należy zwrócić uwagę, że zastosowanie dekoratora zmienia referencję do obiektu, do którego odwołuje się klient. Aby uniknąć błędów, warto tworzenie i stosowanie dekoratorów powierzyć specjalizowanej metodzie (typu Factory Method).

Ponieważ dekoratory służą do modyfikacji zachowania, a nie przechowywania danych (w szczególności dekoratory mogą być obiektami bezstanowymi), nie należy przechowywać w nich informacji. Pozwala to utrzymać ich relatywnie niewielki rozmiar.

Stosowanie dekoratorów przyczynia się do łatwiejszego testowania jednostkowego systemu, ponieważ każdy dekorator wymaga jedynie testów specyficznych dla siebie, a nie dla kompletnie udekorowanego obiektu.



Wzorzec Bridge jest niezależnym od języka programowania i oferowanych przez niego możliwości sposobem na rozdzielenie interfejsu i implementacji. W ten sposób oba elementy mogą zmieniać się niezależnie od siebie, tworzyć swoje podklasy etc.



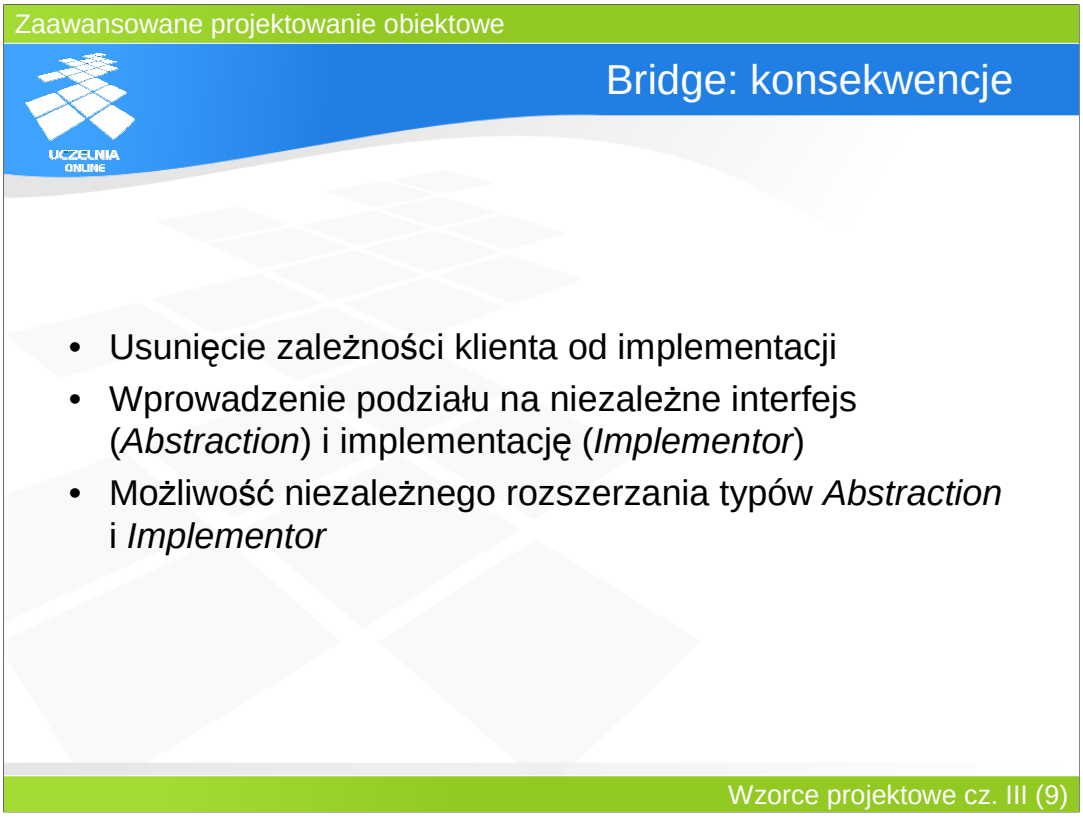
Wzorzec składa się z dwóch interfejsów: Abstraction i Implementor, oraz ich implementacji. Oba interfejsy mogą w rzeczywistości być zwykłymi klasami, jeżeli użyty język programowania nie posiada interfejsów jako swoich elementów. Klient kontaktuje się z obiektem Abstraction i nie jest w żaden sposób zależny od obiektu Implementor. Abstraction jest związany relacją kompozycji z wybranym obiektem Implementor, i do niego deleguje wszystkie żądania przesłane przez klienta.

Struktura wzorca bardzo przypomina wzorzec Adapter, jednak cel jest zupełnie inny: intencją jest rozdzielenie abstrakcji od implementacji, tak aby implementacja nie była dostępna dla klienta. Taka struktura pozwala m.in na zmianę obiektu Implementor w trakcie działania programu.

Bridge: uczestnicy

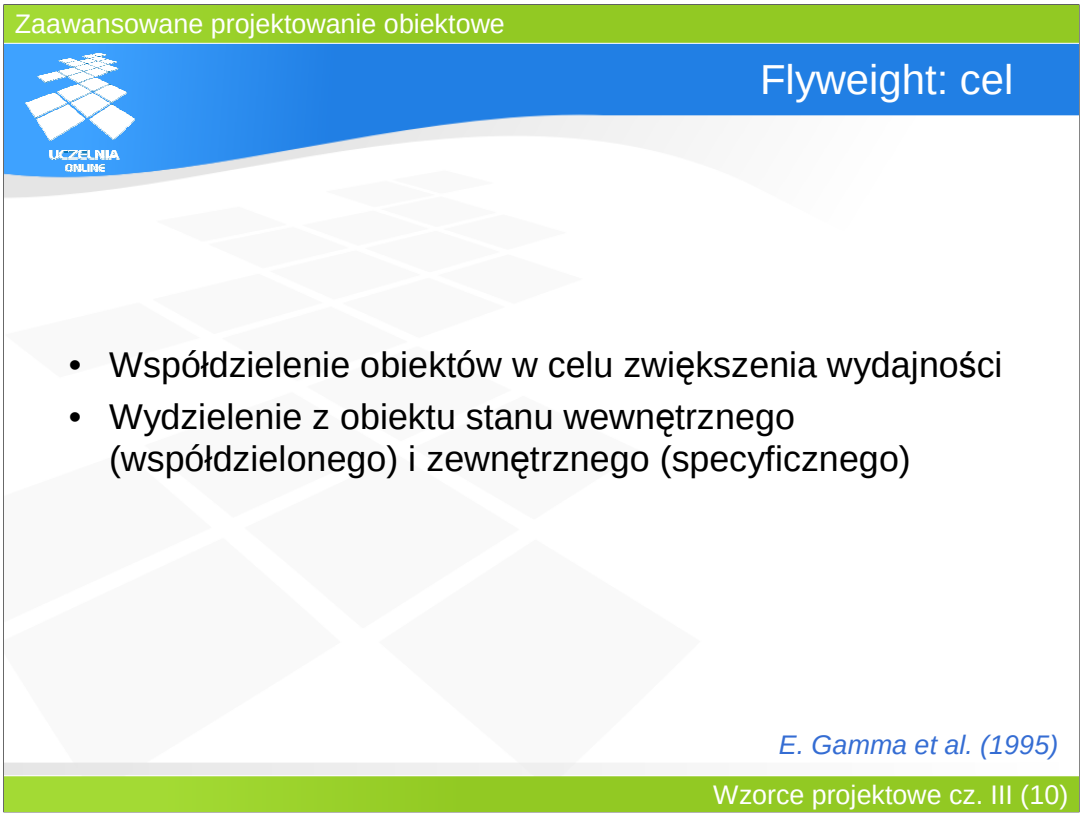
- **Abstraction**
  - definiuje interfejs zewnętrzny (abstrakcję)
  - posiada referencję do obiektu typu *Implementor*
  - deleguje żądania do obiektu typu *Implementor*
- **Refined Abstraction**
  - rozszerza interfejs *Abstraction*
- **Implementor**
  - definiuje interfejs wewnętrzny (implementację)
  - niespokrewniony z typem *Abstraction*
- **Concrete Implementor**
  - implementuje typ *Implementor*

Z punktu widzenia klienta obiekt Abstraction jest wykonawcą jego poleceń. Aby pełnić swoją rolę, obiekt ten musi posiadać referencję do obiektu Implementor, definiującego interfejs wewnętrzny i faktycznie realizującego wymaganą funkcjonalność. Co ważne, obiekty Abstraction i Implementor nie muszą w żaden sposób (przez dziedziczenie, implementację interfejsu etc.) być ze sobą spokrewnione.



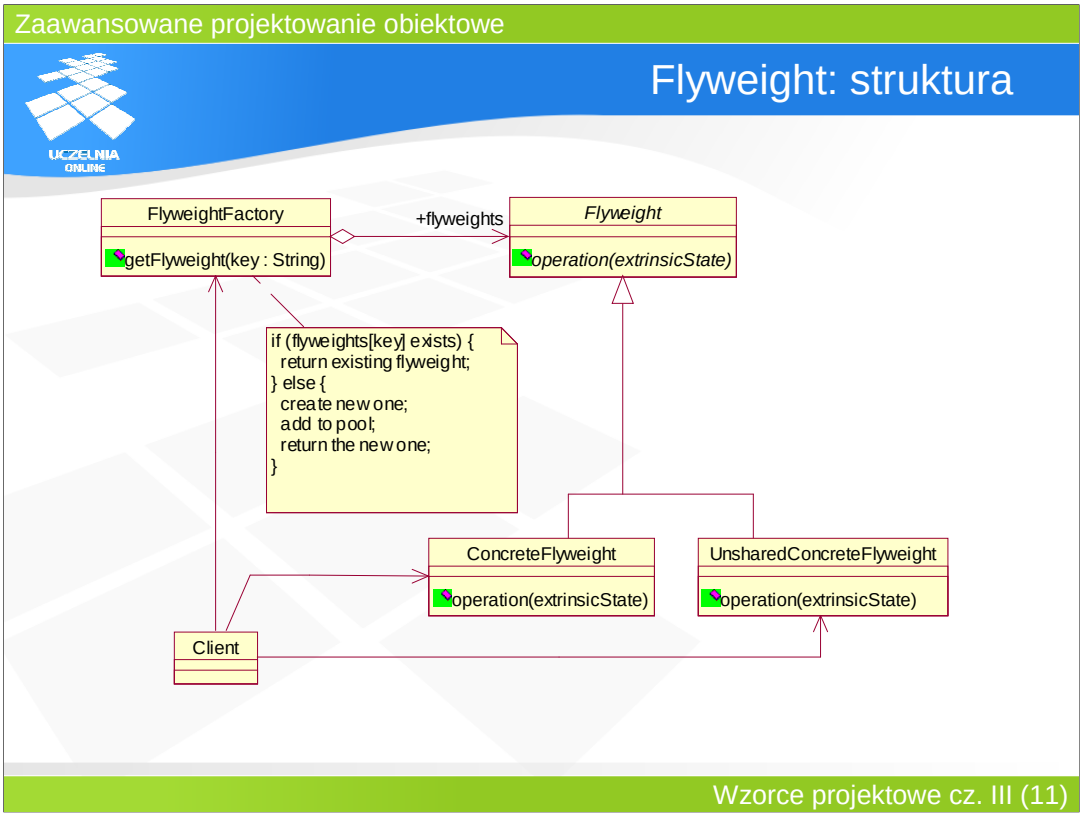
Wzorzec Bridge, jak nazwa wskazuje (z ang. most), łączy abstrakcję i implementację. Dzięki temu klienci zależą jedynie od abstrakcji i pozostają niezależni od implementacji, która może się zmieniać. Oba „przyczółki mostu” – abstrakcja i implementacja – mogą być rozszerzane przez dziedziczenie niezależnie od siebie.

Wzorzec Bridge jest ciekawym sposobem uzupełnienia możliwości oferowanych przez niektóre języki programowania o możliwość tworzenia obiektów o funkcjonalności interfejsu.



Flyweight jest wzorcem opisującym zasadę współdzielenia obiektów w sytuacjach, gdy są one potrzebne niejednocześnie i tylko przez pewien okres czasu. Flyweight różni się od wzorca Pool of Objects, ponieważ pozwala współdzielić obiekty stanowe, których dane zależą od kontekstu. Wykorzystanie wzorca ma na celu przede wszystkim podniesienie wydajności aplikacji przez ograniczenie liczby obiektów oraz wydzielenie z nich stanu zewnętrznego (specyficznego dla każdej instancji i zależnego od kontekstu) oraz zawarteg w nich tzw. stanu wewnętrznego (współdzielonego przez wszystkie instancje).

Na przykład litery w procesorze tekstu są reprezentowane przez obiekty klasy Litera, w której stanem wewnętrznym jest kod znaku, a zewnętrznym – krój litery, jej wielkość, dekoracje etc.



Obiektem współdzielonym jest Flyweight, który posiada dwa rodzaje stanu: wewnętrzny, który jest współdzielony przez wszystkie instancje tej klasy, oraz zewnętrzny, który jest specyficzny dla danej instancji. Stan wewnętrzny nie musi być modyfikowany, zatem nie ma konieczności bezpośredniego dostępu do niego. Stan zewnętrzny natomiast musi być dostarczony z zewnątrz w momencie, gdy klient zażąda użycia obiektu Flyweight w konkretnym kontekście.

Zarządzaniem obiektami Flyweight zajmuje się obiekt Flyweight Factory, która udostępnia klientowi metodę do pobierania instancji Flyweight. Na podstawie parametrów przekazanych tej metodzie fabryka może ustalić, jaki stan zewnętrzny odpowiada żądanemu obiektowi, i dostarcza go. Fabryka zarządza także pulą generycznych, pozbawionych stanu obiektów Flyweight. W momencie żądania dostarczenia obiektu fabryka pobiera obiekt z puli, konfiguruje go odpowiednim stanem zewnętrznym, i zwraca klientowi gotowy do użycia obiekt.

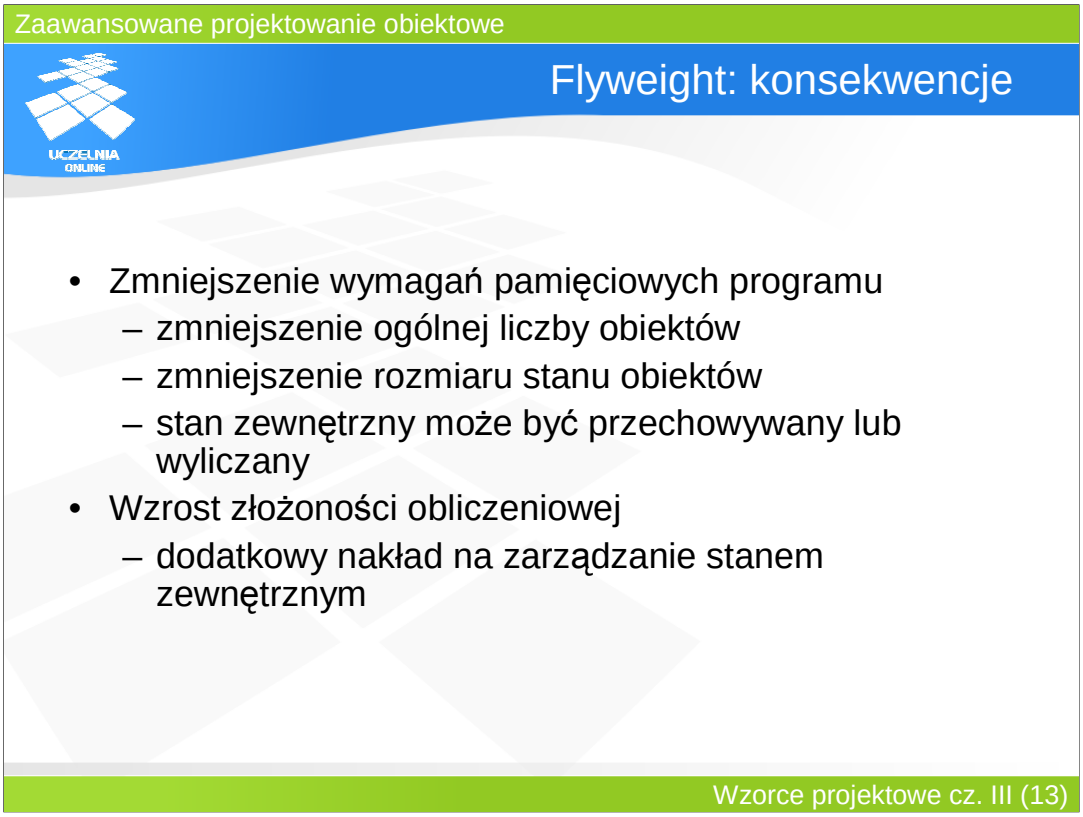
Wzorzec przewiduje też specjalną podklasę UnsharedConcreteFlyweight do reprezentowania tych obiektów, które celowo nie powinny być współdzielone. Jej użycie pozwala na zachowanie struktury wzorca i jego funkcjonalności z punktu widzenia klienta.

Flyweight: uczestnicy

- **Flyweight**
  - podlega współdzieleniu między klientów
  - definiuje interfejs do przyjmowania i odtwarzania stanu zewnętrznego obiektu
- **Concrete Flyweight**
  - przechowuje stan wewnętrzny (współdzielony)
  - jest niezależny od kontekstu (z wyjątkiem stanu zewnętrznego)
- **Flyweight Factory**
  - tworzy i przechowuje obiekty *Flyweight*
- **Client**
  - otrzymuje obiekty *Flyweight* za pośrednictwem *Flyweight Factory*

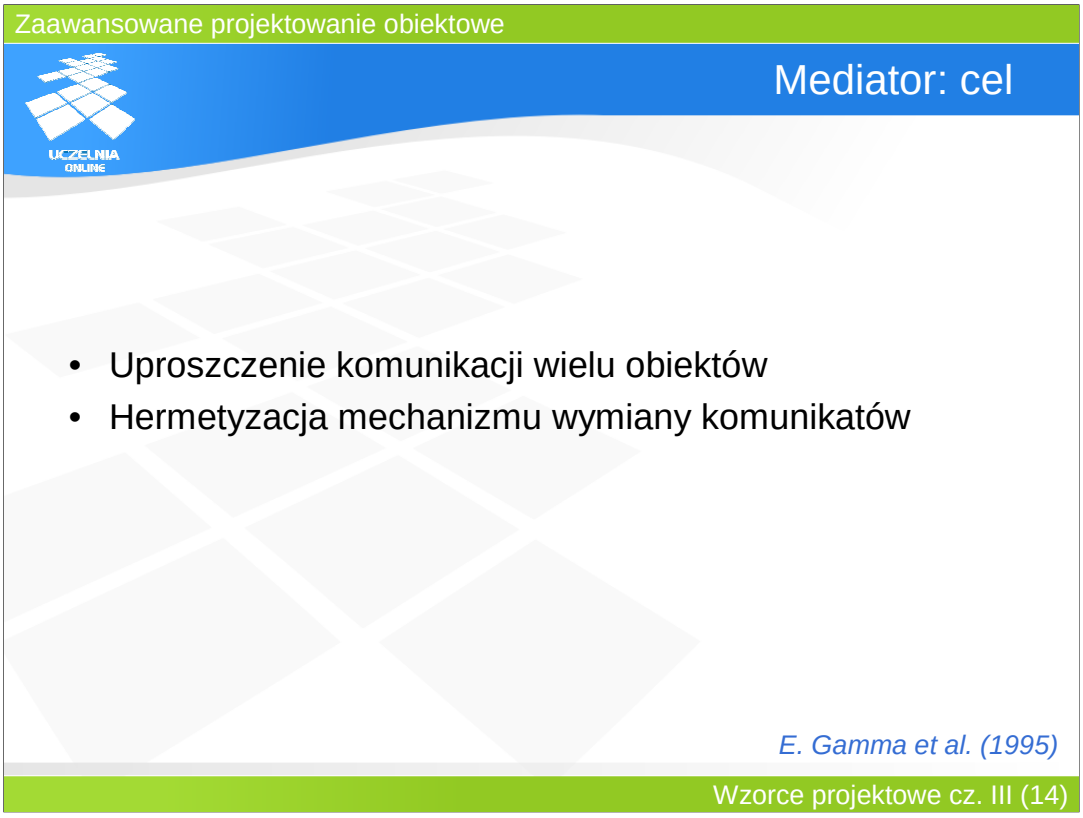
Obiekt Flyweight musi posiadać interfejs do obsługi stanu zewnętrznego. Zwykle są to metody dostępowe typu get/set, które konfigurują obiekt.

Flyweight Factory stanowi (z punktu widzenia klienta) fabrykę do tworzenia obiektów. Obiekt ten posiada pamięć (pulę obiektów), w której przechowuje wcześniej utworzone instancje. Zajmuje się także zapisem i odtwarzaniem (serializacją i deserializacją) stanu zewnętrznego obiektu.



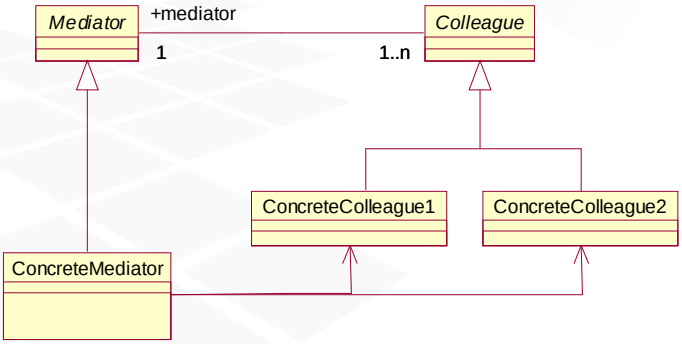
Zastosowanie tego wzorca pozwala na znaczne oszczędności pamięci, szczególnie w aplikacjach korzystających z dużej liczby instancji tego samego typu. Z jednej strony ulega zmniejszeniu ogólna liczba utworzonych obiektów, a z drugiej – rozmiar ich stanów wewnętrznych. Oczywiście, konieczne jest także przechowywanie stanu zewnętrznego, jednak w pewnych sytuacjach może on być obliczony, a nie przechowywany, a ponadto nie wymaga on tworzenia i usuwania obiektów, co jest głównym problemem w tego typu aplikacjach.

Przykładem zastosowania tego wzorca jest mechanizm zarządzania komponentami EJB w kontenerach. Gdy klient zażąda stworzenia instancji komponentu, kontener pobiera "pustą" instancję z puli i aktywuje ją poprzez wprowadzenie do niej danych, które wynikają z żądania klienta. Po zakończeniu korzystania z instancji przez klienta lub gdy jest ona przez dłuższy czas niewykorzystywana, następuje jej pasywacja, tzn. jej stan zewnętrzny jest z niej usuwany i zapisywany poza nią, a ona sama wraca do puli gotowych obiektów. Ten mechanizm pozwala na znaczną poprawę efektywności kontenera EJB.



Mediator znajduje zastosowanie w sytuacji, gdy wiele obiektów o wspólnym interfejsie musi komunikować się ze sobą w celu wykonania określonego zadania. Najprostszym, lecz trochę naiwnym rozwiązaniem jest powiązanie wszystkich obiektów ze sobą w topologii grafu pełnego. Takie rozwiązanie jest jednak słabo skalowalne: dołączenie kolejnego obiektu powoduje konieczność powiadomienia o zmianie wszystkich pozostałych, aby potrafili skomunikować się z nowym uczestnikiem interakcji. Ponadto powoduje, że mechanizm komunikacji jest rozproszony, co utrudnia jego modyfikację i dalszy rozwój.

Mediator: struktura



Obiekty *Colleague* i obiekt *Mediator* tworzą topologię gwiazdy. Komunikaty między obiektami *Colleague* są przekazywane za pośrednictwem mediatora

Wzorzec Mediator proponuje topologię gwiazdy, w której centrum znajduje się właśnie obiekt Mediator. Posiada on referencje do pozostałych obiektów (Colleague) i zna ich zakres odpowiedzialności. Komunikacja pomiędzy obiektami Colleague wymaga pośrednictwa Mediatora, który potrafi przekazać komunikat do właściwego odbiorcy.

Mediator: uczestnicy

- **Mediator**
  - definiuje interfejs dołączania i odłączania kolegów
- **Concrete Mediator**
  - implementuje mechanizm komunikacji pomiędzy obiektami *Colleague*
  - posiada referencje do zarejestrowanych obiektów *Colleague*
- **Colleague**
  - definiuje wspólny interfejs dla komunikujących się obiektów
  - posiada referencję do obiektu *Mediator*
  - komunikuje się z innymi obiektami za pośrednictwem obiektu *Mediator*

Mediator posiada metody służące do dołączania i odłączania obiektów Colleague. Ponadto jego zadaniem jest implementacja mechanizmu komunikacji, czyli podejmowanie decyzji który z obiektów Colleague powinien wykonać określone żądanie.

Obiekty Colleague nie są obciążone zadaniem komunikacji z pozostałymi obiektami. Ich wiedza jest ograniczona do znajomości Mediatora. Także dołączenie i odłączenie obiektu Colleague wymaga jedynie powiadomienia Mediatora, a nie wszystkich obiektów.

Struktura ta jest przybliżoną analogią do sieci komputerowych, w których komputery znajdujące się w różnych podsieciach komunikują się za pośrednictwem routera. Poszczególne Komputery nie muszą znać adresów wszystkich innych komputerów na świecie, a jedynie adres najbliższego routera.

Mediator: konsekwencje

- Centralizacja mechanizmu komunikacji
  - wyłączna odpowiedzialność obiektu *Mediator*
  - zmiana mechanizmu wymaga tylko zmiany *Mediatora*
  - prostota komunikacji vs. złożoność *Mediatora*
- Niezależność obiektów *Colleague* od siebie
- Uproszczenie protokołów obiektowych
  - Zamiana relacji wiele-wiele na relacje jeden-wiele

Mediator narzuca centralizację mechanizmu komunikacji. Odpowiedzialność za komunikację przejmuje w całości Mediator, co z jednej strony pozwala w łatwy sposób modyfikować go lub wymieniać, z drugiej jednak powoduje znaczny wzrost złożoności tego obiektu. Wydaje się jednak, że zamiana taka jest opłacalna, ponieważ uwalnia od problemów związanych z komunikacją resztę obiektów w systemie.

Drugą ważną zaletą wzorca jest uniezależnienie obiektów Colleague od siebie: nie posiadają one o sobie żadnej wiedzy, co pozwala modyfikować ich liczbę i funkcjonalność.

Mediator: przykład

```
public class Mediator {
   private boolean slotFull = false;
   private int number;

   public synchronized void storeMessage(int num) {
      while (slotFull) {
         try { wait(); } catch (InterruptedException ex ) { }
      }
      number = num;
      slotFull = true;
      notifyAll();
   }
   public synchronized int retrieveMessage() {
      while (slotFull) {
         try { wait(); } catch (InterruptedException ex ) { }
      }
      notifyAll();
      slotFull = false;
      return number;
   }
}
```


Prostym przykładem wzorca Mediator może być znany problem producentów i konsumentów. Producenci nie muszą posiadać jakiejkolwiek wiedzy o konsumentach, ponieważ ich zadaniem jest tylko zapełnianie bufora. Podobnie, konsumenci w żaden sposób nie zależą od producentów, a jedynie od bufora. Bufor pełni rolę mediatora, który koordynuje komunikację między dwoma typami obiektów.

Dzięki zastosowaniu wzorca Mediator możliwe jest zwiększanie lub zmniejszanie liczby producentów i konsumentów bez zmiany struktury systemu.

Mediator: przykład cd.

```
public class Producer extends Thread {
   private Mediator m;
   private int no;
   private static int count = 1;

   public Producer(Mediator m) {
      mediator = m;
      no = count++;
   }

   public void run() {
      int num;
      while (true) {
         m.storeMessage(num = (int)(Math.random()*100));
         System.out.print("p" + no + "-" + num + " ");
      }
   }
}
```


Slajd przedstawia kod klasy producenta. Jej logika jest zawarta w metodzie *run()*, która próbuje wstawić do bufora kolejną liczbę.

Mediator: przykład cd.

```
public class Consumer extends Thread {
   private Mediator m;
   private int no;
   private static int count = 1;

   public Consumer(Mediator m) {
      count = m;
      id = count++;
   }

   public void run() {
      while (true) {
         System.out.print("c" + no + "-"
            + m.retrieveMessage());
      }
   }
}
```

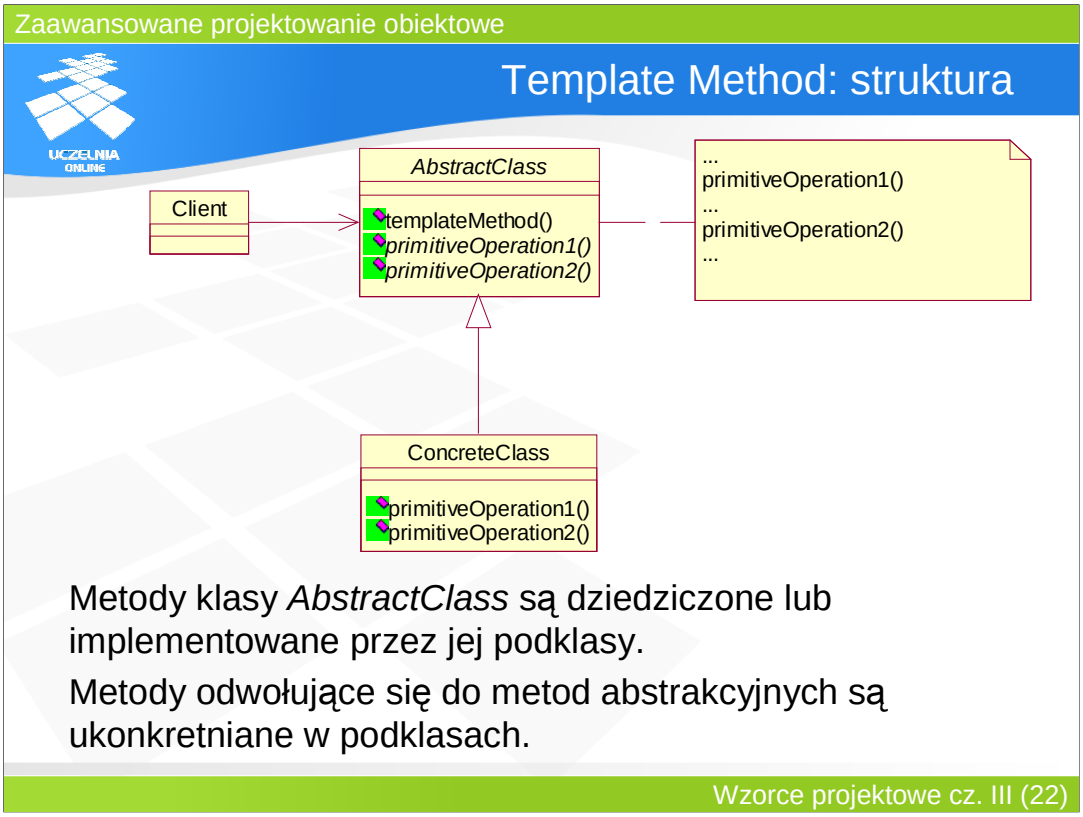

Klasa konsumenta, której kod został przedstawiony na slajdzie, próbuje w nieskończonej pętli odczytać wartość liczby przechowywanej w buforze Mediatora.

Template Method: cel

- Stworzenie szkieletu algorytmu w postaci klasy
- Przesunięcie niektórych operacji do podklas

E. Gamma et al. (1995)

Template Method jest prostym wzorcem opisującym sposób współpracy między nadklasą i jej podklasami. Celem zastosowania tego wzorca jest stworzenie szkieletu algorytmu w nadklasie i określenie jego poszczególnych kroków w podklasach.



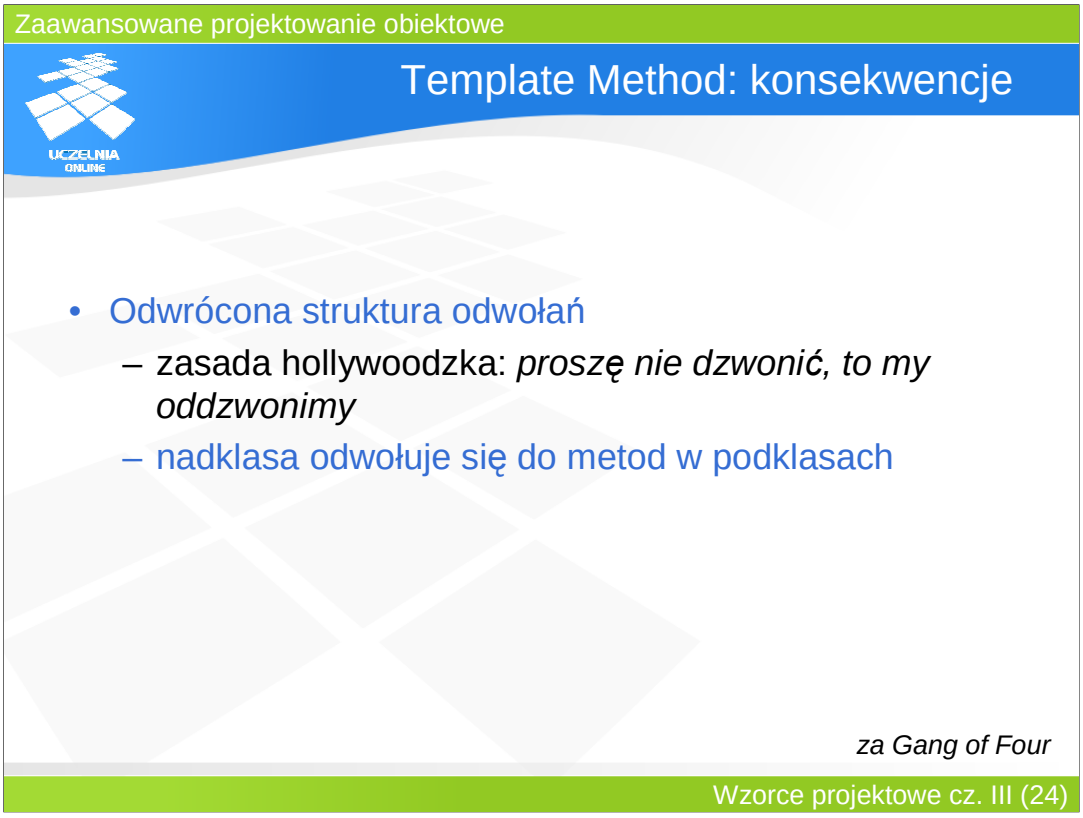
Klasa abstrakcyjna AbstractClass posiada metodę *templateMethod()* definiującą szkielet algorytmu. Metoda ta odwołuje się do innych metod w tej klasie definiujących podstawowe kroki algorytmu. Część z nich to metody wykorzystywane przez wszystkie podklasy, dlatego są one zdefiniowane w nadklasie i dziedziczone po niej przez podklasy. Ponieważ pozostałe kroki algorytmu mają różną postać w każdym algorytmie, dlatego na poziomie klasy AbstractClass są one deklarowane jako abstrakcyjne. Ich implementacja jest wówczas przesunięta do klas dziedziczących.

Template Method: uczestnicy

- **Abstract Class**
  - definiuje szkielet algorytmu w postaci metody
  - szkielet odwołuje się do prostych metod abstrakcyjnych
- **Concrete Class**
  - implementuje proste metody abstrakcyjne
  - pokrywa inne, wybrane metody odziedziczone z *AbstractClass*

Klasa abstrakcyjna definiuje szkielet algorytmu w postaci metody, która z założenia nie powinna być pokrywana w klasach potomnych i często jest deklarowana jako sfinalizowana. Podklasy muszą dostarczyć implementacji metod abstrakcyjnych nadklasy oraz mogą pokryć niektóre inne odziedziczone metody.

Instancja wybranej podklasy dziedziczy zatem szkielet algorytmu i część kroków algorytmu, oraz definiuje samodzielnie pozostałe kroki.



Analizując konsekwencje zastosowania tego wzorca, warto zauważyć, że taka struktura odwołań jest odwrotna w stosunku do typowej sytuacji, w której podklasa odwołuje się do swojej nadklasy w celu wykorzystania jej funkcji. W tym przypadku to nadklasa odsuwa implementację pewnych kroków algorytmu do podklas.

Oczywiście, w celu wykorzystania takiego rozwiązania należy utworzyć obiekt podklasy i wykonać jego metodę-szkielet algorytmu, jednak można tę klasę traktować w sposób abstrakcyjny jako pewną implementację interfejsu nadklasy.

Wzorzec Template Method jest powszechnie stosowany w implementacji różnego rodzaju sterowników i wtyczek.

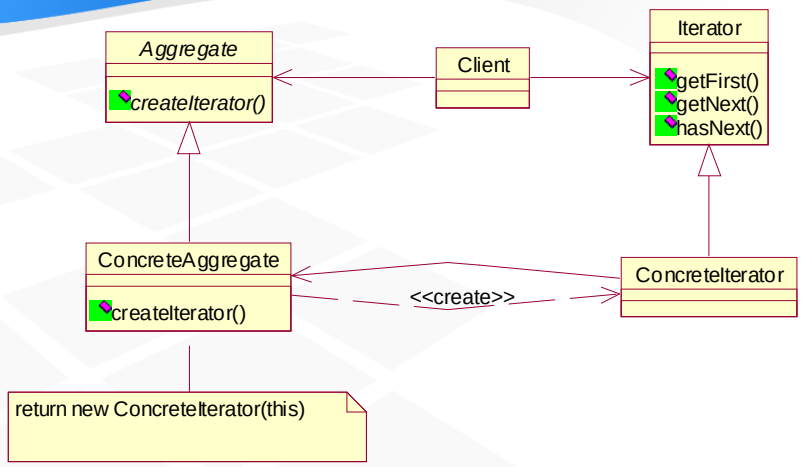
Iterator: cel

Umożliwienie sekwencyjnego dostępu do elementów kolekcji bez ujawniania jej wewnętrznej implementacji

E. Gamma et al. (1995)

Różnorodność kolekcji obiektowych (listy, kolejki, stosy, zbiory, multizbiory, mapy etc.), zarówno funkcjonalna, jak i implementacyjna, powoduje, że wiele z nich wymaga specyficznej obsługi i stosowania zróżnicowanych metod dostępu do elementów. Wzorzec Iterator odpowiada na potrzebę zunifikowanego dostępu do elementów kolekcji, który pozwoli pominąć różnice w ich implementacji. Dzięki niemu, niezależnie od rodzaju kolekcji, jej elementy mogą być przetwarzane sekwencyjnie, z zachowaniem własności poszczególnych kolekcji.

Iterator: struktura



Każdy *Aggregate* tworzy specyficzny dla siebie iterator.

Klient odwołuje się do niego poprzez abstrakcyjny interfejs.

Wzorzec Iterator składa się z dwóch klas abstrakcyjnych: Aggregate i Iterator, oraz dwóch klas konkretnych: ConcreteAggregate i ConcreteIterator. Wszystkie kolekcje są implementacją interfejsu Aggregate, tzn. posiadają metodę tworzącą iterator. Iterator, podobnie jak Aggregate, jest jedynie specyfikacją interfejsu, jaki każdy iterator musi posiadać. Klient, odwołując się do metody *createIterator()* w kolekcji, otrzymuje klasę implementującą interfejs Iterator. Dzięki temu klient nie zna konkretnej klasy implementacyjnej, a jedynie interfejs, do którego musi się odwoływać. Taka sytuacja ma miejsce np. w bibliotece Java Collections: każda kolekcji tworzy swój własny iterator, który jednak jest dostępny wyłącznie poprzez wspólny interfejs Iterator. W ten sposób mogą one być traktowane w jednolity sposób.

Iterator posiada wewnętrzny wskaźnik, który wskazuje na aktualny element kolekcji. Iteratory definiują podstawowe operacje pozwalające na sekwencyjny dostęp do wszystkich elementów dowolnej kolekcji: *getFirst()* – ustawiająca wskaźnik iteratora na początek kolekcji, *getNext()* – zwracająca kolejny element, *hasNext()* – sprawdzająca, czy kolejny element istnieje. W niektórych implementacjach iterator pozwala także na modyfikacje kolekcji, np. dodawanie i usuwanie elementów. Kolekcje o specyficznej strukturze, np. listy mogą udostępniać iteratory wykorzystujące wiedzę o tej strukturze, np. udostępniającą możliwość swobodnego dostępu do elementów kolekcji, zmiany kierunku trawersu kolekcji etc.

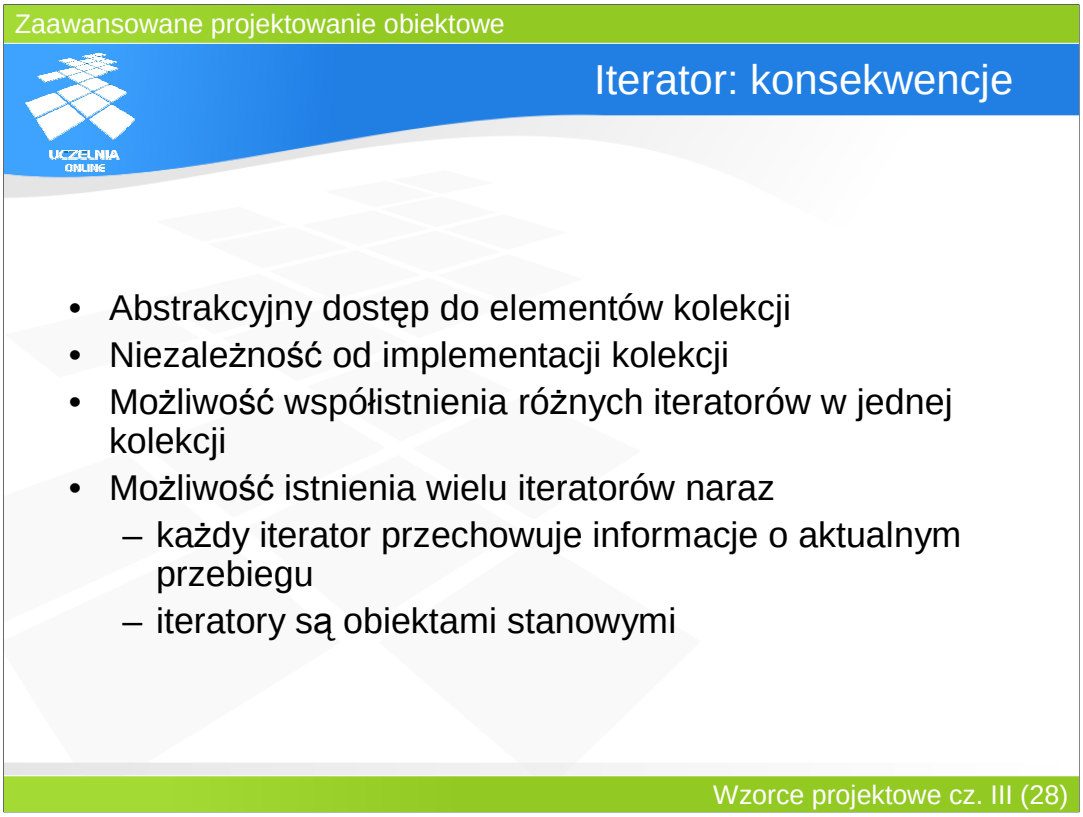
Z uwagi na konieczność dostępu do elementów kolekcji, iterator musi posiadać prawo odwołania się do nich. W praktyce jest on zatem zwykle klasą zaprzyjaźnioną lub wewnętrzną kolekcji.

Iterator: uczestnicy

- **Aggregate**
  - ogólny interfejs każdej kolekcji
  - deklaruje interfejs do tworzenia iteratora
- **Concrete Aggregate**
  - tworzy iterator specyficzny dla własnej struktury
- **Iterator**
  - definiuje interfejs sekwencyjnego dostępu do obiektu *Aggregate* (*getFirst()*, *getNext()*, *hasNext()*)
- **Concrete Iterator**
  - implemenetacja interfejsu *Iterator* specyficzna dla konkretnej kolekcji

We wzorcu uczestniczą dwie hierarchie obiektów: związanych z kolekcjami (Aggregate i jej klasy potomne) i związanych z iteracją (Iterator i jego podklasy). Obie hierarchie są powiązane ze sobą wyłącznie poprzez interfejsy.

Warto zwrócić uwagę, że struktura wzorca i role pełnione przez poszczególne klasy są szczególnym przypadkiem struktury i ról zdefiniowanych we wzoru Factory Method. Tam również klient odwołuje się do abstrakcyjnej metody klasy-fabryki w celu otrzymania abstrakcyjnego produktu, a faktycznie wywołuje metody w implementacji klasy-fabryki i otrzymuje konkretny produkt zależny od użytej fabryki.



Iterator pozwala na oddzielenie kolekcji, czyli klasy związanej z przechowywaniem obiektów, od mechanizmu dostępu do tych obiektów. Dzięki temu klient odwołuje się do obiektów w sposób abstrakcyjny, niezależny od konkretnej implementacji kolekcji.

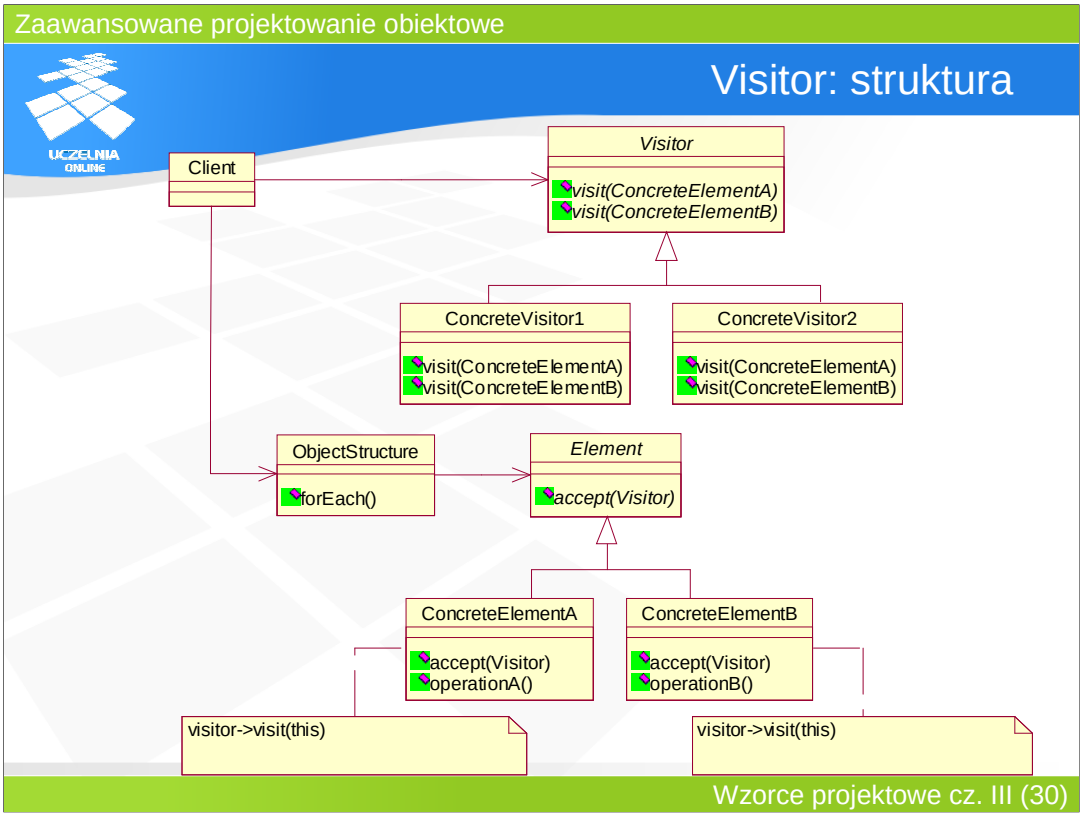
Konstrukcja iteratora pozwala na jednoczesne współistnienie wielu niezależnych iteratorów, ponieważ każdy przechowuje wewnętrznie wskaźnik do aktualnie wskazywanego obiektu w kolekcji. Niektóre kolekcje mogą definiować kilka różnych iteratorów, o zróżnicowanej funkcjonalności (w przypadku np. listy)

Visitor: cel

- Reprezentacja operacji do wykonania na elementach heterogenicznej struktury
- Realizacja operacji w sposób specyficzny dla typu odwiedzanego elementu
- Umożliwienie tworzenia nowych operacji bez konieczności modyfikacji klas wewnątrz struktury

E. Gamma et al. (1995)

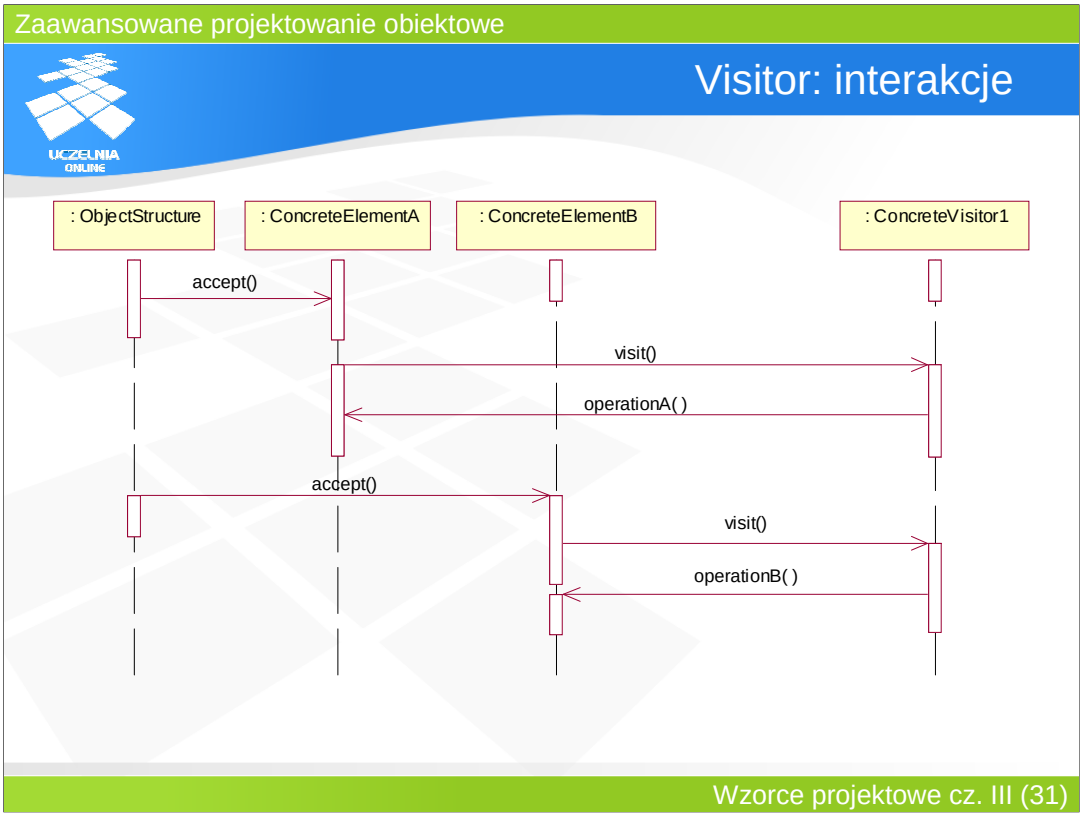
Wzorzec Visitor jest rozszerzeniem idei stojącej za wzorcem Iterator na kolekcje heterogeniczne, składające się z elementów różnych typów. Jego celem jest dostęp do każdego elementu takiej struktury i wykonanie na nim operacji w sposób dla niego specyficzny.



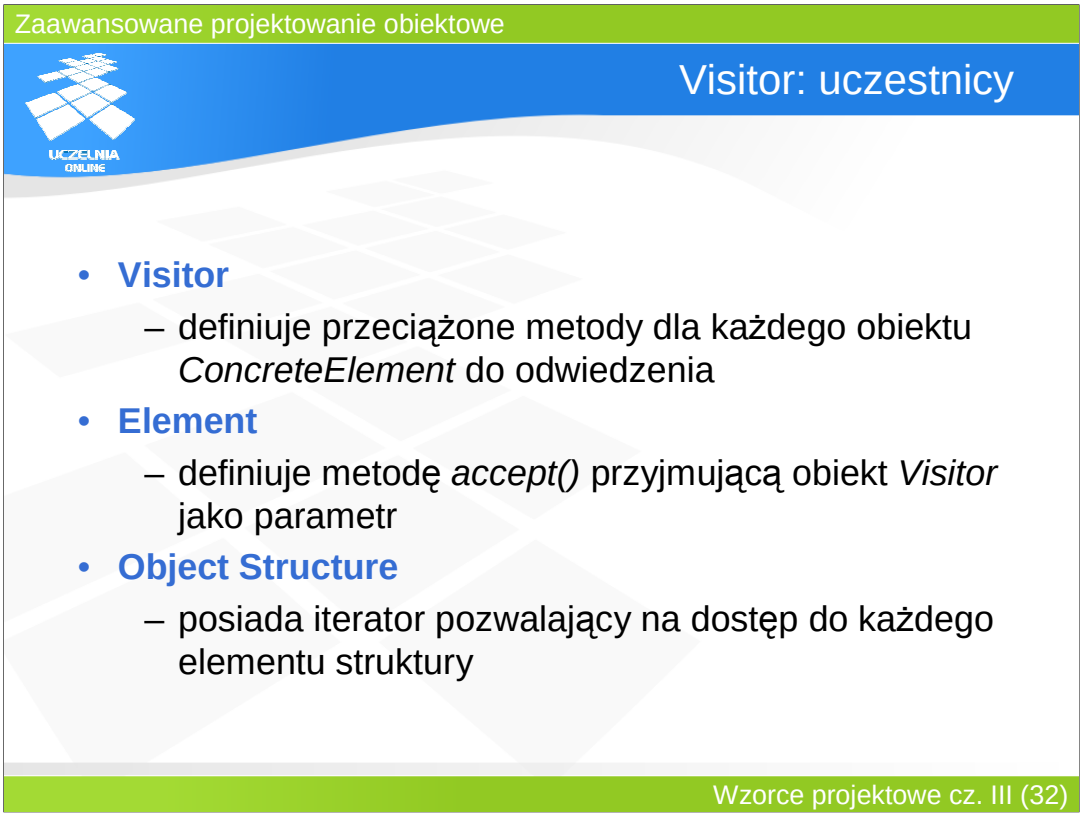
Obiekty wchodzące w skład struktury, które obiekt Visitor ma odwiedzić, implementują interfejs Element. Interfejs ten definiuje tylko jedną metodę – *accept(),* przyjmującą jako jedyny parametr obiekt typu Visitor. Co więcej, metoda ta w każdym przypadku ma identyczną postać: w obiekcie Visitor wywoływana jest metoda *visit()* z parametrem *this*, oznaczająca referencję do własnego obiektu. W ten sposób następuje tzw. odwrócenie sterowania: zamieniają się nadawca i odbiorca komunikatu

Z drugiej strony we wzorcu uczestniczą obiekty implementujące interfejs Visitor, który definiuje grupę metod *visit(),* przyjmujących jako parametr obiekty typów, które należy odwiedzić. Każda z tych metod jest zatem przygotowana do odwiedzenia obiektu jednego typu w sposób całkowicie zależny od niego i do niego dostosowany.

Jednak konieczne jest automatyczne określenie, która z metod *visit()* zostanie wykonana. Do tego służy właśnie owo odwrócenie sterowania: przekazywany parametr *this* jest zawsze typu klasy, w której się znajduje, dzięki czemu metoda zostanie dopasowana automatycznie do typu parametru.



Obiekt ObjectStructure po kolei wywołuje na każdym ze znajdujących się w nim obiektów Element metodę *accept(),* przekazując jako parametr obiekt Visitor. Odwiedzany element, jeżeli zgadza się na odwiedziny, wywołuje na obiekcie Visitor metodę *visit(),* przekazując referencję do samego siebie, reprezentowaną przez *this*. W ten sposób przekazuje mu swoją zgodę na odwiedziny, a jednocześnie przekazuje referencję do siebie, co umożliwia wykonanie na nim dowolnych publicznych metod (*operationA(), operationB()*).



Rolą interfejsu Visitor jest zdefiniowanie przeciążonych metod dla każdego obiektu typu ConcreteElement, który należy odwiedzić. Zatem Visitor jako cały obiekt reprezentuje pewną operację, którą należy wykonać na wszystkich elementach struktury danych, w sposób od nich zależny.

Elementy tej struktury posiadają tylko jedną wspólną metodę *accept(),* która przyjmuje parametr typu Visitor i umożliwia mu (poprzez odwrócenie sterowania) wywołanie metod obiektu *Element*. Dodatkową zaletą tego rozwiązania jest możliwość zabezpieczenia się niektórych obiektów Element przed odwiedzeniem przez konkretny Visitor, co może mieć znaczenie w niektórych obszarach zastosowań.

Obiekt ObjectStructure jest kolekcją, która udostępnia wszystkie swoje elementy i pozwala przekazywać im obiekt Visitor.

Visitor: konsekwencje

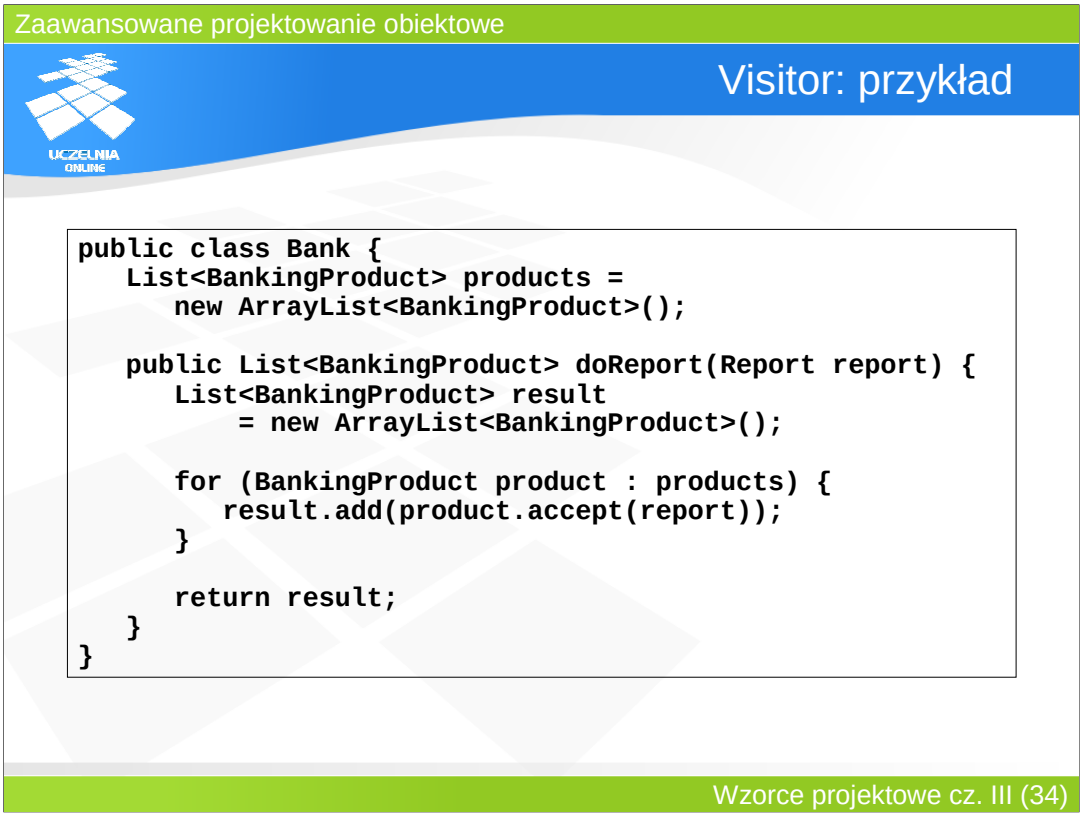
- Możliwość odwiedzenia obiektów niespokrewnionych ze sobą
- Łatwe dodawanie nowych obiektów *Visitor*
- Utrudnione dodawanie nowych typów *Element*
  - konieczność modyfikacji klasy *Visitor* i jej podklas
- Możliwość zbierania informacji z elementów struktury w sposób dla nich specyficzny
- Naruszenie hermetyzacji obiektów *Visitor* i *Element*
  - obiekty muszą odwoływać się do swoich publicznych metod

Najważniejszą zaletą tego wzorca jest możliwość wykonania przy odwiedzinach każdego elementu w strukturze pewnego kodu zależnego od typu tego elementu. Ta cecha decyduje o popularności i szerokich możliwościach stosowania tego wzorca.

Dodawanie nowych implementacji interfejsu Visitor jest łatwe, ponieważ wymaga jedynie stworzenia nowej klasy i zaimplementowania jej metod. Z drugiej strony, dodanie nowego elementu do odwiedzenia jest trudne, ponieważ wymaga dodania nowej metody do wszystkich obiektów Visitor.

Wzorzec ten w szczególności nadaje się do akumulacji stanu podczas przejścia przez strukturę obiektów (akumulacja odbywa się wówczas wewnątrz obiektu Visitor).

Należy jednak zwrócić uwagę, że wzajemne wywoływanie metod *visit()* i *accept()* w obiektach Visitor i Element wymaga, aby metody te były dla siebie wzajemnie dostępne. W C++ można wykorzystać w tym celu klasy zaprzyjaźnione, natomiast w Javie konieczne jest upublicznienie metod, co w pewnym stopniu narusza ich hermetyzację.



Przykładem zastosowanie wzorca Visitor może być sposób wykonywania raportów bankowych na podstawie wszystkich produktów bankowych, jakie są uruchomione w banku.

W banku wykonywane są rozmaite raporty, wymagające inspekcji każdej instancji produktu bankowego, jaka jest prowadzona w banku. Metoda *doReport()* przyjmuje obiekt raport (czyli właśnie obiekt Visitor) i następnie przekazuje go każdemu produktowi jako parametr metody *accept().* Wyniki tej metody (domyślnie – referencja do tego rachunku lub wartość *null*) jest dołączana do wynikowej listy raportu.

Visitor: przykład cd.

```
public abstract class BankingProduct {
}

public interface Element {
   public BankingProduct accept(Report report);
}

public class Account extends BankingProduct implements Element {
   public BankingProduct accept(Report report) {
      if (isPriviliged(report)) {
         return report.visit(this);
      }
      return null;
   }
}

public class Credit extends BankingProduct implements Element {
   public BankingProduct accept(Report report) {
      return report.visit(this);
   }
}
```


Dwie klasy reprezentujące produkty bankowe: Account i Credit implementują metodę *accept()*. W przypadku klasy Account wymagane jest uprawnienie weryfikowane przez metodę *isPriviliged()*, natomiast w przypadku kredytu weryfikacja ta nie jest przeprowadzana

Visitor: przykład cd.

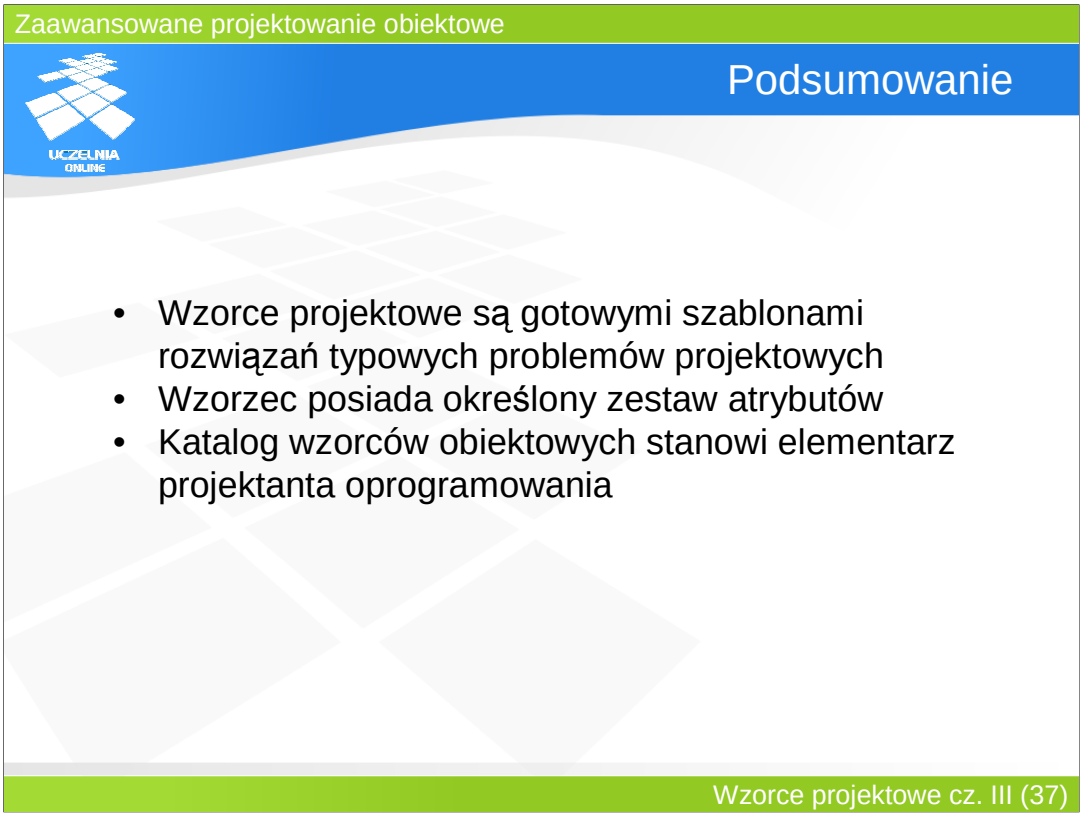
```
public class Over1000Report implements Visitor {
   public BankingProduct visit(Account acc) {
      if (acc.balance > 1000)
         return acc;
      return null;
   }
   public BankingProduct visit(Credit credit) {
      if (credit.draft > 1000 && credit.isActive())
        return credit;
      return null;
   }
}
```

```
public class PassAllReport implements Visitor {
   public BankingProduct visit(Account acc) {
      return this;
   }
   public BankingProduct visit(Credit credit) {
      return this;
   }
}
```


Raport Over1000Report służy do zebrania danych o produktach bankowych o wartości powyżej 1000 PLN. W przypadku odwiedzin obiektu Account wartość 1000 PLN odnosi się do pola *balance*, natomiast warunkiem odwiedzenia obiektu Credit jest jego aktywacja (metoda *isActive()*) i wartość pola *draft* wynosząca powyżej 1000.

Drugi raport, PassAllReport, służy do zestawienia wszystkich produktów bankowych bez względu na ich właściwości, dlatego nie dokonuje on żadnej weryfikacji.

Wykonanie metody *doReport()* z obiektem Over1000Report jako parametrem zwróci listę produktów bankowych zawierającą jedynie te z nich, których charakterystyka jest zgodna z odpowiednimi metodami *visit()* tego raportu, natomiast wykonanie raportu PassAllReport zwróci pełną listę produktów uruchomionych w banku.



Podczas trzech wykładów przedstawiono genezę wzorców projektowych oraz ich typową strukturę. Największą częścią wykładu był przegląd wzorców zaproponowanych przez Bandę Czterech wraz z przykładami. Użycie sprawdzonych rozwiązań, jakimi są wzorce projektowe, pozwala lepiej projektować oprogramowanie obiektowe.