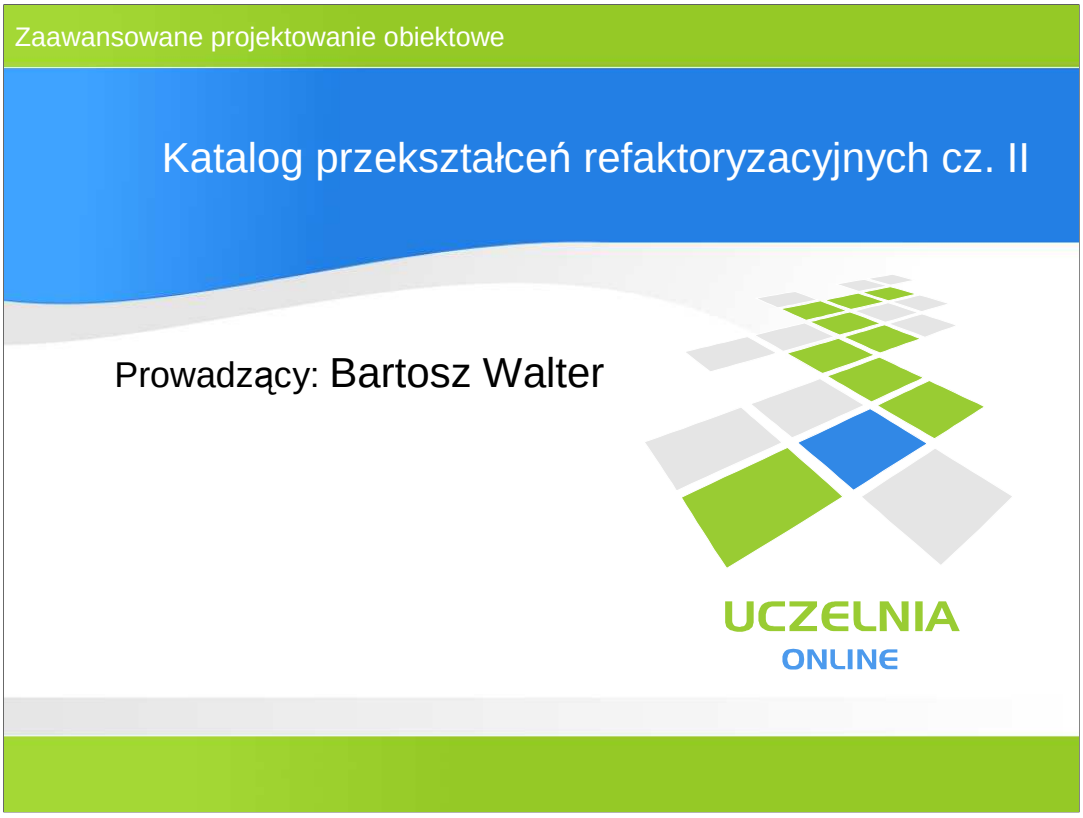
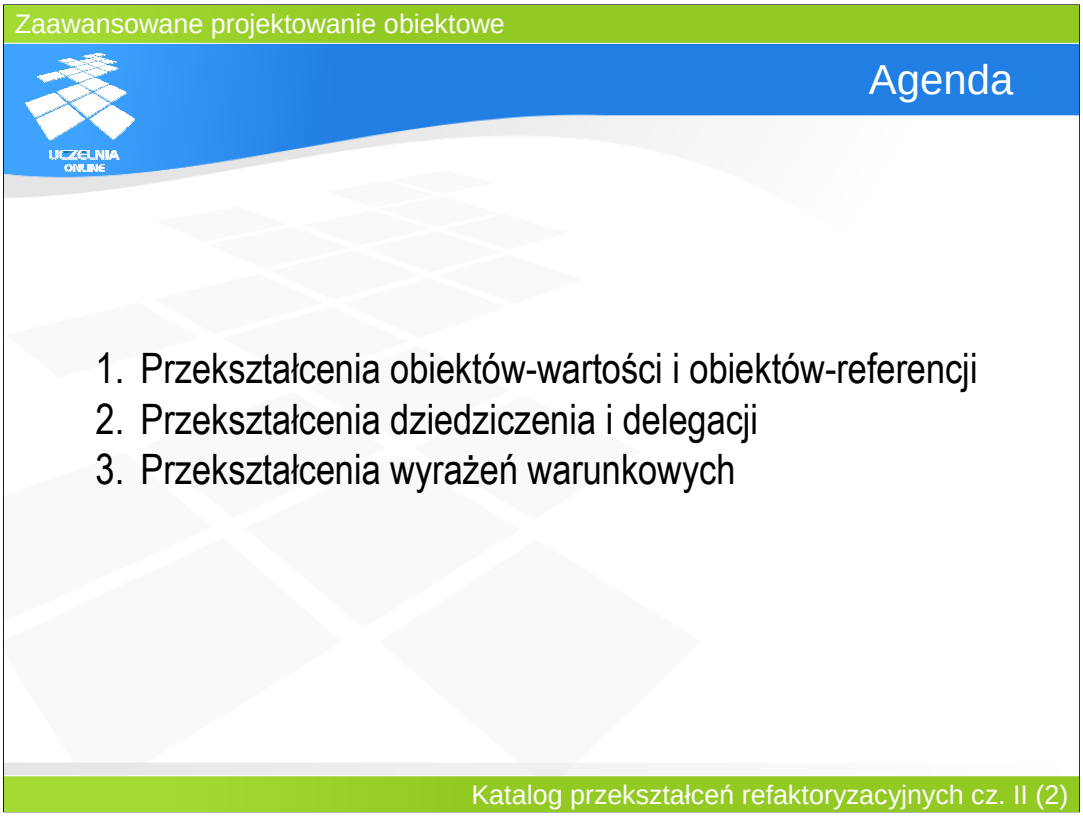
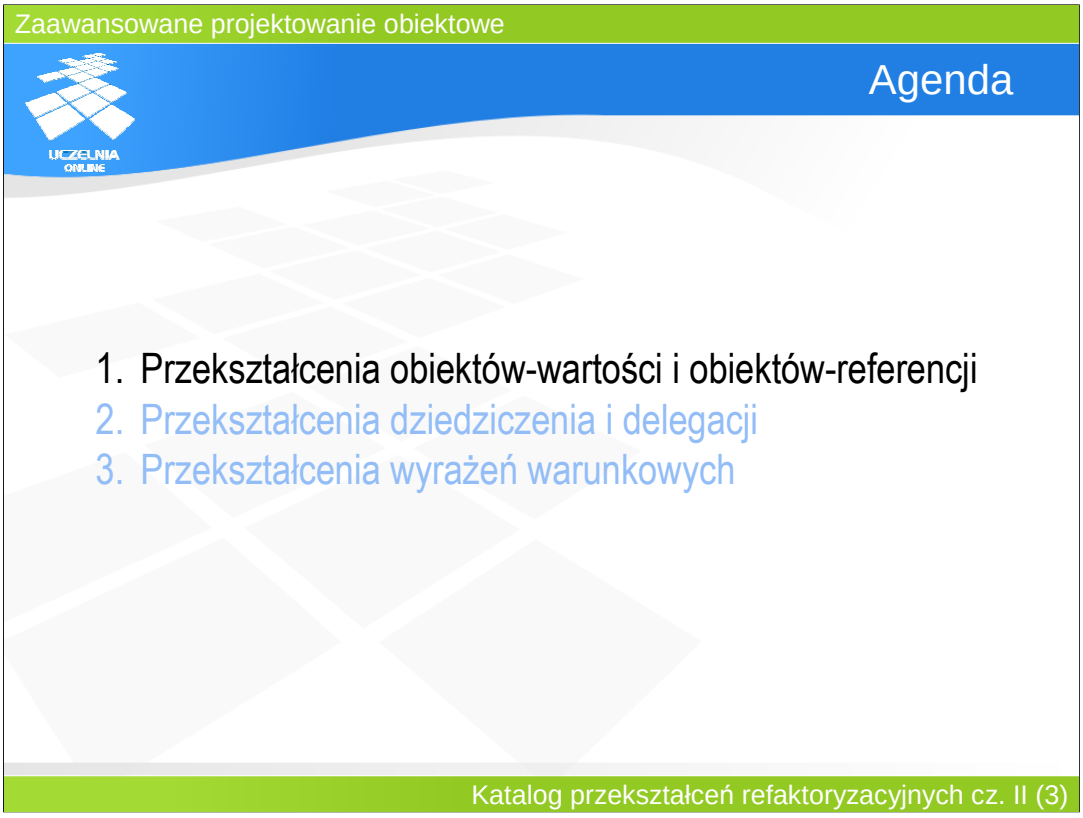
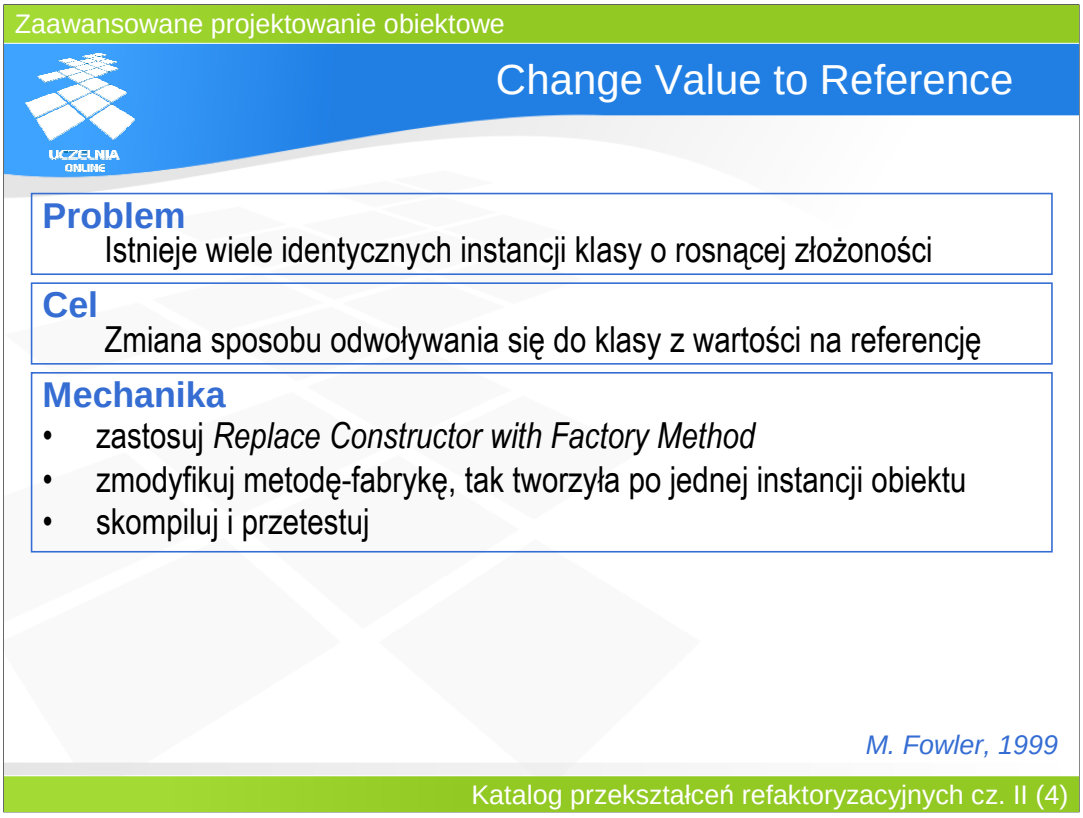

Zaawansowane projektowanie obiektowe
Katalog przekształceń refaktoryzacyjnych cz. II
Prowadzący: Bartosz Walter
UCZELNIA
ONLINE



Jest to drugi wykład poświęcony przeglądowi przekształceń refaktoryzacyjnych. W jego trakcie zostaną przedstawione trzy grupy przekształceń: dotyczących zamiany obiektów-wartości w obiekty-referencje (oraz przekształceń odwrotnych), przekształceń relacji dziedziczenia w relację delegacji oraz przekształceń wyrażeń warunkowych.



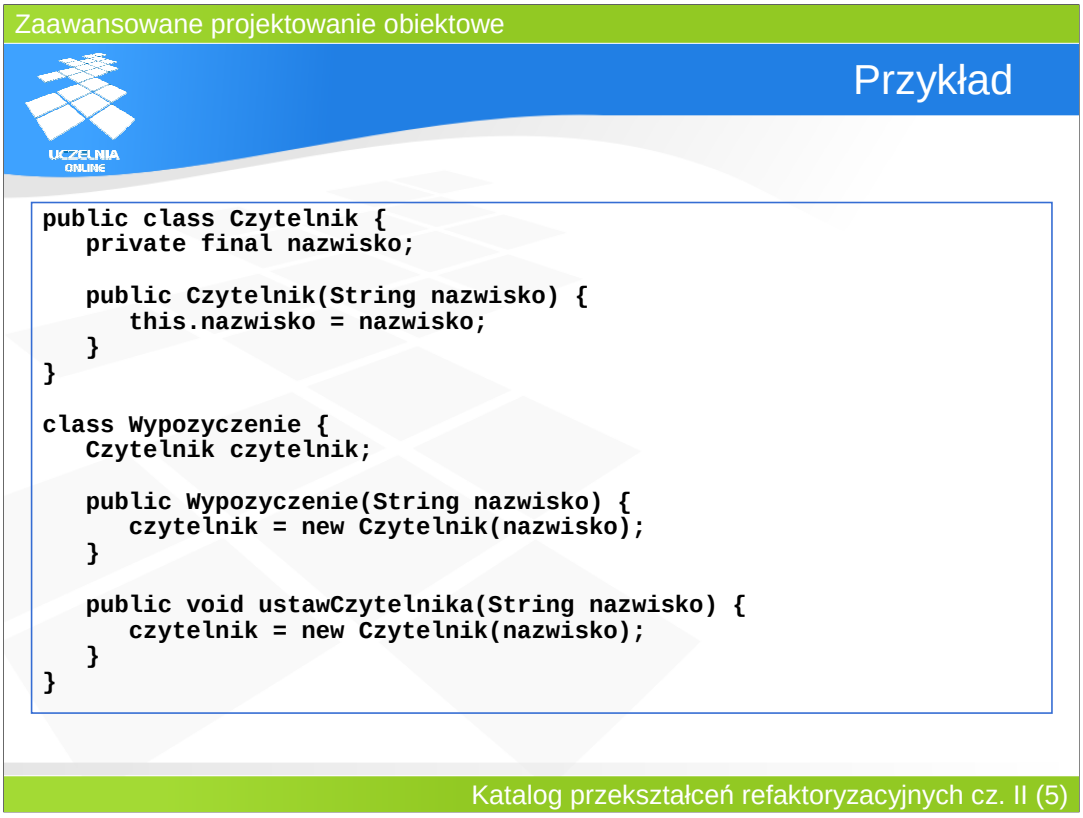
Pierwsza część wykładu będzie obejmowała jedynie dwa przekształcenia, zmieniające sposób odwoływania się do obiektu z wartości na referencję i z referencji na wartość. Znaczenie oraz interpretacja tych dwóch rodzajów obiektów (lub sposobów odwoływania się do nich) zostały przedstawione podczas pierwszego wykładu.



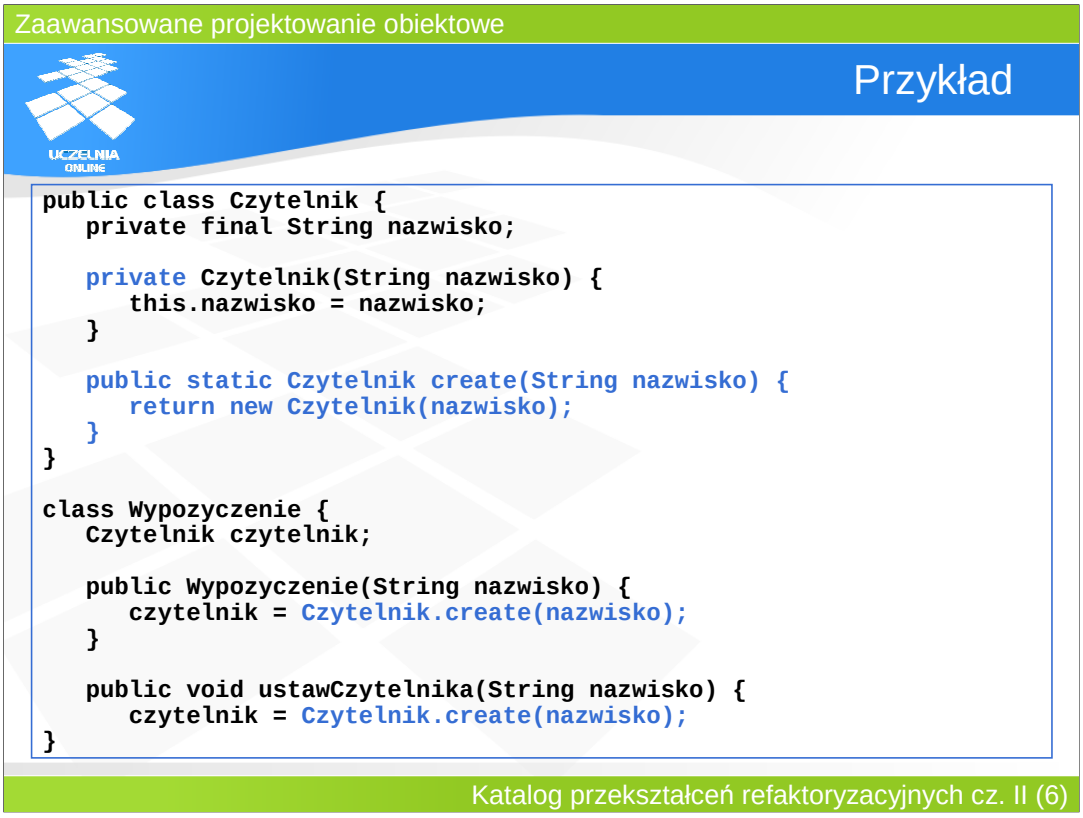
Przekształcenie to służy do zmiany obiektu traktowanego jako wartość w obiekt-referencję. Podczas pierwszego wykładu (zob. wprowadzenie do przedmiotu) przedstawiono różnicę pomiędzy tymi dwoma rodzajami obiektów: obiekty-wartości reprezentują niewielkie obiekty rzeczywiste i ich tożsamość (czyli cechę, która odróżnia je od innych obiektów) jest wyznaczana na podstawie ich stanu, to znaczy wartości pól (co oznacza, że obiektów tych nie można porównywać za pomocą operatora ==, który sprawdza identyczność referencji, a nie stanu obiektu!). Może istnieć wiele obiektów-wartości reprezentujących ten sam obiekt rzeczywisty, jednak właśnie ze względu na niemożność synchronizacji ich stanu nie mogą one zmieniać go po ich utworzeniu.

Obiekty-referencje zachowują się w przeciwny sposób: ponieważ może istnieć tylko jeden obiekt reprezentujący obiekt rzeczywisty, dlatego nie ma konieczności porównywania ich poprzez stan; można w tym celu wykorzystać referencje i łatwiejszy w użyciu operator ==. Nie ma również zakazu modyfikacji obiektu, ponieważ problem synchronizacji nie istnieje.

Przekształcenie rodzaju obiektu z wartości na referencję zwykle jest efektem wzrostu złożoności obiektu i wysokiego kosztu utworzenia nowych instancji. Dlatego mechanika tego przekształcenia polega na zastąpieniu możliwości tworzenia instancji obiektu poprzez wywołanie konstruktora metodą fabryką. W kolejnym kroku jest ona wyposażona w pamięć podręczną, w której przechowuje utworzone dotychczas obiekty. Dzięki temu można wykorzystać je ponownie do obsługi kolejnych żądań utworzenia obiektów.

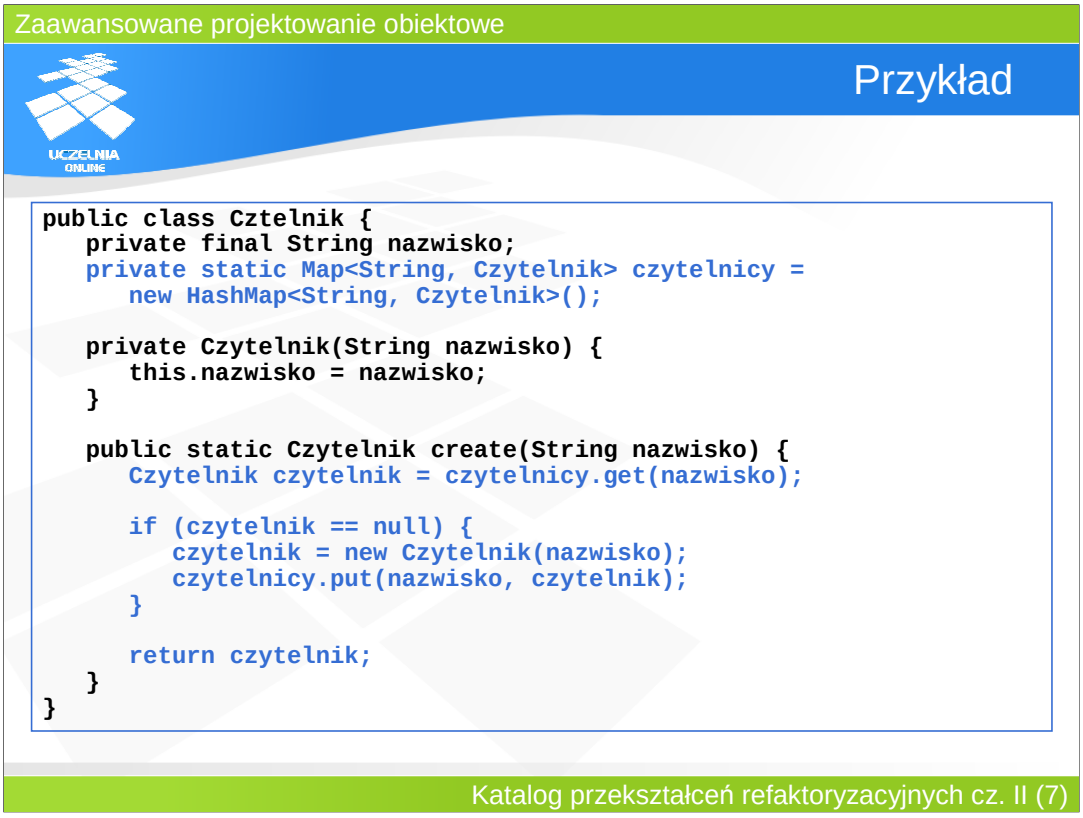


Przykład przedstawia relację pomiędzy klasami Czytelnik i Wypożyczenie. Obiekt klasy Wypożyczenie przechowuje referencję do obiektu Czytelnik, jednak podczas tworzenia swojej instancji, a także zmieniając przypisanego Czytelnika, zawsze tworzy nową instancję tej klasy. Zatem obiekt Czytelnik jest traktowany jako obiekt-wartość, mimo że domniemana wysoka złożoność tego obiektu nie uzasadnia takiej decyzji. Dlatego konieczne jest wykonanie przekształcenia, które zmieni sposób tworzenia i odwoływania się do obiektów klasy Czytelnik.



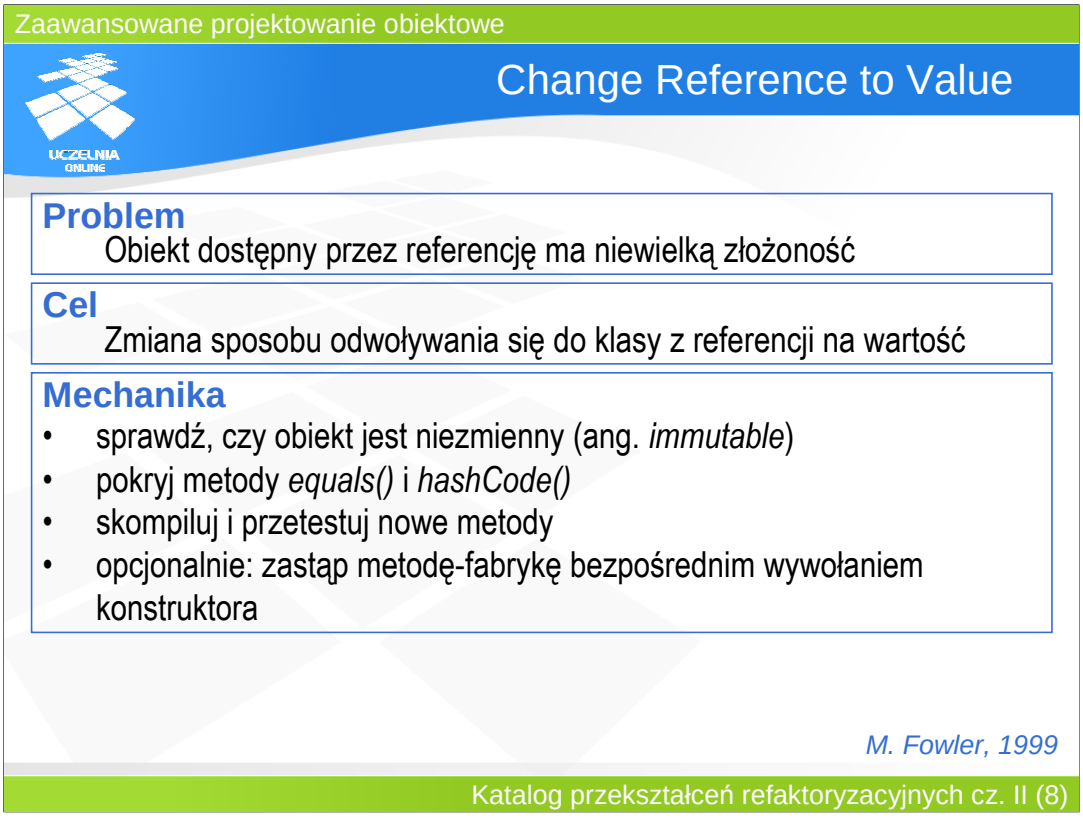
Pierwszym krokiem jest ukrycie konstruktora klasy Czytelnik i wprowadzenie na jego miejsce metody-fabryki, która przejmie odpowiedzialność za tworzenie obiektów. Metoda-fabryka w tym przypadku to metoda statyczna, która przyjmuje jako parametr nazwisko Czytelnika (mogłaby przyjmować także inne dane dotyczące stanu tego obiektu), a następnie na tej podstawie zwraca instancję tego obiektu. Na tym etapie metoda-fabryka po prostu wywołuje konstruktor (co oznacza, że na razie nie zachodzi żadna zmiana w zachowaniu programu).

Odpowiednio do tej zmiany dostosowane są metody w klasie Wypożyczenie: tworzenie obiektów klasy Czytelnik wymaga teraz wywołania metody-fabryki, a nie jej konstruktora.



Następnie klasa Czytelnik jest wyposażona w pamięć podręczną, implementowaną jako mapa odwzorowująca napisy (nazwiska czytelników) w obiekty klasy Czytelnik. Metoda-fabryka obecnie próbuje najpierw znaleźć żądany obiekt w pamięci podręcznej, a dopiero w przypadku niepowodzenia utworzyć go za pomocą wywołania konstruktora i następnie umieścić obiekt w pamięci podręcznej, zwracając wynik klientowi (w tym przypadku będzie nim obiekt klasy Wypożyczenie).

W ten sposób ograniczono liczbę tworzonych instancji klasy Czytelnik do jednej na każdy obiekt rzeczywisty oraz uniemożliwiono bezpośredni dostęp do konstruktora tej klasy, tworząc w efekcie obiekt-referencję. Obiekty klasy Czytelnik są zatem traktowane jako referencje, a nie wartości.



Potrzeba wykonania odwrotnego przekształcenia dotyczy zwykle sytuacji, w której obiekt dostępny przez referencję traci stopniowo swoją funkcjonalność, jego zakres odpowiedzialności staje się coraz mniejszy, a proces tworzenia obiektu nie wymaga istotnych nakładów czasowych. Wówczas warto zmienić taki obiekt w obiekt-wartość, co uprości sposób posługiwania się nim.

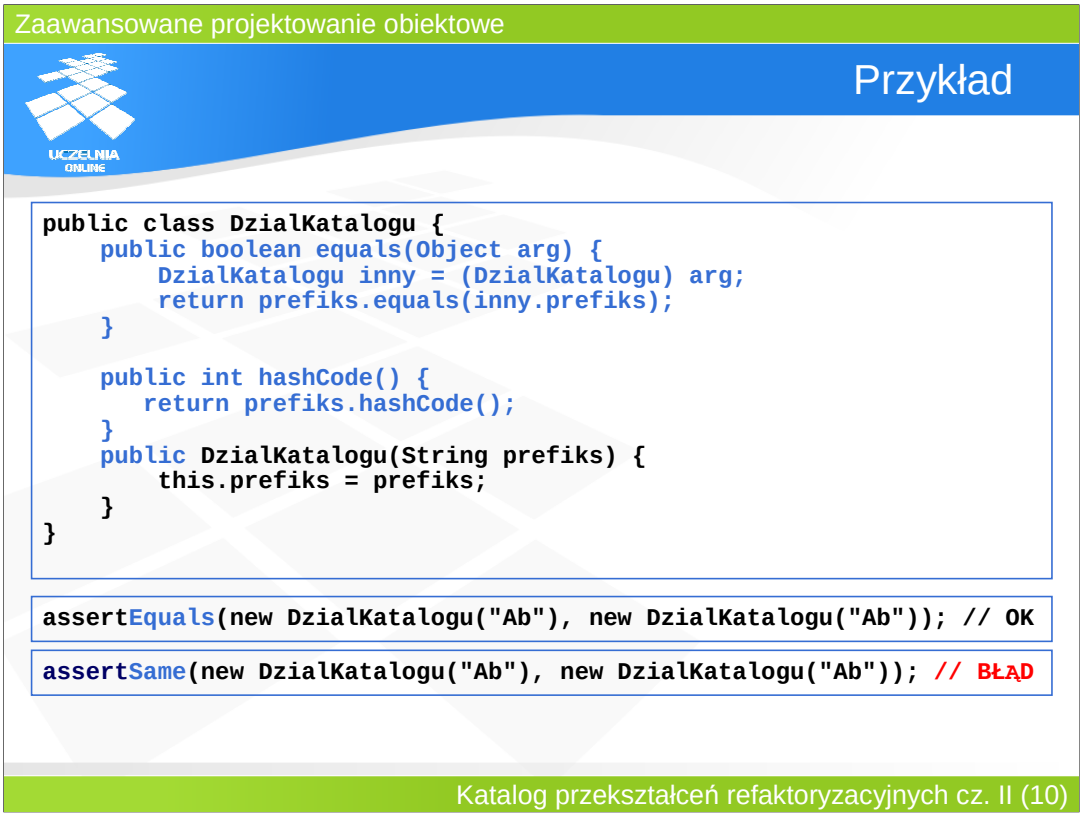
Mechanika przekształcenia składa się z następujących kroków: Najpierw należy sprawdzić, czy obiekt ten po przekształceniu faktycznie może być niezmienny. Jeżeli ten warunek nie jest spełniony, wówczas przekształcenie nie może być poprawnie zakończone. Jeżeli jednak tak jest, wtedy należy przygotować obiekt do porównywania jego stanu z innymi obiektami, tzn. zaimplementować jego dwie metody: *equals(),* służącą do bezpośrednich porównań, oraz *hashCode(),* często wykorzystywaną w tym celu metodę pomocniczą, która oblicza skrót obiektu, służący jako podstawa do porównań.

Po wykonaniu tych dwóch kroków obiekt w zasadzie może być traktowany jako obiekt-wartość. Opcjonalnie można zakończyć przekształcenie upubliczniając konstruktor, aby było możliwe bezpośrednie tworzenie instancji tej klasy, bez pośrednictwa metody-fabryki.

Przykład

```
public class DzialKatalogu {
    private String prefiks;
    private static Map<String, DzialKatalogu> dzialy =
        new HashMap<String, DzialKatalogu>();

    private  Dzial (String prefiks) {
        this.prefiks = prefiks;
    }
    public String prefiks() {
        return prefiks;
    }
    public static DzialKatalogu get(String prefiks) {
        DzialKatalogu dzial = dzialy.get(prefiks);
        if (dzial == null) {
            dzial = new DzialKatalogu(prefiks);
            dzialy.put(prefiks, dzial);
        }
        return dzial;
    }
}
```

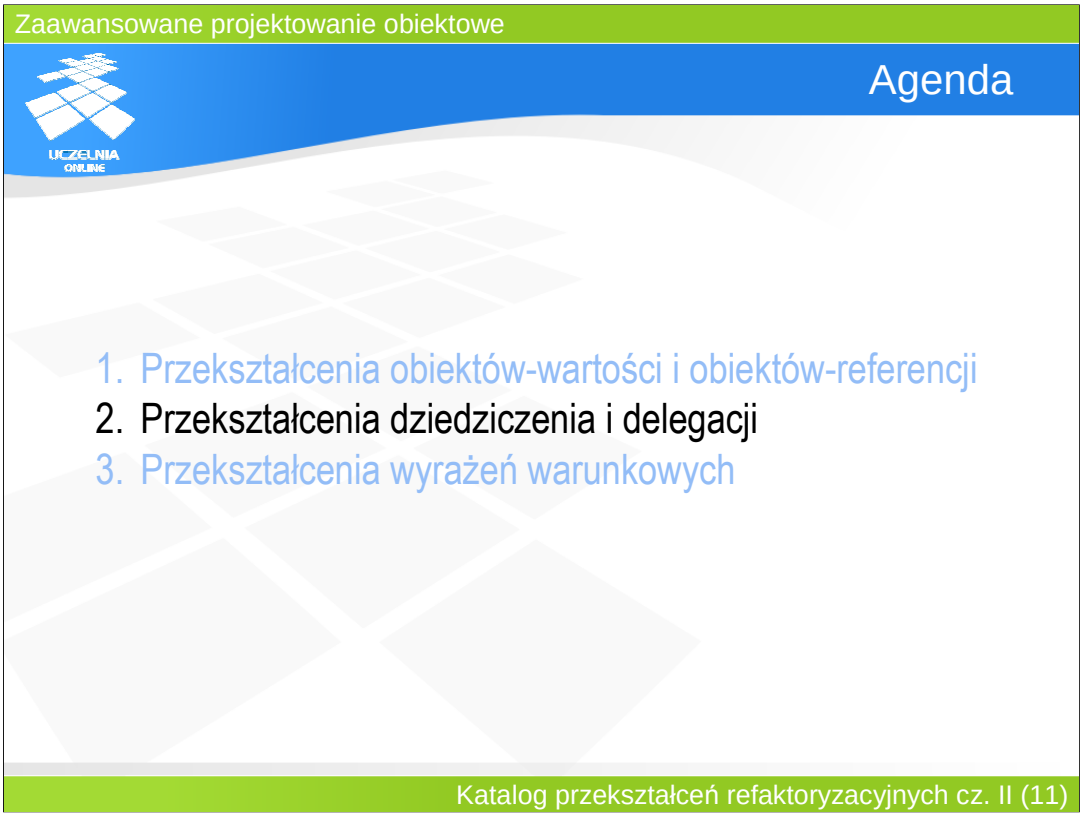

Przykład dotyczy klasy DziałKatalogu. Reprezentuje ona dział katalogu w bibliotece, jednak jedynym jego wyróżnikiem (czyli zakresem odpowiedzialności) jest prefiks dodawany przed identyfikatorem książki. Działy z zasady nie zmieniają swoich prefiksów, dlatego obiekt ten spełnia warunek niezmienności. Zgodnie z zasadami dotyczącymi obiektów-referencji, przypomnianymi przy poprzednim przekształceniu, klasa ta obecnie posiada metodę-fabrykę zajmującą się tworzeniem i zapamiętywaniem utworzonych instancji.



Przekształcenie polega na zaimplementowaniu dwóch metod: *equals()* oraz *hashCode().* Metoda *equals()* przyjmuje jako argument obiekt klasy Object i traktuje go jak inny obiekt własnej klasy (w tym przypadku będzie to DziałKatalogu). Porównanie własnego obiektu z obiektem przekazanym jako argument może być zrealizowane np. poprzez porównanie prefiksów opisujących działy książek, które są jedynym polem w tej klasie. Metoda *hashCode()* jest zaimplementowana podobnie – poprzez delegację do identycznej metody w obiekcie prefiksu.

Ostatnim krokiem jest zmiana kwalifikatora dostępu do konstruktora na public.

Warto pamiętać, że otrzymany obiekt-wartość nie może być już porównywany za pomocą operatora porównania, dlatego wyrażenia dotyczące porównania powinny również zostać dostosowane do tej zasady. Przedstawione dwa przypadki testowe pokazują ideę tego przekształcenia: porównanie za pomocą metody *equals()* nadal jest poprawne, natomiast za pomocą operatora == już nie.



Druga część dzisiejszego wykładu będzie dotyczyć zagadnień związanych z przekształcaniem relacji dziedziczenia i delegacji. Podczas pierwszego wykładu przedstawiono znaczenie, jakie posiadają oba typy relacji, i sposób ich wykorzystania. Niewłaściwe użycie każdej z relacji powoduje potrzebę dostosowania jej do faktycznych okoliczności.

Change Unidirectional Association with Bi

Problem

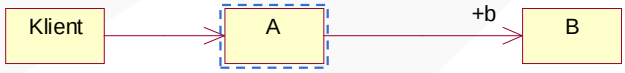
Jednokierunkowa asocjacja między klasami o podobnej wadze

Cel

Dodanie asocjacji powrotnej

Mechanika

- utwórz w klasie *B* pola przechowujące referencje powrotne (typu *A* dla krotności 1, typu *Set<A>* dla krotności n)
- utwórz metodę aktualizującą (setter) referencje po stronie kontrolowanej
- zmień metodę aktualizującą po stronie kontrolującej, tak aby wywoływała metodę aktualizującą po stronie kontrolowanej



M. Fowler, 1999

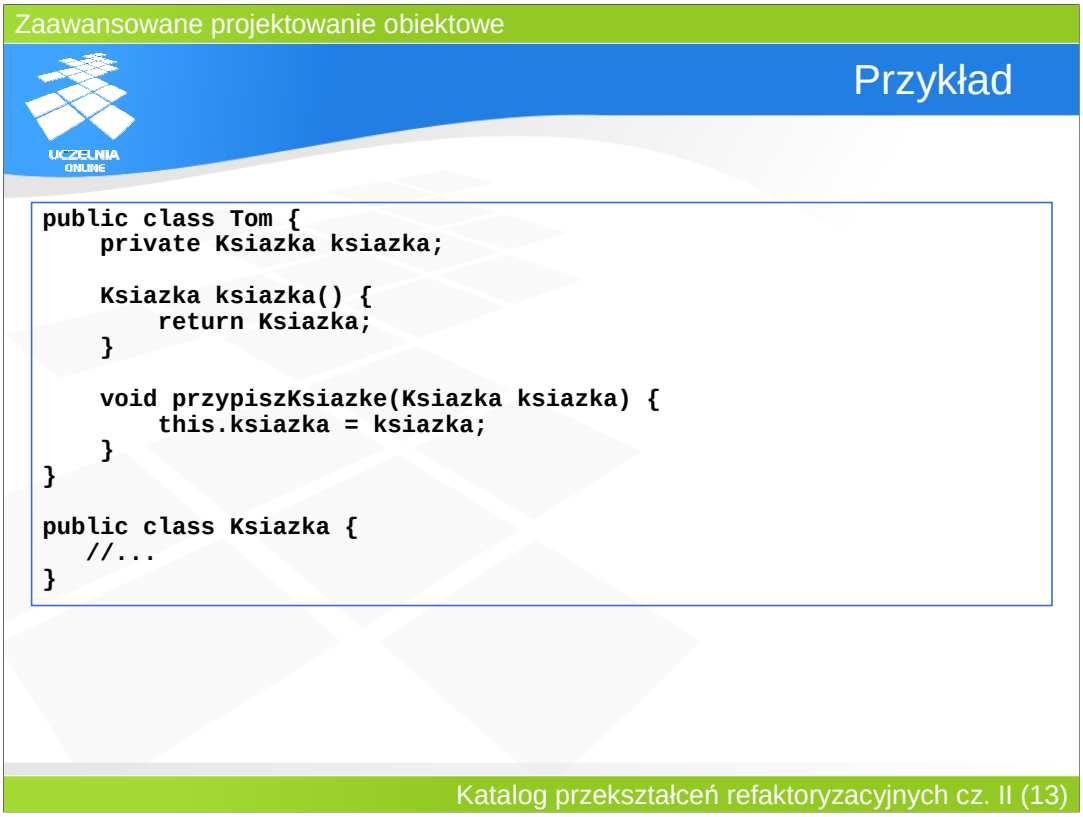
Przekształcenie to dotyczy zmiany sposobu nawigacji wewnątrz relacji asocjacji z jednokierunkowej na dwukierunkową. W asocjacji takiej wyróżnia się dwie strony: kontrolującą, do której dostęp ma klient i która zarządza relacją, oraz kontrolowaną, odgrywającą bardziej pasywną rolę. Na rysunku rolę kontrolującą (oznaczoną niebieską obwódką) odgrywa obiekt A, natomiast rolę kontrolowaną – obiekt B.

Przekształcenie polega na stworzeniu po stronie kontrolowanej referencji do obiektu kontrolującego oraz wprowadzeniu mechanizmu synchronizującego referencje po obu stronach relacji.

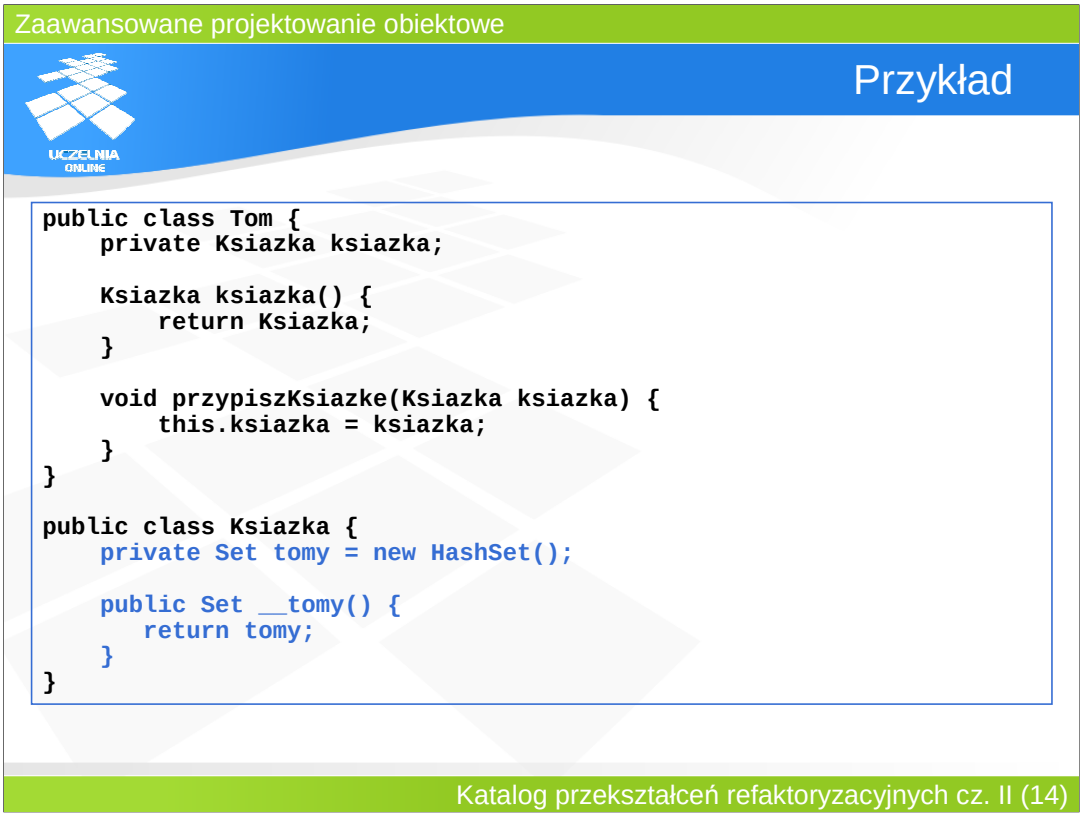
Pierwszym krokiem jest wprowadzenie pola, które będzie przechowywać referencje powrotne typu A. W przypadku relacji o krotności 1 wystarczy zwykła referencja typu A, natomiast w przypadku krotności n typem tym może być Set<A>, czyli zbiór referencji typu A.

Następnie należy utworzyć po stronie kontrolowanej metodę, która będzie wywoływana przez stronę kontrolującą w celu aktualizacji referencji. Metoda ta powinna być niedostępna bezpośrednio dla klienta, dlatego w języku C++ należy ją uczynić prywatną, a następnie zaprzyjaźnić z klasą A. Ponieważ jednak w języku Java pojęcie klas zaprzyjaźnionych nie istnieje, dlatego konieczne jest zastosowanie innego rozwiązania, np. nazwanie metody w sposób jednoznacznie wskazujący na jej przeznaczenie tylko do użytku klasy A.

Ostatnim krokiem jest modyfikacja metody w klasie A, która odpowiada za dodawanie obiektów typu B, tak aby za pomocą metody w klasie B aktualizowała także referencje powrotne.

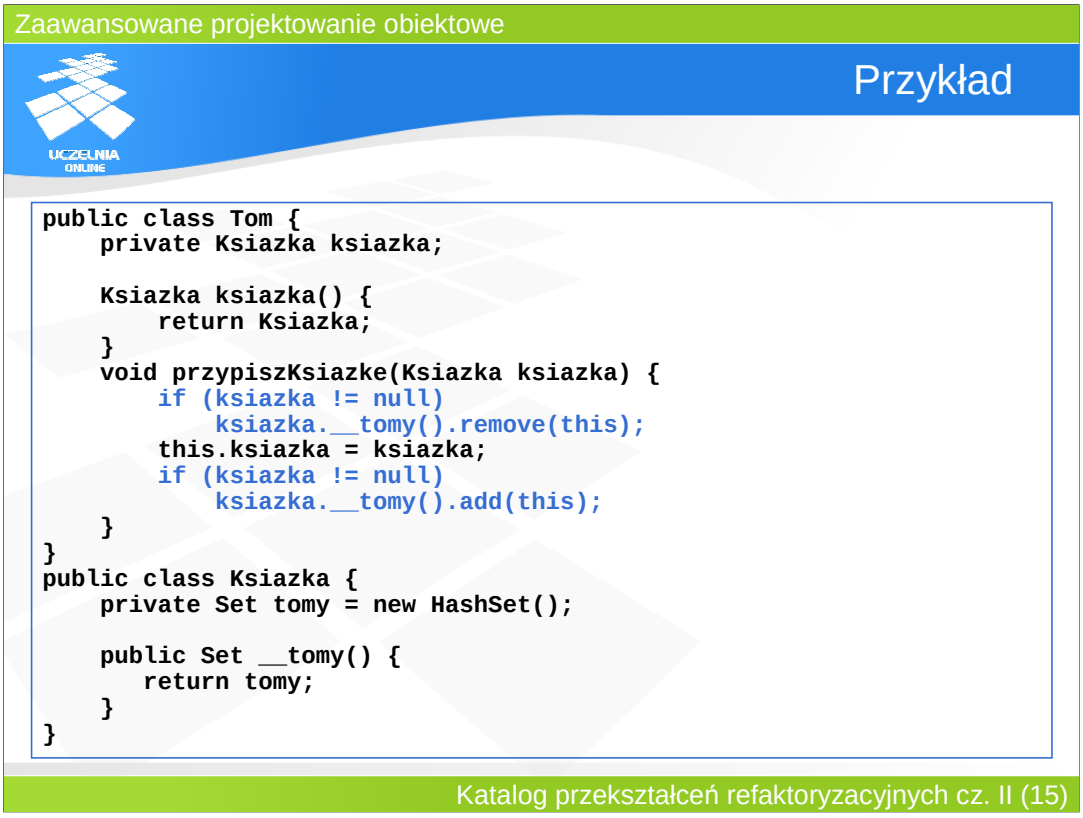


Jako przykład rozpatrzmy relację pomiędzy Książką i Tomami, które wchodzą w jej skład. W obecnej implementacji Tom posiada referencję do Książki, natomiast Książka nie zna Tomów, z których się składa.



Pierwszym krokiem jest stworzenie pola służącego do przechowywania referencji powrotnej w klasie Książka. Ponieważ Książka może składać się z wielu Tomów, dlatego najlepszą strukturą do ich przechowywania jest zbiór (zapewnia on także unikatowość elementów, choć z natury nie zachowuje ich porządku).

Następnie w klasie Książka należy dodać metodę *__tomy()*, pozwalającą na dostęp i modyfikację nowopowstałego pola tomy. Warto zwrócić uwagę na nazwę metody, zaczynającą się od dwóch znaków podkreślenia – wskazuje ona na specjalne przeznaczenie metody, która nie powinna być bezpośrednio wywoływana przez klientów.



Ostatnią operacją jest wprowadzenie zmian do metody *przypiszKsiazke()* w klasie Tom, która w tym przypadku pełni rolę strony kontrolującej relację. Pierwszą instrukcją tej metody jest usunięcie referencji do bieżącego obiektu Tom ze zbioru referencji powrotnych w klasie Książka. Wykonanie tej operacji na początku przypisania Książki zapewnia, że usunięte zostają ewentualne istniejące wcześniej referencje do Tomu. Następnie zmieniana jest referencja do Książki przechowywana w klasie Tom, a ostatnim krokiem jest dodanie referencji do Tomu w klasie Książka.

Teraz przypisanie Książki do klasy Tom powoduje jednoczesne dodanie tego Tomu do zbioru referencji przechowywanych w klasie Książka.

Change Bidirectional Association with Uni

Problem

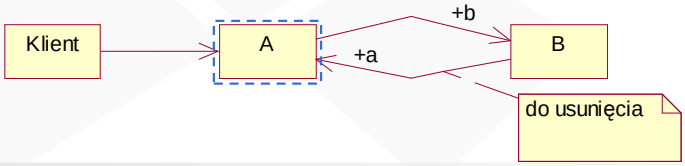
Dwukierunkowa asocjacja wprowadza zbyt silne powiązanie

Cel

Usunięcie jednego z kierunków asocjacji

Mechanika

- określ sposób odwoływania się do obiektu A z obiektu B
- usuń odwołania do metody aktualizującej (set) po stronie obiektu B
- usuń referencję powrotną po stronie obiektu B



M. Fowler, 1999

Przekształcenie to służy do osiągnięcia przeciwnego celu niż poprzednie: zmiany relacji dwukierunkowej na jednokierunkową. Zmiana taka jest wskazana, gdy dwie klasy są zbyt blisko ze sobą związane. Wówczas usunięcie jednego z kierunków asocjacji pozwala "uwolnić" jeden z obiektów i ograniczyć siłę zależności istniejącej między nim a drugim obiektem.

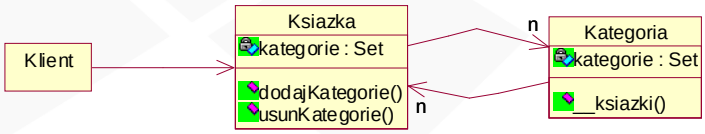
Aby przystąpić do realizacji tego przekształcenia, należy najpierw określić, który z kierunków asocjacji ma pozostać, oraz w jaki sposób można zastąpić usuwany kierunek relacji. Nie można bowiem całkowicie usunąć asocjacji bez wpływu na sposób komunikowania się obiektów, dlatego konieczne jest wprowadzenie rozwiązania zastępczego, które nie będzie wymagało istnienia stałej asocjacji.

Kolejnym krokiem jest usunięcie odwołań do metody aktualizującej referencje powrotne po stronie obiektu kontrolowanego (w tym przypadku B) z metody w obiekcie kontrolującym (czyli A). Po wykonaniu tego kroku obiekt kontrolowany nie będzie posiadał aktualnych referencji do obiektów po stronie kontrolującej, a zatem jest to krok wprowadzający istotną zmianę mogącą mieć wpływ na zachowanie programu.

Po usunięciu odwołania do metody po stronie obiektu B można usunąć zarówno pole reprezentujące referencje powrotne w tej klasie, jak i samą metodę. Ostatnim krokiem jest modyfikacja metod w klasie B, które wymagają referencji do klasy A: może ona np. zostać przekazana jako parametr lub uzyskana za pośrednictwem innego obiektu.

Przykład

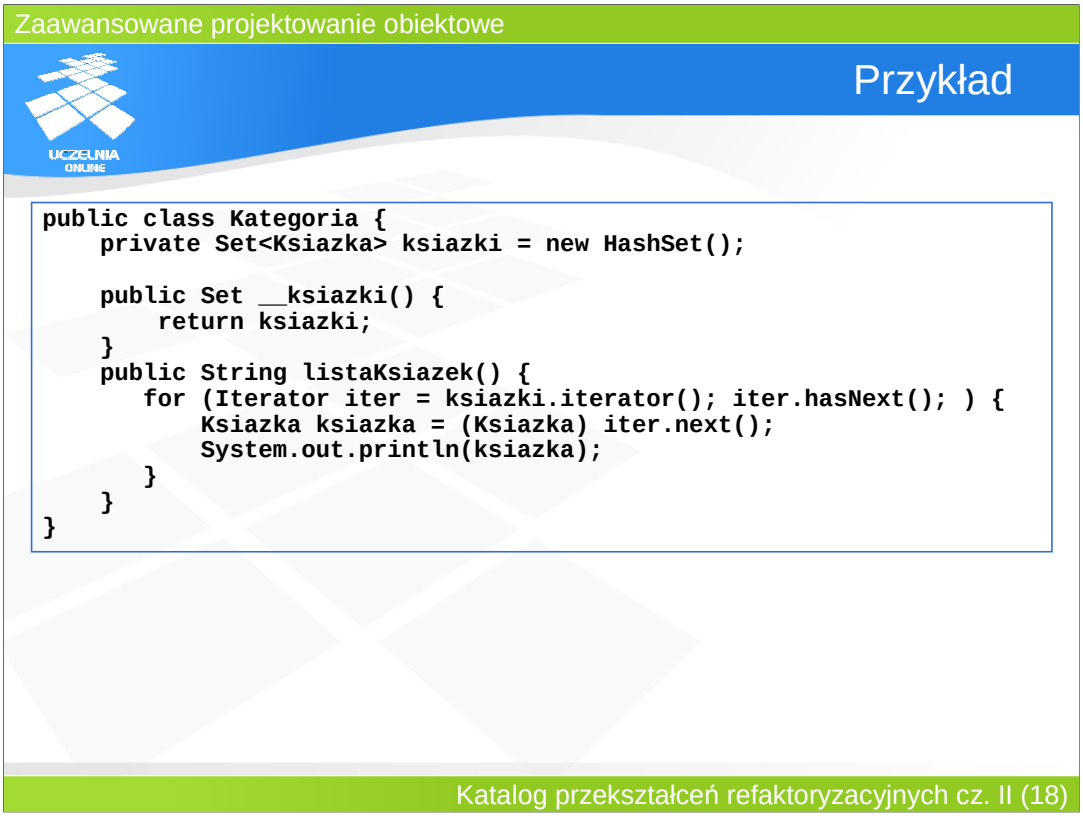
```
public class Książka {
    private Set<Kategoria> kategorie;

    public void dodajKategorie(Kategoria kategoria) {
        if (kategoria != null) kategoria.__ksiazki().remove(this);
        kategorie.remove(kategoria);
        kategorie.add(kategoria);
        if (kategoria != null) kategoria.__ksiazki().add(this);
    }
    public void usunKategorie(Kategoria kategoria) {
        if (kategoria != null) kategoria.__ksiazki().remove(this);
        kategorie.remove(kategoria);
    }
}
```



Przekształcenie zostanie omówione na przykładzie relacji pomiędzy obiektami Książka i Kategoria. Przykład ten różni się od poprzedniego, poza kierunkiem wykonania przekształcenia, także krotnością relacji. Książka może należeć do wielu Kategorii, a każda Kategoria składa się z wielu Książek.

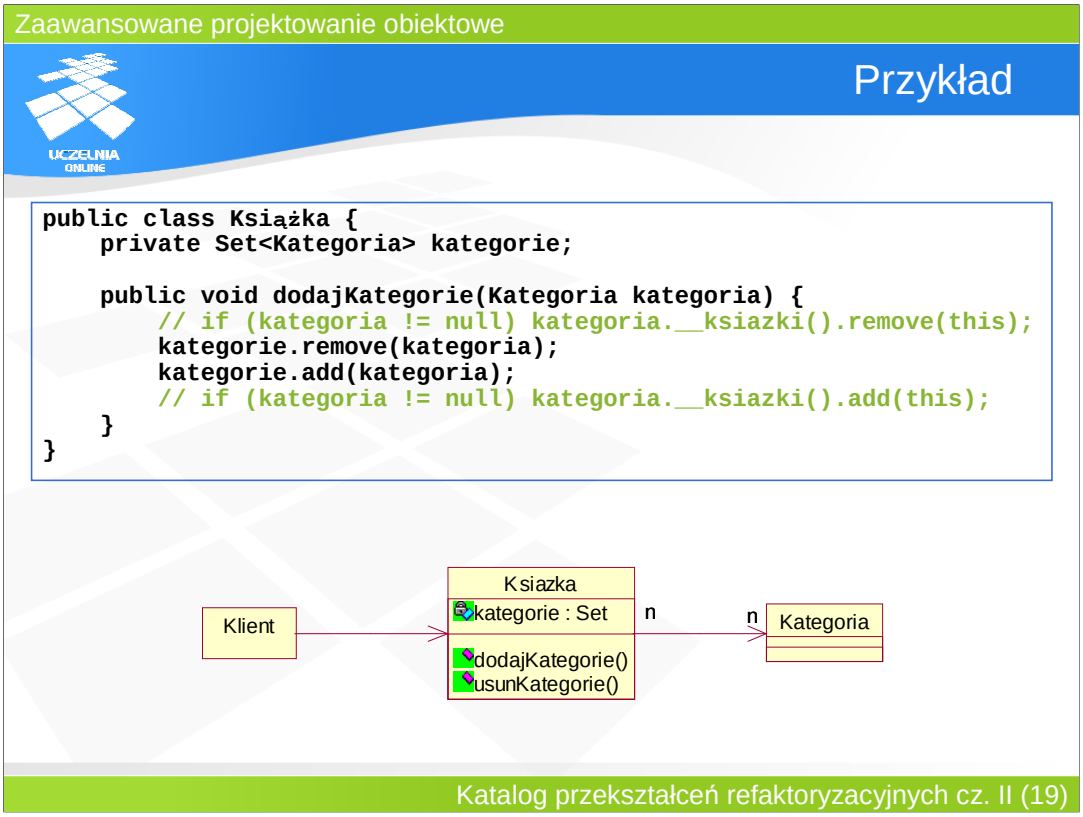
Na początku Książka i Kategoria posiadają referencje do siebie nawzajem. Usunięta ma zostać możliwość nawigacji od Kategorii do Książki.

Slajd ten pokazuje implementację fragmentu klasy Książka, przede wszystkim metod dodającej i usuwającej referencję do Kategorii w klasie Książka. Łatwo zauważyć algorytm dodawania i usuwania referencji do obiektu omówiony na poprzednim przykładzie.

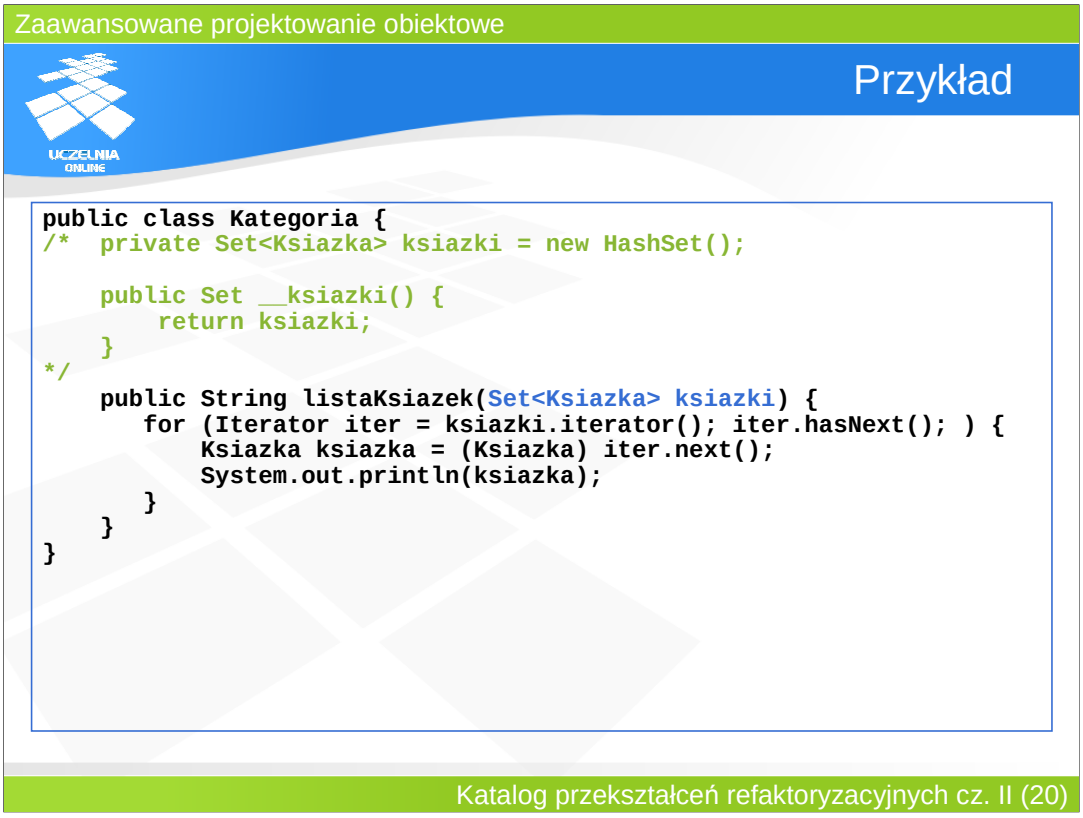


Slajd ten jest kontynuacją opisu stanu początkowego, przed wykonaniem jakiejkolwiek modyfikacji kodu.

Po stronie klasy Kategoria przechowywane są referencje powrotne oraz zdefiniowana jest metoda umożliwiająca modyfikację tych referencji. Ponadto Kategoria posiada metodę *listaKsiążek(),* która korzysta z referencji powrotnej do książek i wyświetla ich listę na ekranie.



Pierwszy krok przekształcenia polega na usunięciu kodu aktualizującego referencje powrotne po stronie klasy Kategoria. Na slajdzie odpowiednie linie kodu zostały umieszczone w komentarzu i zapisane na zielono.



Teraz można usunąć nieużywane referencje powrotne i związany z nimi kod w klasie Książka.

Ostatnim krokiem przekształcenia jest zapewnienie metodom w tej klasie, które dotychczas korzystały z referencji powrotnych, innej metody dostępu do niezbędnych danych. Metoda *listaKsiazek()*, która wyświetlała listę Książek związanych z daną Kategorią, otrzymuje zbiór Książek jako parametr.

W efekcie przekształcenia relacja pomiędzy Książką i Kategorią została zredukowana do asocjacji jednokierunkowej, od Książki do Kategorii.

Replace Inheritance with Delegation

Problem

Podklasa niepotrzebnie dziedziczy pola i metody z nadklasy

Cel

Zastąpienie dziedziczenia jawną delegacją do dawnej nadklasy

Mechanika

- utwórz w podklasie pole typu nadklasy i przypisz mu referencję *this*
- kolejno zmieniaj odwołania do metod nadklasy na odwołania przez delegację
- usuń dziedziczenie pomiędzy nadklasą i podklasą
- wprowadź metody delegujące do wykorzystywanych wcześniej metod dziedziczących z nadklasy
- skompiluj

M. Fowler, 1999

Kolejne przekształcenie pełni bardzo istotną rolę nie tylko jako metoda refaktoryzacji, ale również poznawczą: pokazuje, że w rzeczywistości dziedziczenie i delegacja są sobie równoważne na poziomie fizycznych struktur danych, różnią się jedynie sposobem implementacji relacji. Pamiętamy jednak z pierwszego wykładu, że stosowanie delegacji zamiast dziedziczenia przyczynia się do większej elastyczności systemu i jego otwartości na zmiany.

Przekształcenie to służy do zmiany dziedziczenia w relację delegacji. Klasa, która dotychczas była podklasą, po przekształceniu staje się obiektem delegującym wybrane odwołania do swojej dawnej nadklasy, a wywołania poprzedzone kwalifikatorem nadklasy (w Javie jest to słowo *super*) zostaną zastąpione wyrażoną jawnie referencją.

Pierwszym krokiem przekształcenia jest wprowadzenie do podklasy pola o typie nadklasy i przypisanie mu wartości *this*. Powoduje to utworzenie jawnej referencji do nadklasy; jawnej, ponieważ relacja dziedziczenia zawsze wprowadza inną, niejawną relację pomiędzy podklasą a nadklasą. W rzeczywistości zatem istnieją w tym momencie dwie relacje łączące te same obiekty.

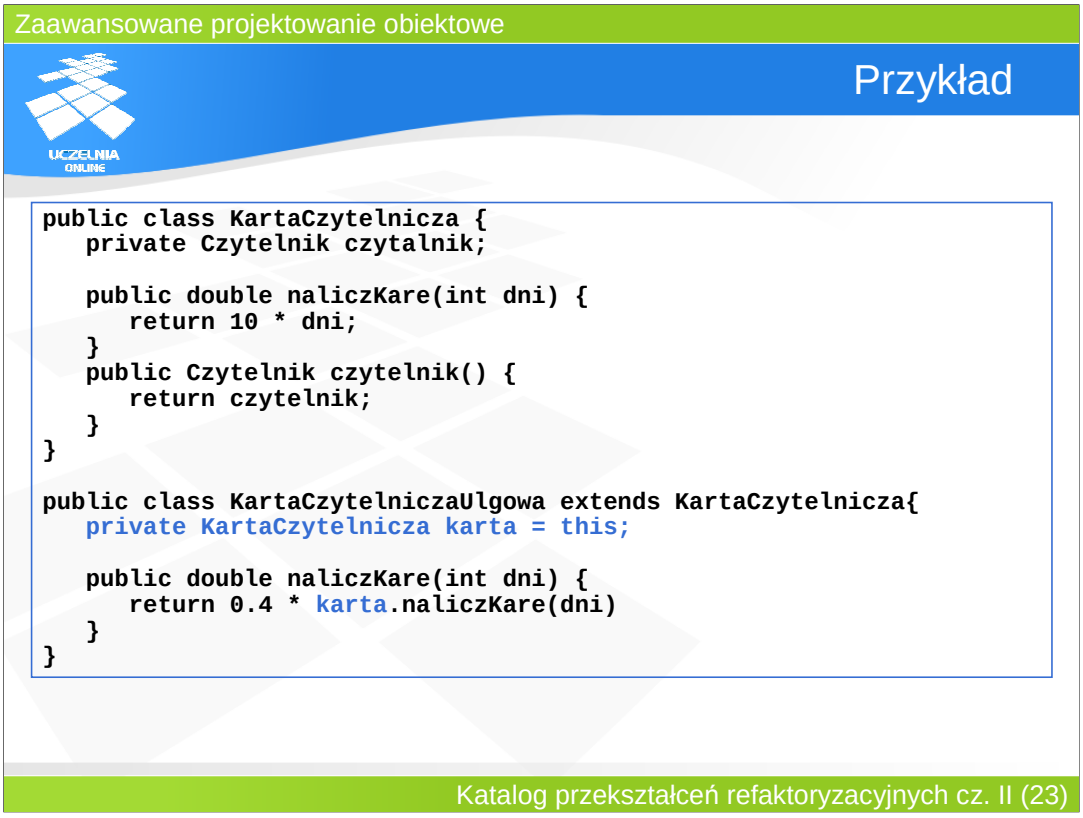
Ta sytuacja pozwala na stopniowe zmienianie odwołań do nadklasy przez referencję niejawną (czyli oznaczaną słowem *this* lub pomijaną) na odwołania przez referencję jawną. W tej fazie zapewniona jest możliwość odwrócenia przekształcenia i powrotu do stanu wyjściowego.

Po aktualizacji wszystkich odwołań można usunąć relację dziedziczenia, co powoduje konieczność wykonania ostatniego kroku przekształcenia: zaimplementowania prostych metod delegujących z "podklasy" do "nadklasy" w miejsce dawnych metod odziedziczonych.

Przykład

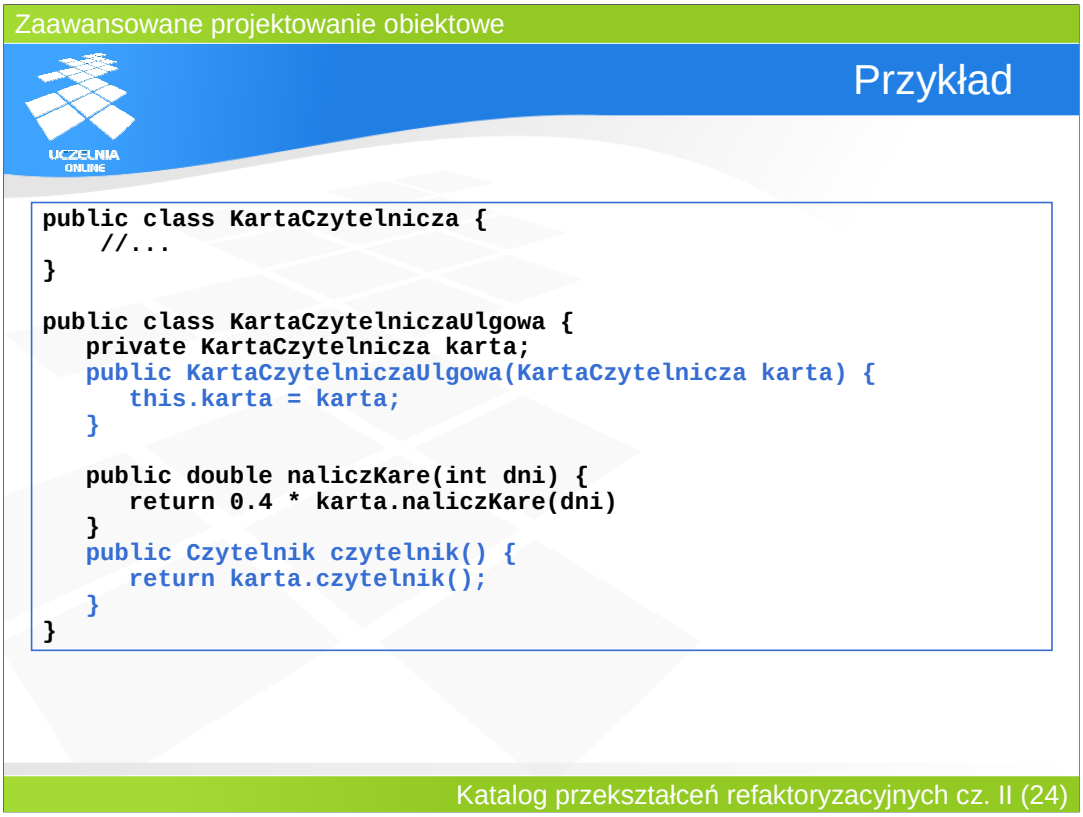
```
public class KartaCzytelnicza {
   private Czytelnik czytelnik;

   public double naliczKare(int dni) {
      return 10 * dni;
   }
   public Czytelnik czytelnik() {
      return czytelnik;
   }
}

public class KartaCzytelniczaUlgowa extends KartaCzytelnicza{

   public double naliczKare(int dni) {
      return 0.4 * super.naliczKare(dni)
   }
}
```


Realizację przekształcenia prześledzimy na kolejnym przykładzie ze świata biblioteki. Klasa KartaCzytelnicza posiada podklasę – KartęCzytelniczą Ulgową, która w stosunku do swojej nadklasy posiada pokrytą metodę *naliczKare()*.



Pierwszy krok przekształcenia to stworzenie w klasie KartaCzytelniczaUlgowa pola typu KartaCzytelnicza i przypisanie mu wartości *this*. Od tego momentu obiekty tych klas są powiązane dwiema równoległymi relacjami.

Zmianie ulega także sposób odwołania w metodzie *naliczKare():* z wywołania metody w nadklasie na wywołanie przez delegację.



Po usunięciu wszystkich powiązań można usunąć deklarację dziedziczenia w klasie KartaCzytelniczaUlgowa. Oczywiście, w tym momencie przypisanie wartości do pola *karta* stanie się niemożliwe, ponieważ zmienna *this* nie odnosi się już do klasy KartaCzytelnicza. Aby poprawnie zainicjować to pole, warto zdefiniować konstruktor, który przyjmuje jako parametr instancję klasy KartaCzytelnicza i przypisze ją do tego pola.

Dokończenie przekształcenia polega na uzupełnieniu w dawnej podklasie brakujących metod, które dotychczas były dziedziczone z nadklasy. Obecnie mają one postać prostych delegacji.

W wyniku przekształcenia relacja dziedziczenia łącząca KartęCzytelniczą i KartęCzytelnicząUlgową została zmieniona w relację delegacji.

Replace Delegation with Inheritance

Problem

W klasie występuje wiele prostych delegacji do jednej klasy

Cel

Zastąpienie delegacji dziedziczeniem po tej klasie

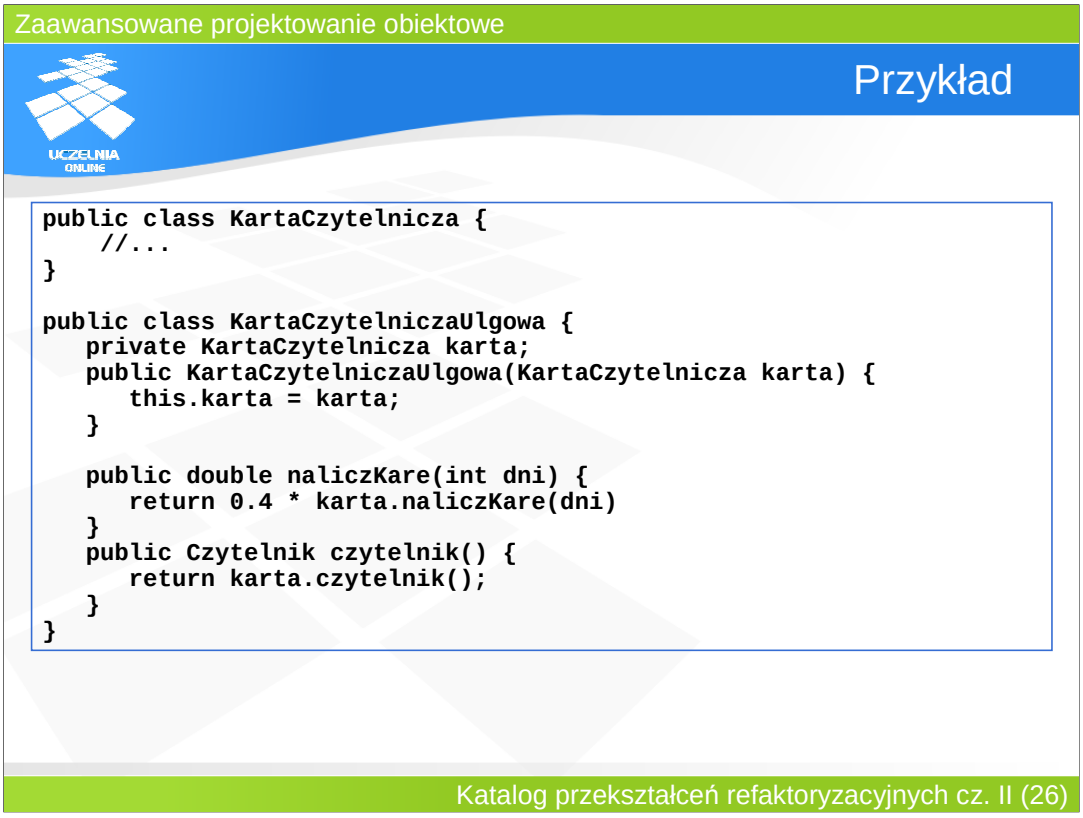
Mechanika

- wprowadź dziedziczenie pomiędzy klasą delegowaną i bieżącą
- przypisz polu delegacji referencję *this*
- kolejno zastąp delegacje wywołaniami metod z nadklasy
- usuń pole delegacji
- skompiluj i przetestuj

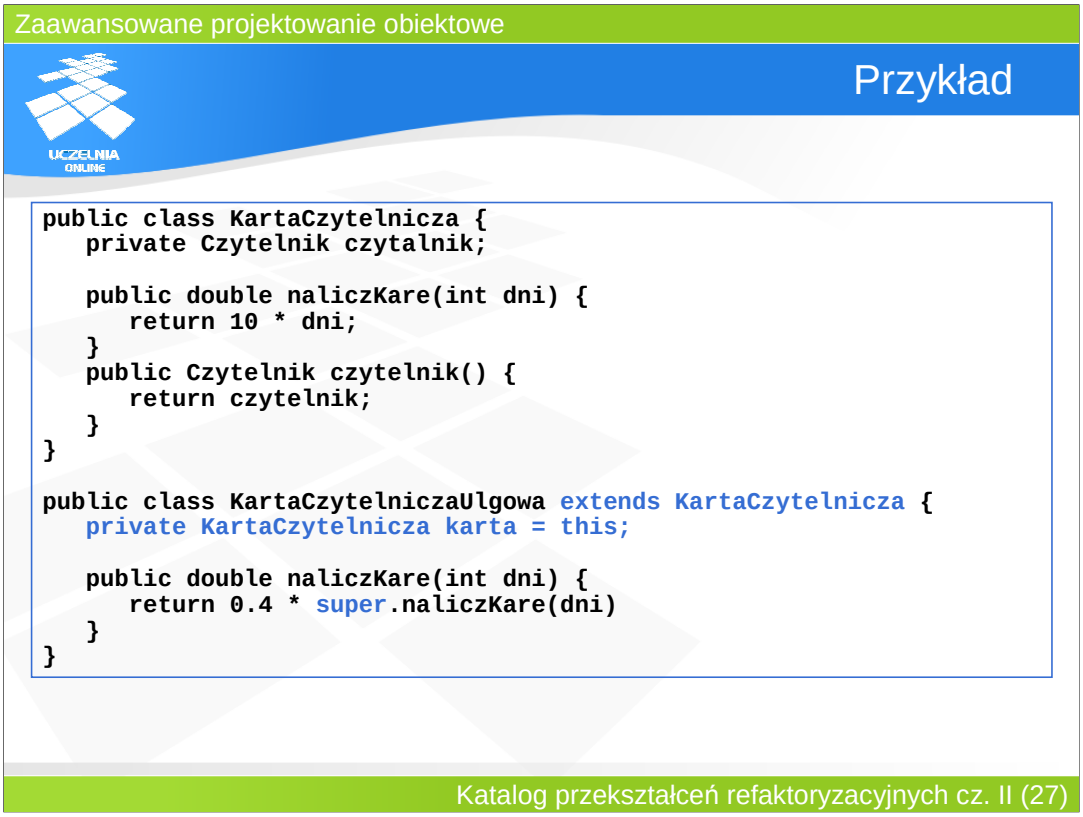
M. Fowler, 1999

Bieżące przekształcenie jest komplementarne do poprzedniego, tzn. zamienia delegację na dziedziczenie. Zmiana taka wymaga lepszego uzasadnienia niż odwrotna transformacja, ponieważ zwykle powoduje usztywnienie schematu klas i utrudnia jego rozszerzanie. Jednak istnieją sytuacje, w których przekształcenie to jest uzasadnione, np. gdy klasa deleguje niemal wszystkie swoje wywołania do innej klasy, i nie zachodzi konieczność zmiany delegacji do innego obiektu w trakcie wykonywania programu.

W zasadzie przekształcenie przebiega w sposób dokładnie odwrotny do przedstawionego na poprzednich slajdach. Rozpoczyna się od wprowadzenia relacji dziedziczenia pomiędzy klasą delegowaną (która staje się nadklasą) a klasą bieżącą (od tego momentu podklasą). Następnie pole przechowujące referencję do delegacji ma wartość zmienianą na *this*. Od tego momentu klasy są związane podwójną relacją, dzięki której można w podklasie stopniowo zmieniać odwołanie do obiektu nadklasy z delegacji na dziedziczenie. Po przeniesieniu wszystkich odwołań można usunąć pole delegacji.

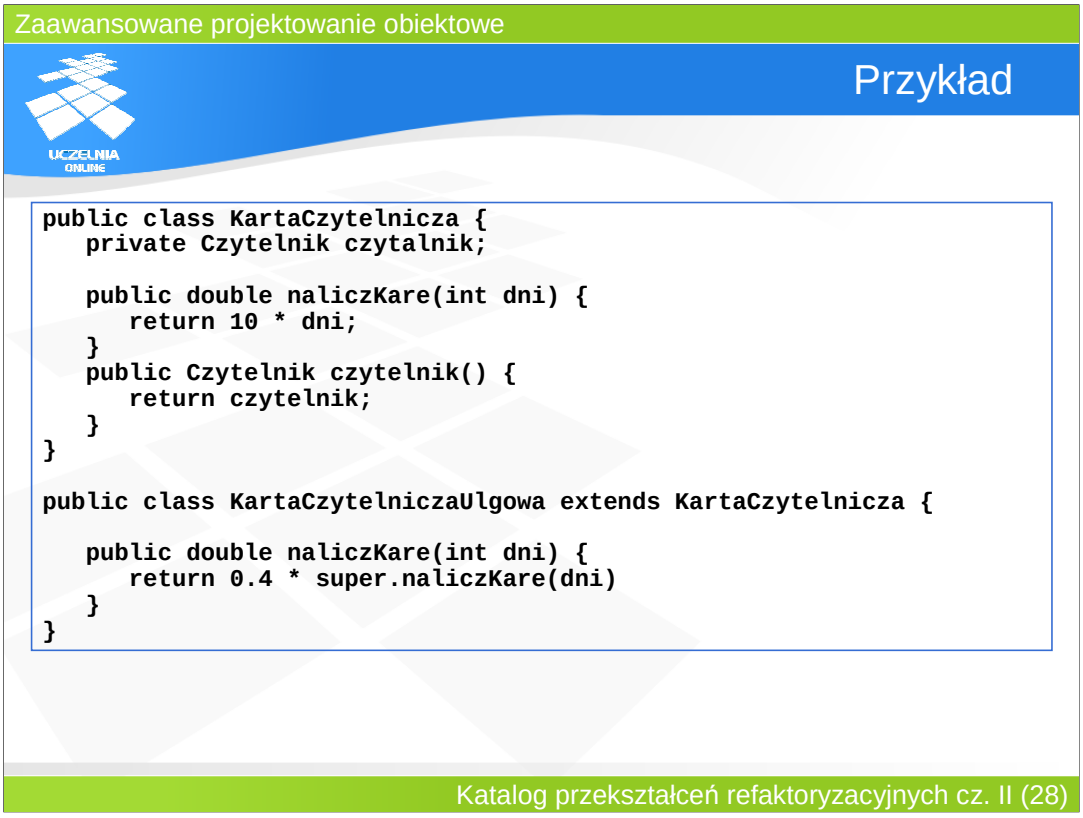


Przykład zaczyna się od stanu, w którym zakończyło się przekształcenie odwrotne: klasa KartaCzytelniczaUlgowa posiada referencję do KartyCzytelniczej i do niej deleguje niektóre swoje metody, np. *naliczKare()* i *czytelnik()*.



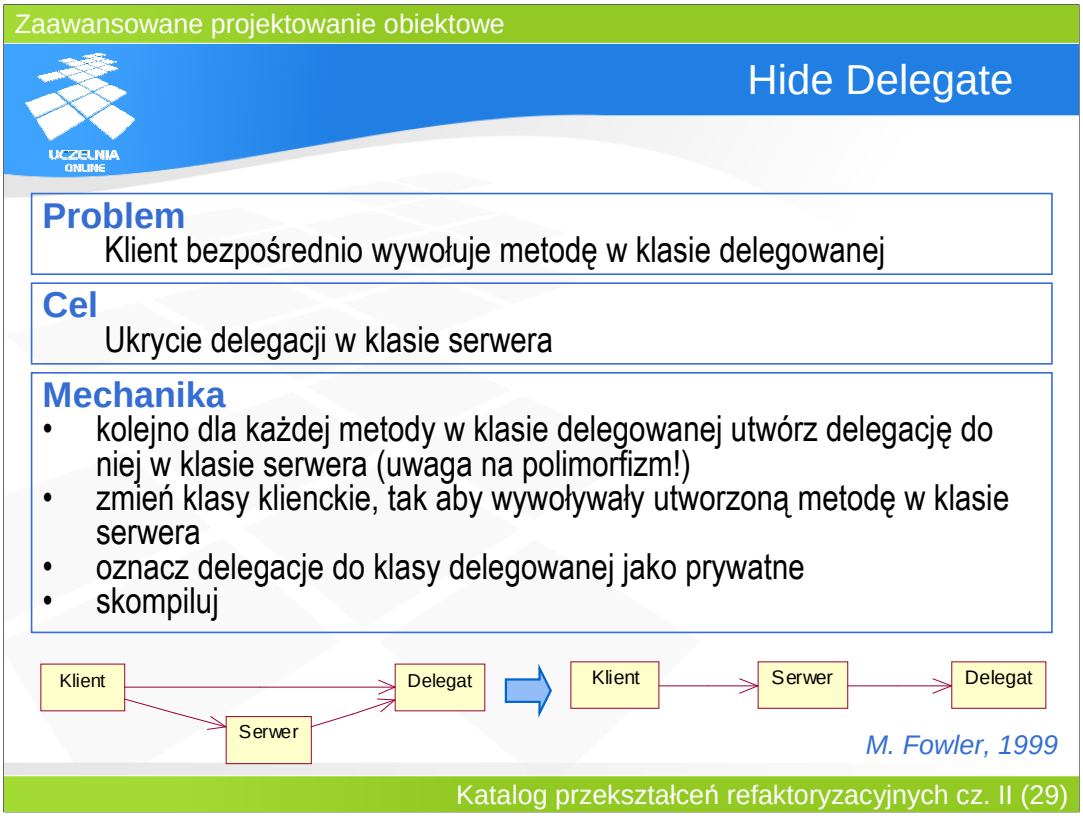
Pierwszym krokiem jest zadeklarowanie w klasie KartaCzytelniczaUlgowa dziedziczenia po klasie KartaCzytelnicza oraz przypisanie do pola przechowującego referencję do KartyCzytelniczej wartości *this*.

Następnie w kolejnych metodach należy zmienić odwołania poprzez referencję na odwołania do nadklasy.



Po zakończeniu przenoszenia odwołań można usunąć elementy związane z delegacją: pole z referencją do KartyCzytelniczej i konstruktor inicjujący to pole.

W efekcie klasa KartaCzytelniczaUlgowa stała się podklasą KartyCzytelniczej.



Celem tego przekształcenia jest poprawa hermetyzacji systemu poprzez ukrycie faktu delegowania żądań przez serwer. W układzie składającym się z trzech klas: klienta, serwera i delegata oznacza to, że klient kontaktuje się jedynie z serwerem, pozostając nieświadomym istnienia delegata, któremu serwer zleca obsługę żądania. Dzięki temu nie występuje powiązanie pomiędzy klientem a delegatem. Pozwala to na elastyczną zmianę delegata w trakcie wykonywania programu.

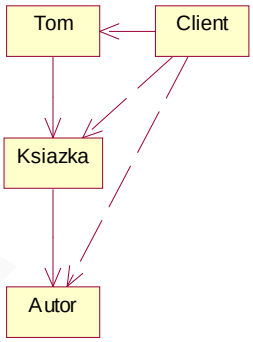
Przekształcenie polega na utworzeniu w klasie serwera prostych delegacji przekierowujących obsługę żądania klienta do delegata. Ponieważ przekształcenie to wiąże się z tworzeniem nowych metod, należy koniecznie upewnić się, że ich powstanie nie zaburza polimorficznego pokrywania w hierarchii dziedziczenia. W takim wypadku przekształcenie to nie może być skutecznie i poprawnie zastosowane.

Następnym krokiem jest modyfikacja klas klienckich, tak aby wywoływały żądania jedynie w klasie serwera, a nie bezpośrednio na delegacie. Po zakończeniu tego kroku można ukryć fakt istnienia delegata poprzez uczynienie prywatną referencji do niego w klasie serwera.

Przykład

```
public class Tom {
   private Ksiazka ksiazka;
   private String numer;

   public Ksiazka ksiazka() {
      return ksiazka;
   }
}

public class Ksiazka {
   private String tytul;
   private Autor autor;

   public Ksiazka(Autor autor) {
      this.autor = autor;
   }
   public Autor autor() {
      return autor;
   }
}
```

```
public class Autor {
   private String nazwisko;
   private Date dataUrodzenia;

}
```



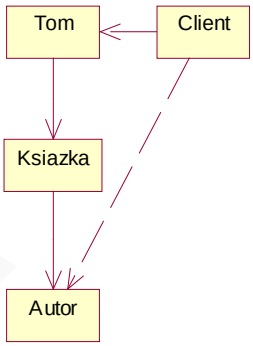
Przekształcenie to prześledzimy na przykładzie klas Tom, Książka i Autor. Klasa Tom posiada referencję do klasy Książka i publiczną metodę *książka(),* która umożliwia dostęp do tej referencji. Podobną strukturę ma klasa Książka: zawiera referencję do klasy Autor i posiada metodę *autor(),* która zwraca tę referencję. Ostatnia w łańcuchu delegacji klasa Autor przechowuje nazwisko i datę urodzenia Autora Książki. Zatem aby z poziomu Tomu lub Książki otrzymać nazwisko Autora, należy stworzyć łańcuch wywołań: *tom.ksiazka().autor().nazwisko*. Łańcuch taki narusza zasady odwołań do obiektów sformułowane przez prawo Demeter (zob. wykład dotyczący metryk obiektowych), ponieważ wymaga od wywołującego cały łańcuch klienta znajomości całego systemu.

Diagram przedstawia relacje asocjacji oraz zależności występujące pomiędzy klasami.

Przykład

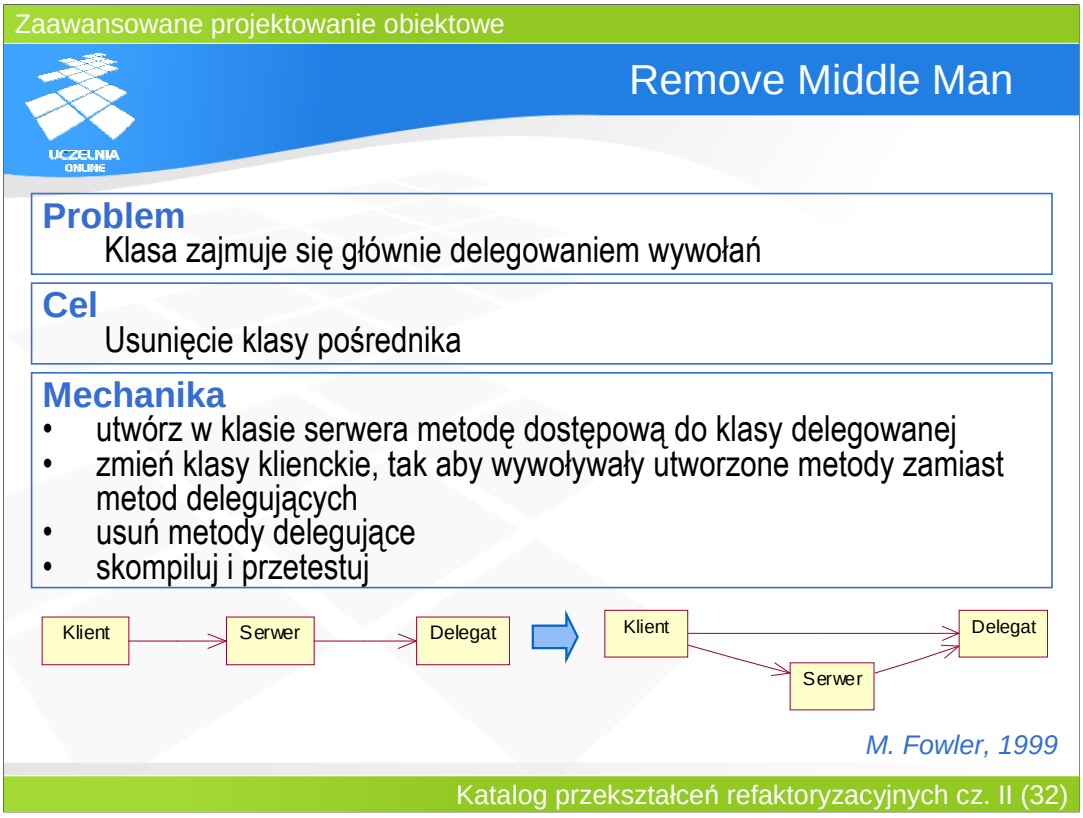
```
public class Tom {
   private Ksiazka ksiazka;
   private String numer;

   private Ksiazka ksiazka() {
      return ksiazka;
   }
   public Autor autor() {
      return ksiazka().autor();
   }
}
public class Ksiazka {
   private String tytul;
   private Autor autor;

   public Ksiazka(Autor autor) {
      this.autor = autor;
   }
   private Autor autor() {
      return autor;
   }
}
```

```
public class Autor {
   private String nazwisko;
   private Date dataUrodzenia;

}
```



W celu ukrycia klasy delegata (w tym przypadku klasy Książka) wprowadzono metodę *autor(),* będącą prostą delegacją do tej klasy, natomiast metodę *książka(),* która dotychczas udostępniała delegata – uczyniono prywatną. Dzięki temu przekształceniu informacja o istnieniu klasy Książka została hermetycznie ukryta w klasie Tom i w pewien sposób pominięta w wywołaniach mających na celu dotarcie do klasy Autor.

Oczywiście, można to przekształcenie kontynuować, usuwając także delegację do klasy Autor i tworząc w klasie Tom proste delegacje zwracające wyniki wykonania metod klasy Autor, jednak wprowadzałoby to zbyt wiele niejasnych delegacji.



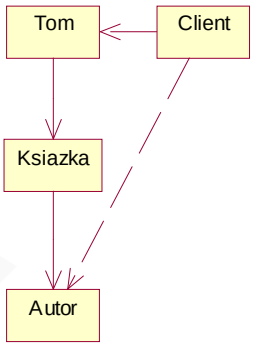
Przekształcenie Remove Middle Man jest przeciwieństwem poprzedniej refaktoryzacji. Jego celem jest usunięcie prostych delegacji z klasy serwera i publiczne udostępnienie delegata. Przekształcenie to ma sens, jeżeli klasa serwera jest prostym tłumaczem protokołów obiektowych i nie dodaje żadnej nowej funkcjonalności w porównaniu do klasy delegata (a jednocześnie nie zachodzą inne przesłanki, np. konieczność dostosowania typów – zob. wzorzec Adapter).

Jego mechanika polega na odwróceniu mechaniki przekształcenia Hide Delegate. Pierwszym krokiem jest utworzenie (lub upublicznienie, jeżeli istniała wcześniej) metody dostępowej do klasy-delegata. Następnie należy zmienić wszystkie klasy klienckie, tak aby wywoływały metody bezpośrednio w klasie delegata, uzyskując do niego referencję poprzez utworzoną w poprzednim kroku metodę dostępową. Po zakończeniu tej czynności metody delegujące z klasy serwera mogą zostać usunięte. Jeżeli serwer po przekształceniu nie posiada już żadnej lub szczątkową funkcjonalność, można go usunąć, stosując np. przekształcenie Inline Class.

Przykład

```
public class Tom {
   private Ksiazka ksiazka;
   private String numer;

   private Ksiazka ksiazka() {
      return ksiazka;
   }
   public Autor autor() {
      return ksiazka().autor();
   }
}
public class Ksiazka {
   private String tytul;
   private Osoba autor;

   public Ksiazka(Osoba autor) {
      this.autor = autor;
   }
   private Autor autor() {
      return autor;
   }
}
```

```
public class Autor {
   private String nazwisko;
   private Date dataUrodzenia;

}
```

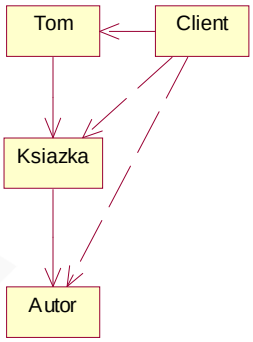


Przykładem jest ten sam zestaw klas, na którym omawiane było przekształcenie Hide Delegate. Stan wyjściowy polega na obecności w klasie Tom metody publicznej *autor(),* która udostępnia obiekt klasy Autor bez pośrednictwa obiektu klasy Książka. Zatem klient, chcąc uzyskać nazwisko Autora Książki, której Tom trzyma w ręku, wywołuje *tom.autor().nazwisko*.

Przykład

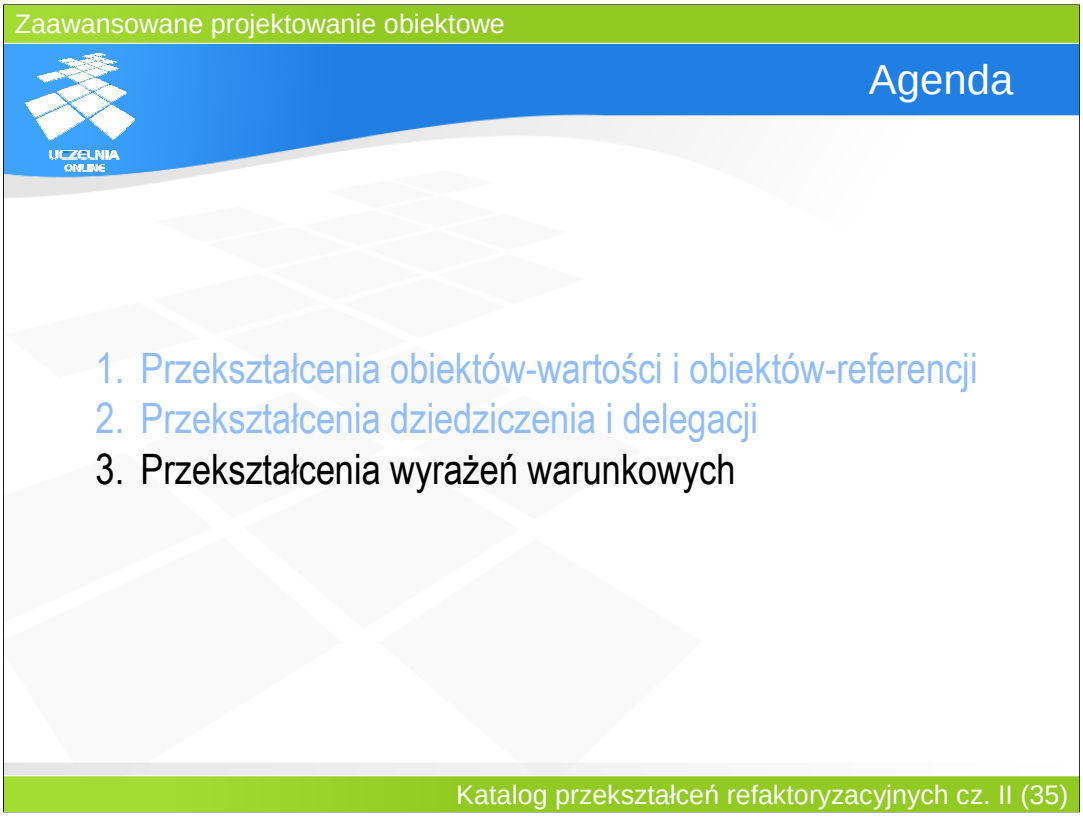
```
public class Tom {
   private Ksiazka ksiazka;
   private String numer;

   public Ksiazka ksiazka() {
      return ksiazka;
   }
   // public Autor autor() {
   //  return ksiazka().autor();
   // }
}
public class Ksiazka {
   private String tytul;
   private Osoba autor;

   public Ksiazka(Osoba autor) {
      this.autor = autor;
   }
   public Autor autor() {
      return autor;
   }
}
```

```
public class Autor {
   private String nazwisko;
   private Date dataUrodzenia;

}
```

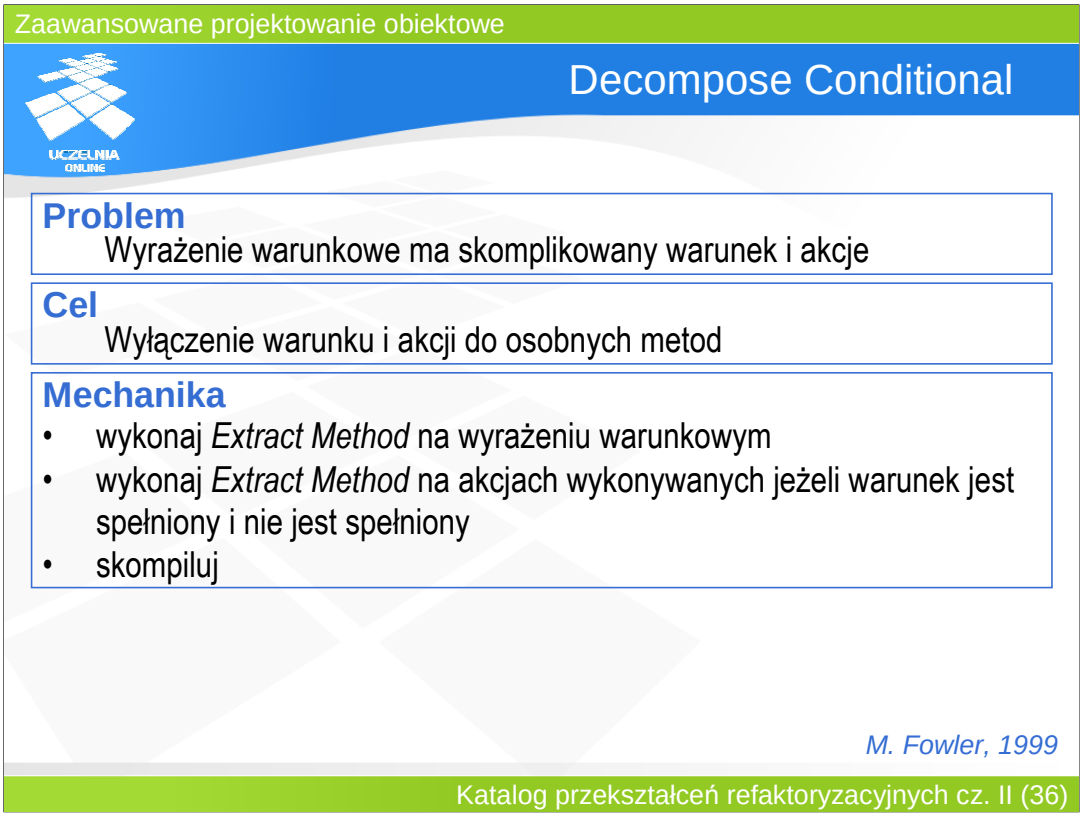


Drugim krokiem przekształcenia jest usunięcie prostej delegacji do klasy Autor, a w zamian udostępnienie metody *książka(),* która umożliwia dostęp do pośredniczącej klasy serwera.

W efekcie przekształcenia dostęp do obiektu Autor jest możliwy jedynie poprzez obiekt klasy Książka, natomiast metody delegujące, umożliwiające dostęp "na skróty", zostają usunięte.



Trzecia część wykładu jest poświęcona przekształceniom wyrażeń warunkowych, które w znacznym stopniu odpowiadają za problem nadmiernej złożoności metod oraz nieprawidłowy dobór relacji pomiędzy obiektami.



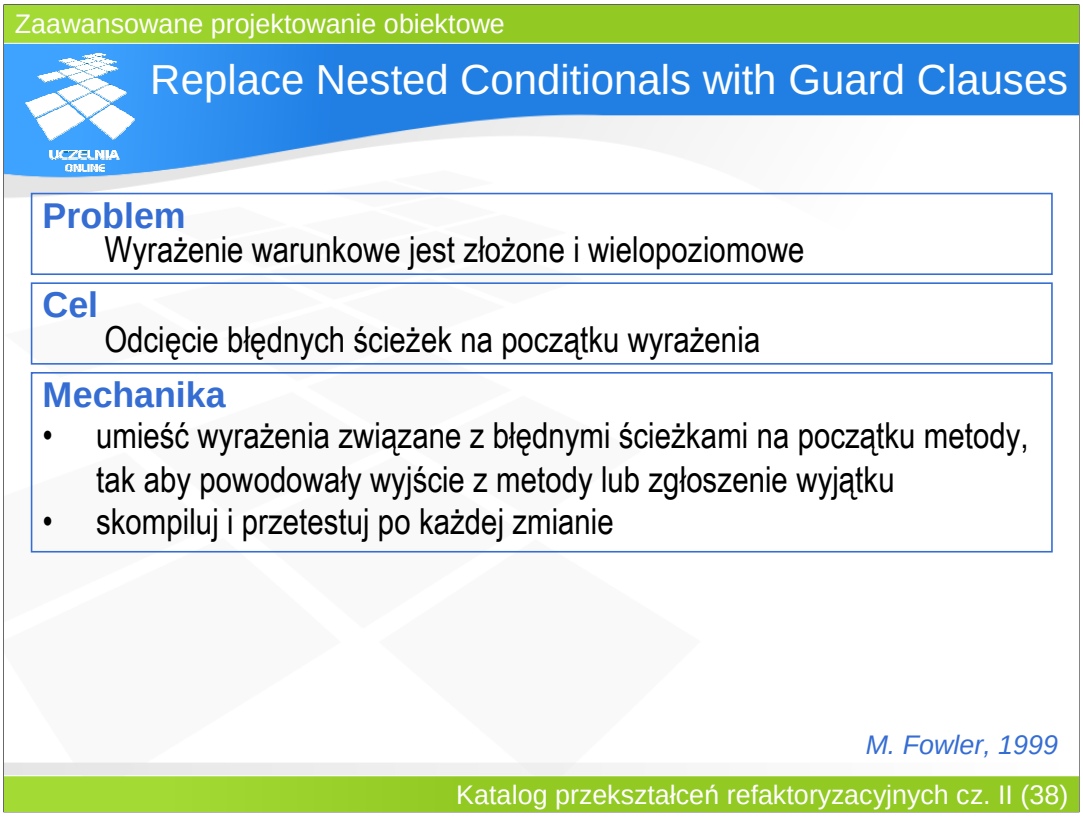
Jest to najprostsze z tej grupy przekształceń, zwiększające przede wszystkim czytelność samego wyrażenia warunkowego. Polega ono na wyłączeniu samego warunku, jak i poszczególnych akcji, do nowych metod. Podobnie mechanika przebiega w przypadku instrukcji wyboru (*switch*).

Przykład

```
if (dzien != 'pn' && dzien != 'wt' && dzien != 'sr'
    && dzien != 'cz' && dzien != 'pt') {
    oplata = 10 + oplataZwykla(liczbaOsob) * 2;
} else {
    oplata = 5 + oplataZwykla(liczbaOsob);
}
```

```
private double dzienWeekendowy(String dzien) {
    return dzien != 'pn' && dzien != 'wt' && dzien != 'sr'
        && dzien != 'cz' && dzien != 'pt'
}
private double oplataDzienWeekendowy(int liczbaOsob) {
    return 10 + oplataZwykla(liczbaOsob) * 2;
}
private double oplataDzienRoboczy(int liczbaOsob) {
    return 5 + oplataZwykla(liczbaOsob);
}
```


Slajd ten przedstawia przykład wyłączania fragmentów wyrażenia warunkowego. W zależności od dnia tygodnia zmienna *oplata* przyjmuje wartości opisane jednym z dwóch wzorów. Przekształcenie polega na utworzeniu trzech prywatnych metod: *dzienWeekendowy(),* sprawdzającej warunek, oraz *oplataDzienWeekendowy()* i *oplataDzienRoboczy(),* obliczających wartości zmiennej *oplata* w zależności od stosowanego algorytmu.

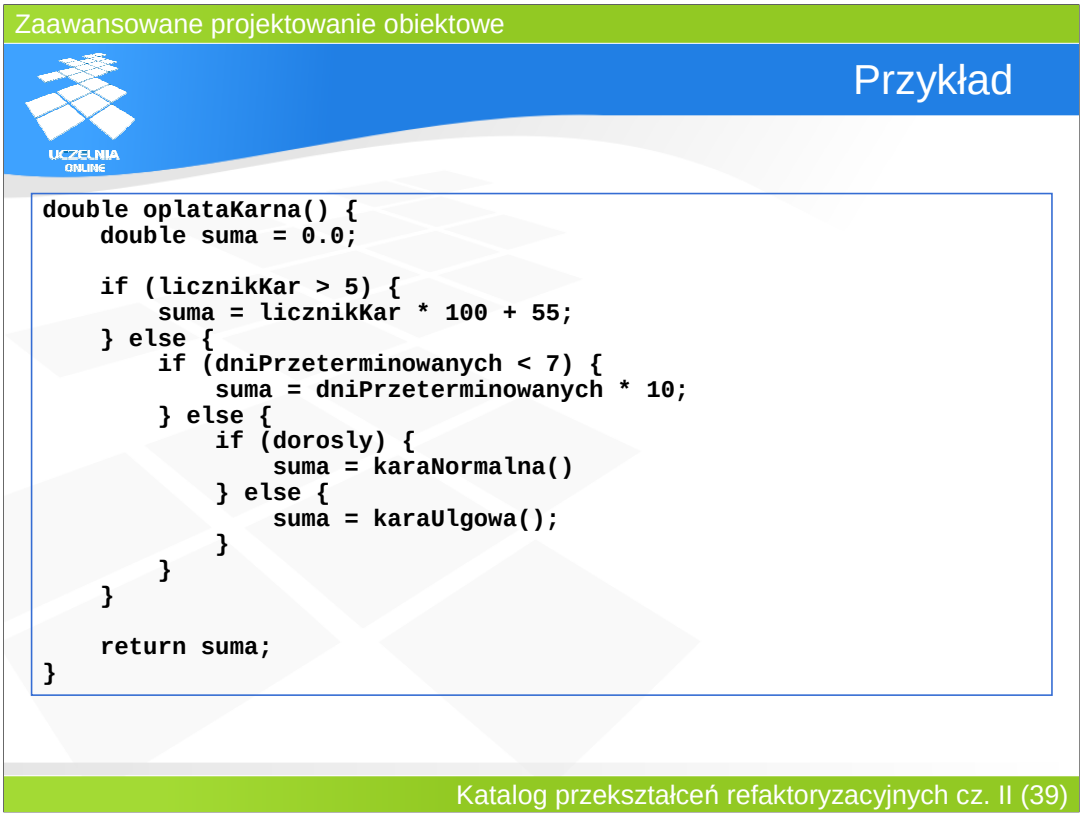


Zagnieżdżone wyrażenia warunkowe w znacznym stopniu komplikują strukturę kodu oraz zmniejszają jego czytelność. Szczególnie negatywnie przyczynia się do tego typowo strukturalny sposób zapisu, w którym metoda (lub blok kodu) może posiadać tylko jeden punkt wyjścia. Konieczne jest wówczas stosowanie złożonych warunków, które weryfikują poprawność poszczególnych ścieżek sterowania, jednak nie urywają ich, nawet gdy okazują się one błędne

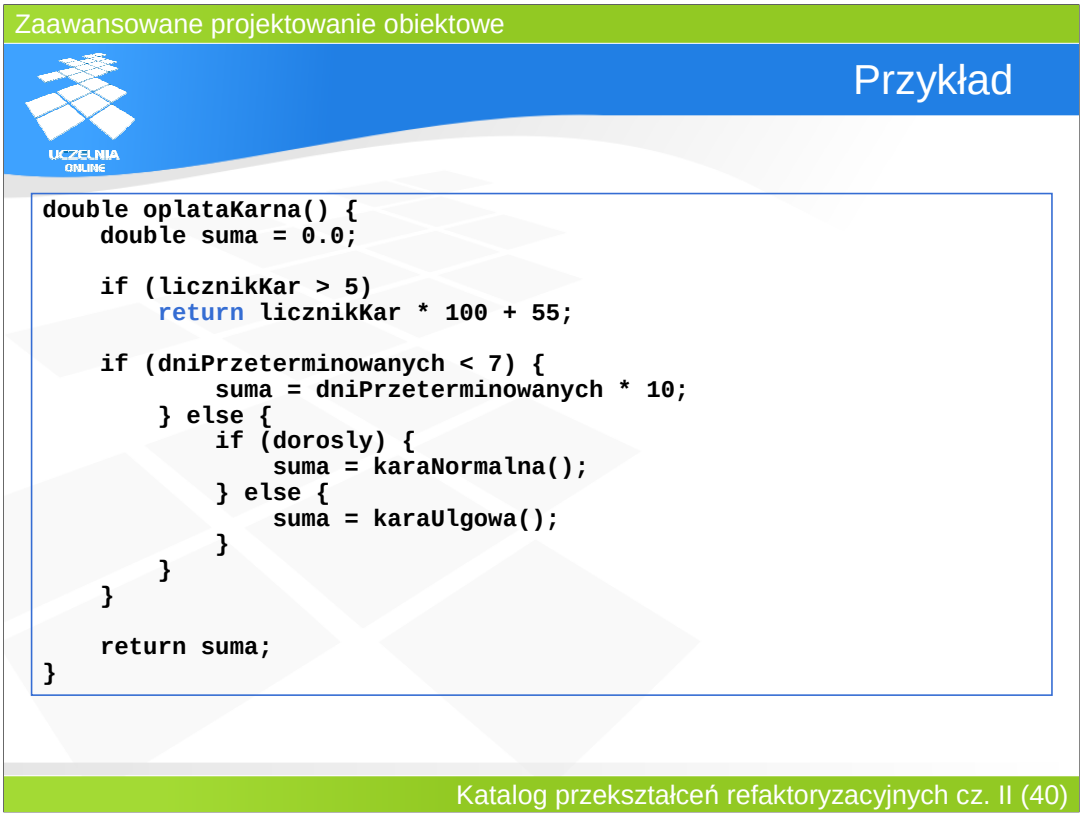
Celem tego przekształcenia jest przesunięcie warunków związanych z błędnymi ścieżkami na początek metody, tak aby można było je odciąć na początku jej wykonywania.

Przekształcenie w zasadzie składa się z jednego kroku: przesunięcia wyrażeń warunkowych weryfikujących np. poprawność parametrów na początek metody, i w przypadku stwierdzenia błędu spowodowania opuszczenia metody (instrukcja *return*) lub bieżącego fragmentu instrukcji wyboru (instrukcje *break* lub *continue*).

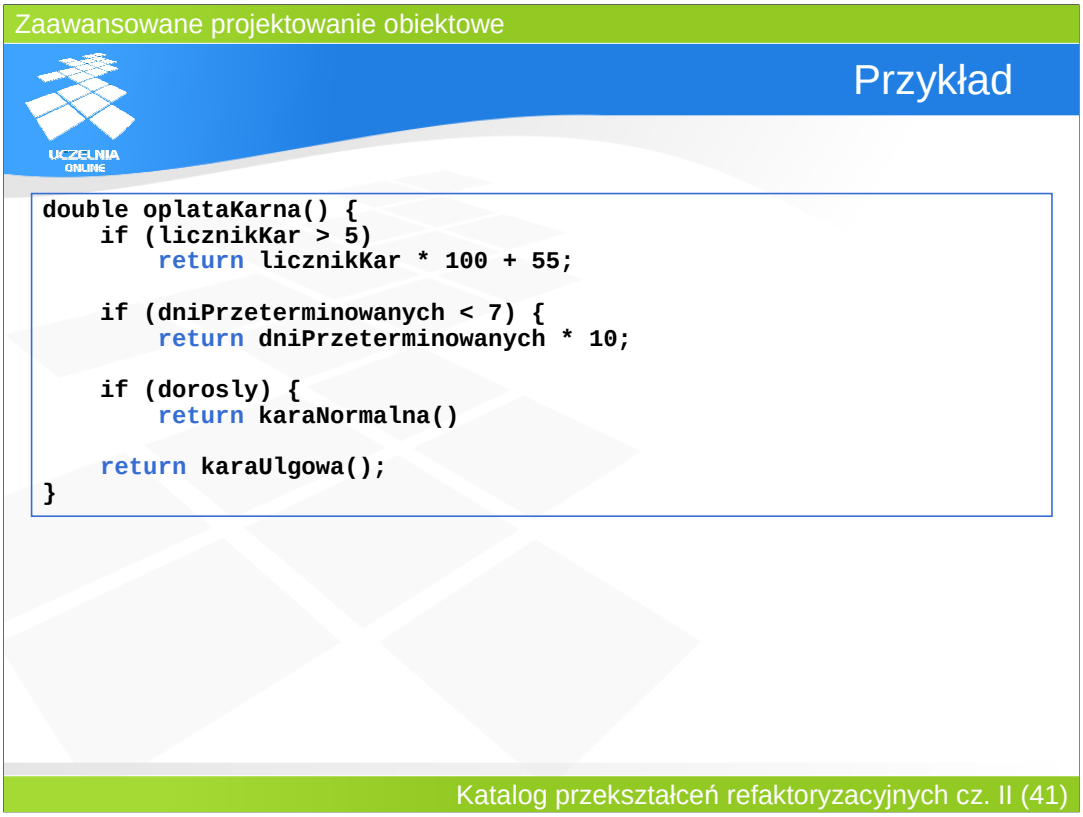
Należy jednak pamiętać, że przekształcenie dotyczy jedynie takich warunków, których spełnienie faktycznie odcina dalsze przetwarzanie, tzn. wartości zmiennych ustalone w tym kroku nie ulegają zmianie do końca metody.



Przykład przedstawia celowo skomplikowaną metodę naliczania opłat karnych za przedłużone wypożyczenie książki. Procedura oceny wysokości kary jest wielopoziomowa i zależy od wyników ewaluacji różnych warunków.



W pierwszym kroku modyfikacji ulega gałąź *else* pierwszej instrukcji warunkowej. Niespełnienie warunku powoduje zwrócenie przez metodę obliczonej wartości, a więc przerwanie dalszego przetwarzania. Pozostałe warunki pozostają niezmienione



Końcowym efektem przekształcenia jest taka postać metody *oplataKarna()*, w której zawiera ona trzy warunki, z których każdy powoduje opuszczenie metody z obliczoną wartością. Jeżeli żaden z warunków nie jest spełniony, funkcja przyjmuje wartość domyślną – wynik funkcji *karaUlgowa().* W przypadku tej metody możliwe było także usunięcie zmiennej lokalnej *suma*, ponieważ jej wartość nie musi być już przechowywana wewnątrz metody.

W wyniku przekształcenia uproszczone zatem zostało złożone wyrażenie warunkowe: w jego miejsce pojawiło się kilka prostych warunków, których spełnienie powoduje przerwanie wykonywania metody. Metoda po przekształceniu posiada zatem kilka punktów wyjścia.

Consolidate Conditional Expressions

Problem

Kilka wyrażeń warunkowych powoduje wykonanie tej samej czynności

Cel

Połączenie wyrażeń warunkowych za pomocą operatorów logicznych

Mechanika

- sprawdź, czy obliczenie wyrażeń warunkowych nie ma efektów ubocznych
- połącz wyrażenia warunkowe korzystając z operatorów logicznych
- skompiluj i przetestuj

M. Fowler, 1999

Prostym rozwiązaniem nieco innego problemu – wielokrotnych wyrażeń warunkowych – jest ich konsolidacja. Jeżeli kilka występujących po sobie wyrażeń warunkowych *if* powoduje wykonanie tej samej czynności lub ma ten sam efekt, wówczas można je połączyć w jedno złożone wyrażenie za pomocą operatorów logicznych.

Poprawność tego przekształcenia zależy od braku efektów ubocznych poszczególnych wyrażeń, ponieważ konsolidacja warunków powoduje, że ewaluowane są tylko te z nich, które są niezbędne do obliczenia końcowej wartości całego wyrażenia.

Zadanie

```
public boolean zarezerwujKsiazke() {
   liczbaRezerwacji++;
   if (liczbaRezerwacji()) > 3
      return false;
   return true;
}

public boolean anulujRezerwacje() {
   liczbaRezerwacji--;
   return true;
}

int ksiazkiWypozyczone = liczbaWypozyczen();
boolean dalszeWypozyczanie = true;

if (ksiazkiWypozyczone > 4 && zarezerwujKsiazke())
    dalszeWypozyczanie = false;
if (ksiazkiWypozyczone > 5 && anulujRezerwacje())
    dalszeWypozyczanie = false;
if (ksiazkiWypozyczone > 6)
    dalszeWypozyczanie = false;
```


Zadanie polega na wykonaniu przekształcenia konsolidującego wyrażenia warunkowe. Podczas jego realizacji należy zwrócić baczną uwagę na potencjalne efekty uboczne obliczanych wyrażeń.

Remove Control Flag

Problem

Zmienna steruje kilkoma wyrażeniami warunkowymi

Cel

Zastąpienie zmiennej instrukcjami przerywającymi wykonywanie

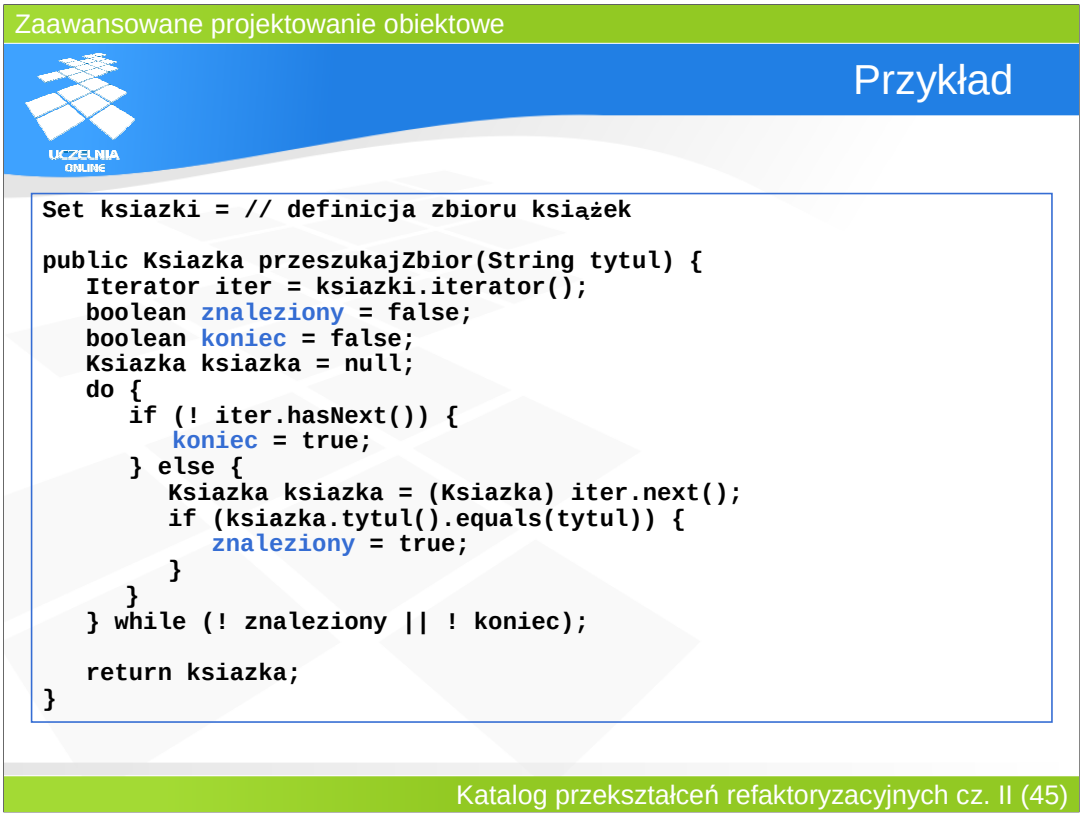
Mechanika

- znajdź zmienne, których wartość decyduje o przerwaniu wykonania bloku kodu
- zastąp przypisania do tych zmiennych instrukcjami *break* lub *continue*
- skompiluj i przetestuj

M. Fowler, 1999

Ostatnie przekształcenie z grupy poświęconej wyrażeniom warunkowym dotyczy problemu flag sterujących – zmiennych, których wartość jest modyfikowana w trakcie wykonywania metody, i która decyduje o przepływie sterowania. Zmienne takie narzucają często proceduralny sposób programowania, w którym metoda posiada tylko jeden punkt wyjścia. Lepszym rozwiązaniem jest wykorzystanie instrukcji sterujących *return*, *break* i *continue*, które poprawiają czytelność kodu.

Przekształcenie rozpoczyna się od identyfikacji flag sterujących. Następnie przypisania do tych zmiennych są zastępowane instrukcjami sterującymi, które w ten sposób decydują o wykonaniu lub opuszczeniu bieżącego bloku kodu.



Przykład przekształcenia zostanie omówiony na metodzie *przeszukajZbior()*, która wyszukuje obiekt klasy Książka na podstawie jej tytułu. W programie występują dwie flagi: *koniec* i *znaleziony*. Pierwsza decyduje o zakończeniu przeszukiwania zbioru ze względu na przejrzenie wszystkich jego elementów, a druga informuje o znalezieniu książki.

Przykład

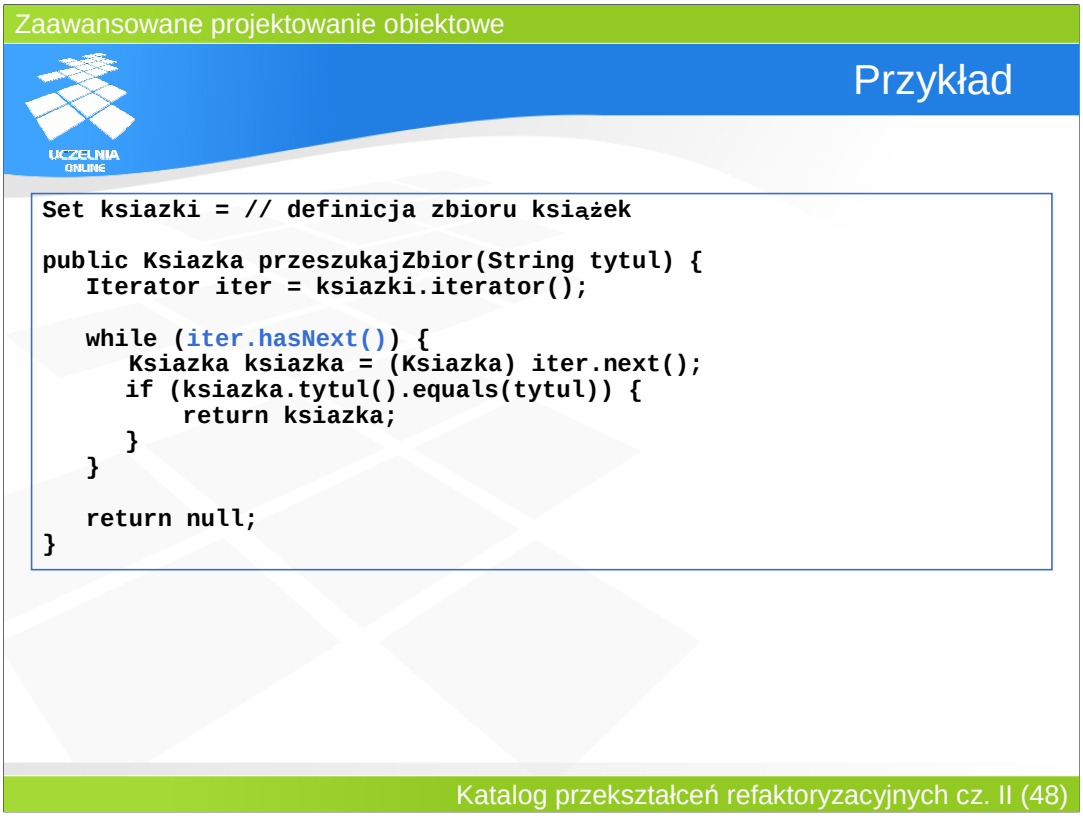
```
Set ksiazki = // definicja zbioru książek

public Ksiazka przeszukajZbior(String tytul) {
   Iterator iter = ksiazki.iterator();
   boolean znaleziony = false;
   boolean koniec = false;
   Ksiazka ksiazka = null;
   do {
      if (! iter.hasNext()) {
         break;
      } else {
         Ksiazka ksiazka = (Ksiazka) iter.next();
         if (ksiazka.tytul().equals(tytul)) {
            return ksiazka;
         }
      }
   } while (! znaleziony || ! koniec);

   return ksiazka;
}
```


Pierwszy krok polega na zastąpieniu przypisania do tych zmiennych instrukcjami *break* i *return*. Instrukcja *return* jest użyta, gdy Książka zostanie znaleziona – zatem może zostać zwrócona przez metodę, a dalsze przetwarzanie jest już niepotrzebne.

Przykład

```
Set ksiazki = // definicja zbioru książek

public Ksiazka przeszukajZbior(String tytul) {
   Iterator iter = ksiazki.iterator();
   Ksiazka ksiazka = null;


   do {
      if (! iter.hasNext()) {
         break;
      } else {
         Ksiazka ksiazka = (Ksiazka) iter.next();
         if (ksiazka.tytul().equals(tytul)) {
            return ksiazka;
         }
      }
   } while (true);

   return ksiazka;
}
```

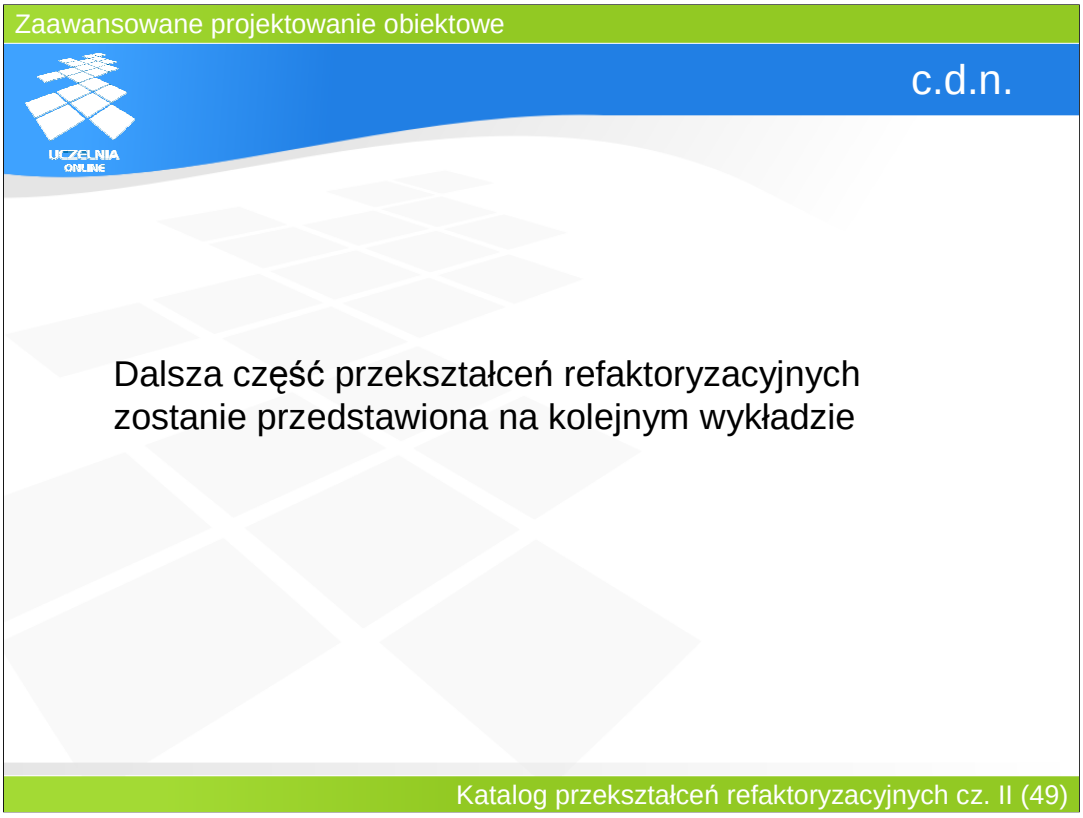

Ponieważ w poprzednim kroku zostały usunięte przypisania do flag, dlatego istniejący warunek wyjścia z pętli (while (true)) przestaje mieć znaczenie. Pętla staje się pętlą nieskończoną, którą można opuścić, gdy zostanie wykonana instrukcja *break* lub *return*.



Ostatni etap przekształcenia polega na uproszczeniu metody: warunek *iter.hasNext(),* który powodował wykonanie instrukcji *break*, został przeniesiony do warunku pętli, dzięki czemu jej ciało stało się znacznie czytelniejsze.

Usunięta została także zmienna przechowująca referencję do znalezionej Książki – była ona wykorzystywana jedynie w celu zwrócenia pustej referencji, co w rzeczywistości nie wymaga osobnej zmiennej.

W efekcie przekształcenia dwie zmienne lokalne, które pełniły rolę flag sterujących, zostały usunięte, a w ich miejsce zastosowano instrukcje modyfikujące przepływ sterowania.



Ostatnia, trzecia część katalogu przekształceń refaktoryzacyjnych, zostanie przedstawiona podczas kolejnego wykładu.