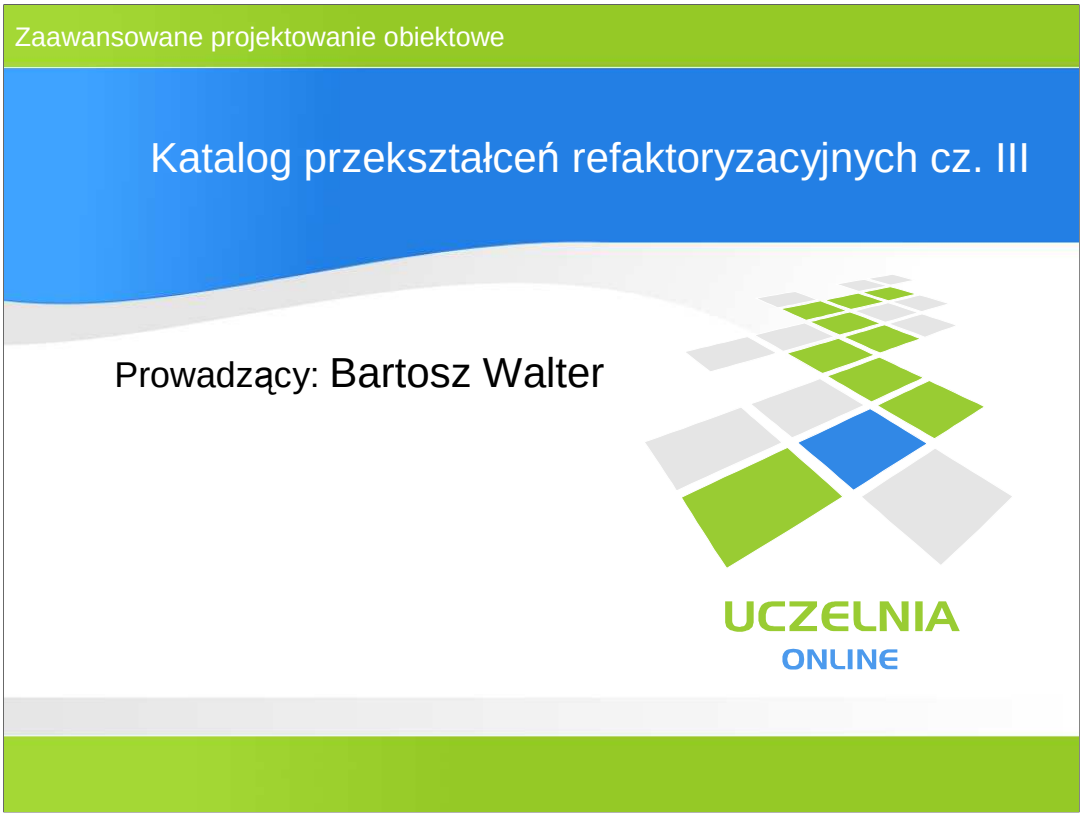
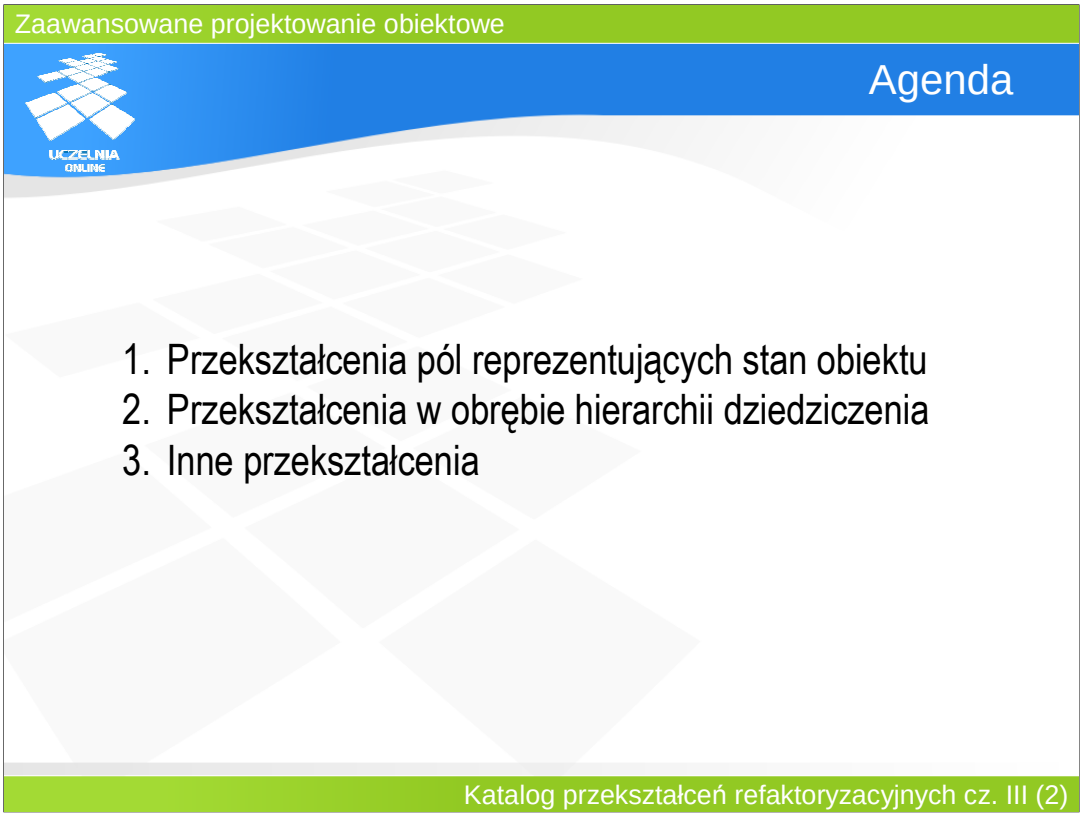
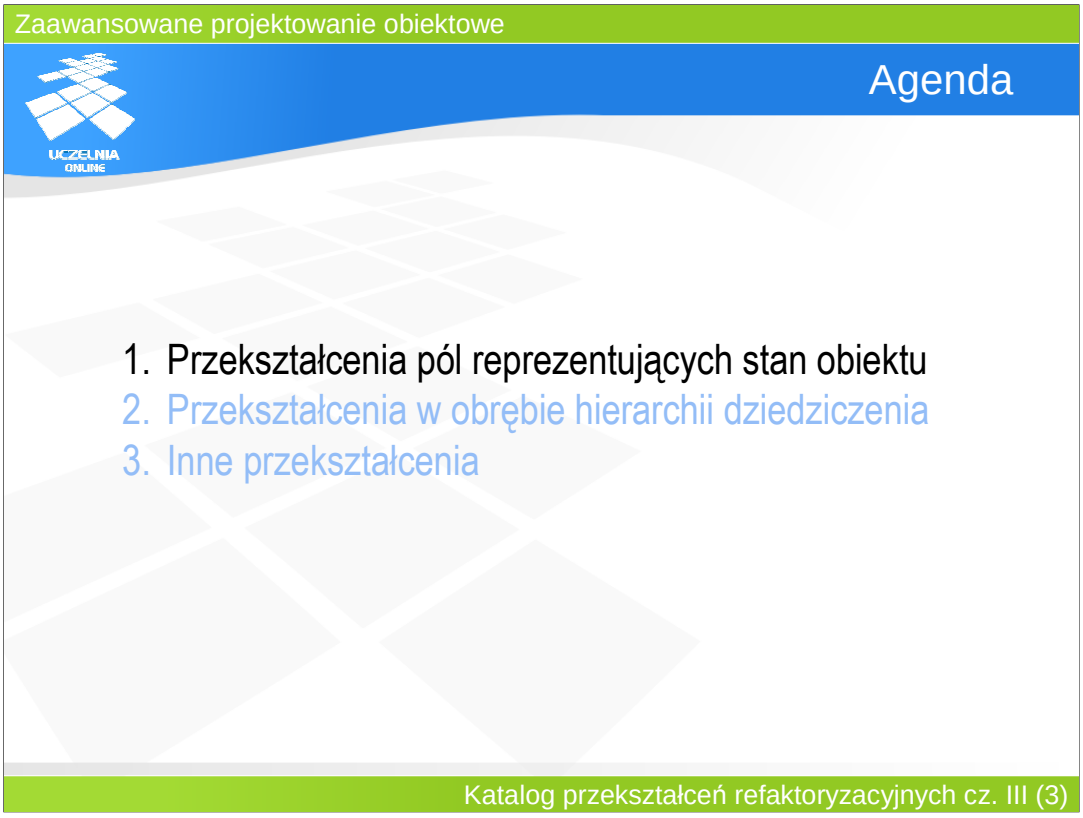

Zaawansowane projektowanie obiektowe
Katalog przekształceń refaktoryzacyjnych cz. III
Prowadzący: Bartosz Walter
UCZELNIA
ONLINE

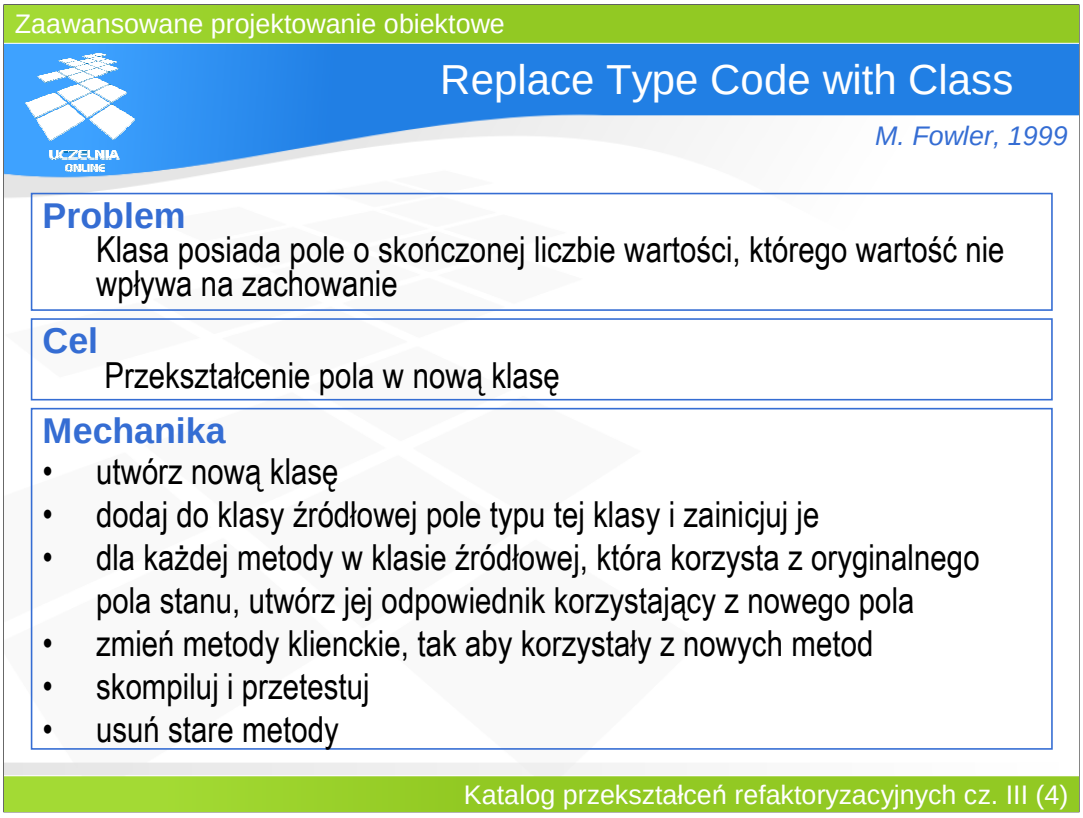


Wykład ten jest trzecim i ostatnim poświęconym przeglądowi przekształceń refaktoryzacyjnych. Podczas niego zostaną omówione następujące grupy przekształceń:

- przekształcenia pól reprezentujących stan obiektu
- przekształcenia hierarchii dziedziczenia
- pozostałe, nie ujęte w innych grupach przekształcenia



Pierwsza część dotyczy refaktoryzacji pól reprezentujących stan obiektu.



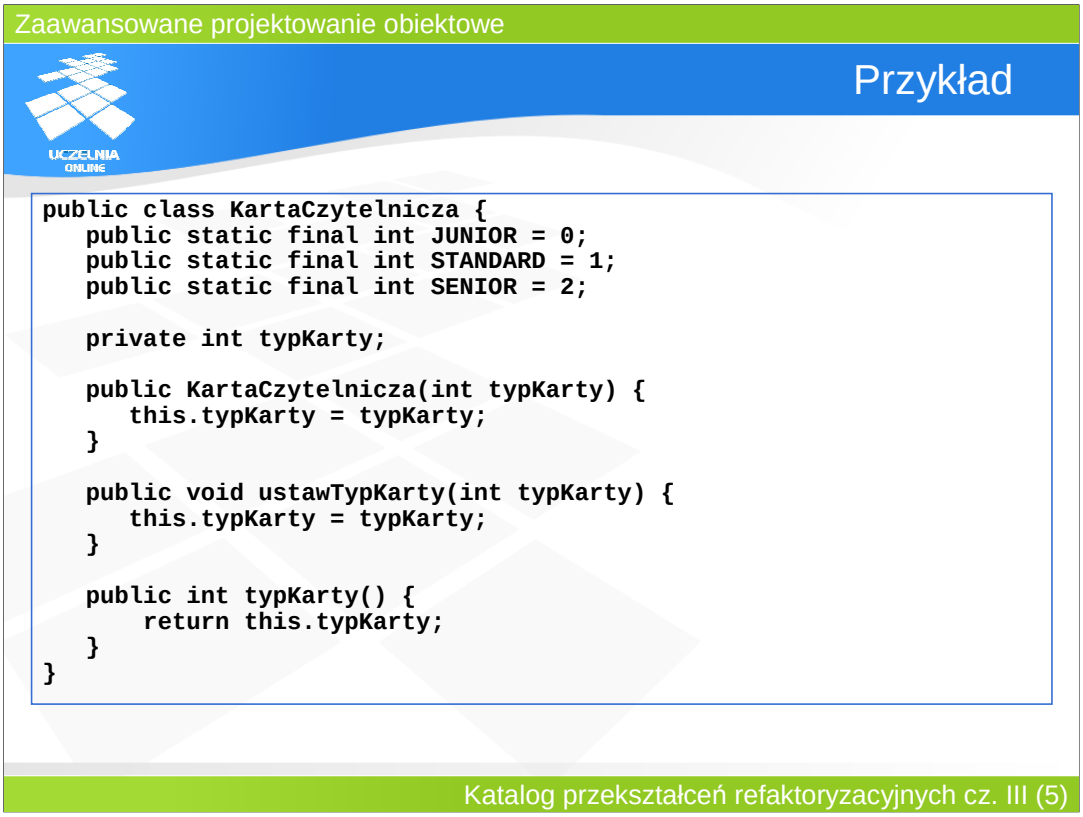
Przechowywanie informacji o stanie obiektu w postaci pola typu prymitywnego o zmiennej wartości jest bardzo powszechnym rozwiązaniem. Jednak w zależności od sposobu, w jaki pole reprezentujące stan wpływa na zachowanie swojego właściciela, z zagadnienia tego można wyróżnić kilka odrębnych problemów, i każdy z nich rozwiązywać za pomocą innego przekształcenia refaktoryzacyjnego.

Pierwsze z nich dotyczy sytuacji, gdy pole przyjmuje skończoną i dobrze określoną liczbę wartości, raczej nie zmienia wartości w trakcie swojego istnienia, oraz, co bardzo ważne, wartość pola nie wpływa na zachowanie obiektu. W związku z ostatnią cechą zmiana może zostać zrealizowana za pomocą jednej klasy, bez konieczności definiowania podklas czy polimorficznego pokrywania metod.

Celem przekształcenia jest transformacja pola w nową klasę, której wartości pól będą odzwierciedlały stan właściciela tego obiektu.

Pierwszym krokiem zmiany jest utworzenie nowej klasy oraz dodanie do klasy oryginalnej referencji do obiektu nowoutworzonej klasy. Następnie należy dla każdej metody korzystającej z dotychczasowego pola w klasie źródłowej stworzyć jej odpowiednik korzystający z nowego pola i nowej klasy. Warto zauważyć, że w ten sposób klasa w dotychczasowej postaci może istnieć bez zmiany swojej funkcjonalności. Kolejnym krokiem jest modyfikacja klientów, tak aby korzystały z nowych metod.

Po zakończeniu przekształcenia, które pozwala nam hermetyzować proces zarządzania zmiennymi reprezentującymi stan, można usunąć stare, nieużywane metody.



Jako przykład posłuży klasa KartaCzytelnicza, która przechowuje swój typ w postaci pola *typKarty*. Przyjmuje ono trzy wartości, reprezentowane przez stałe tej klasy: JUNIOR, STANDARD oraz SENIOR. Klasa KartaCzytelnicza posiada także metody pozwalające odczytać i zmienić jej typ.

Przykład

```
public class TypKarty {
   public static final TypKarty JUNIOR = new TypKarty(0);
   public static final TypKarty STANDARD = new TypKarty(1);
   public static final TypKarty SENIOR = new TypKarty(2);

   private int kodTypuKarty;

   public TypKarty(int kodTypuKarty) {
      this.kodTypuKarty = kodTypuKarty;
   }
   public void ustawTypKarty(int kodTypuKarty) {
      this. kodTypuKarty = kodTypuKarty;
   }
   public static int kodTypuKarty(int kodTypuKarty) {
       switch (kodTypuKarty) { case JUNIOR: return 0;
           case STANDARD: return 1; case SENIOR: return 2;
       }
   }
   public int typKarty() {
       return kodTypuKarty(this.kodTypuKarty);
   }
}
```


Pierwszym etapem przekształcenia jest utworzenie nowej klasy TypKarty i stworzenie w niej odpowiedników pól i stałych z klasy źródłowej. Klasa ta ponadto jest wyposażona w metody umożliwiające – podobnie jak dotychczas w klasie źródłowej – ustawianie, odczytywanie i dekodowanie typu karty.

Przykład

```
public class KartaCzytelnicza {
   public static final int JUNIOR = TypKarty.JUNIOR.typKarty(0);
   public static final int STANDARD = TypKarty.STANDARD.typKarty(1);
   public static final int SENIOR = TypKarty.SENIOR.typKarty(2);

   private TypKarty typKarty;

   public KartaCzytelnicza(int typKarty) {
      this.typKarty = TypKarty.kodTypuKarty(typKarty);
   }
   public KartaCzytelnicza(TypKarty typKarty) {
      this.typKarty = typKarty;
   }
   public void ustawTypKarty(TypKarty typKarty) {
      this.typKarty = typKarty;
   }
   public TypKarty typKarty() {
       return this.typKarty;
   }
}
```


Zastąpienie pola typu prymitywnego reprezentującego stan obiektu źródłowego poprzez referencję do nowej klasy wymusza aktualizację niektórych typów metod (w tym przypadku *typKarty()*) oraz stałych (są one obliczane poprzez odwołanie do definicji stałych w klasie TypKarty).

Przykład

```
public class KartaCzytelnicza {

   private TypKarty typKarty;

   public KartaCzytelnicza(int typKarty) {
      this.typKarty = TypKarty.typKarty(typKarty);
   }
   public KartaCzytelnicza(TypKarty typKarty) {
      this.typKarty = typKarty;
   }
   public void ustawTypKarty(TypKarty typKarty) {
      this.typKarty = typKarty;
   }
   public TypKarty typKarty() {
       return this.typKarty;
   }
   public int kodTypuKarty() {
       return typKarty.kodTypuKarty();
   }
}
```


W efekcie tego przekształcenia odpowiedzialność za reprezentację stanu obiektu została wydzielona z klasy KartaCzytelnicza do nowej klasy. KartaCzytelnicza posiada obecnie referencję do obiektu klasy TypKarty, który przejął od niej tę odpowiedzialność.

Replace Type Code with Subclasses

Problem

Klasa posiada niezmienne pole o skończonej liczbie wartości, którego wartość wpływa na zachowanie

Cel

Przekształcenie każdej możliwej wartości pola w podklasę

Mechanika

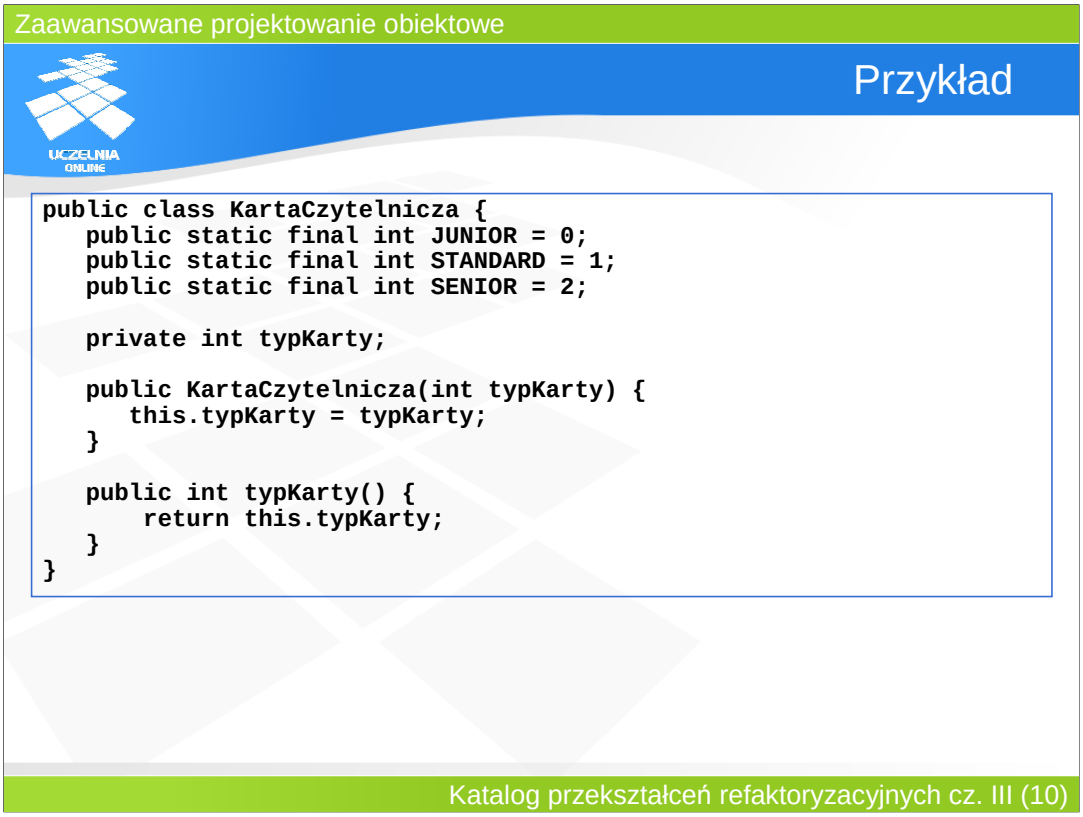
- wykonaj *Self-Encapsulate Field* na polu stanu
- dla każdej wartości pola, utwórz nową podklasę i dostosuj w niej metodę zwracającą wartość pola
- usuń pole z nadklasy, zadeklaruj metodę zwracającą wartość pola jako abstrakcyjną
- skompiluj i przetestuj

M. Fowler, 1999

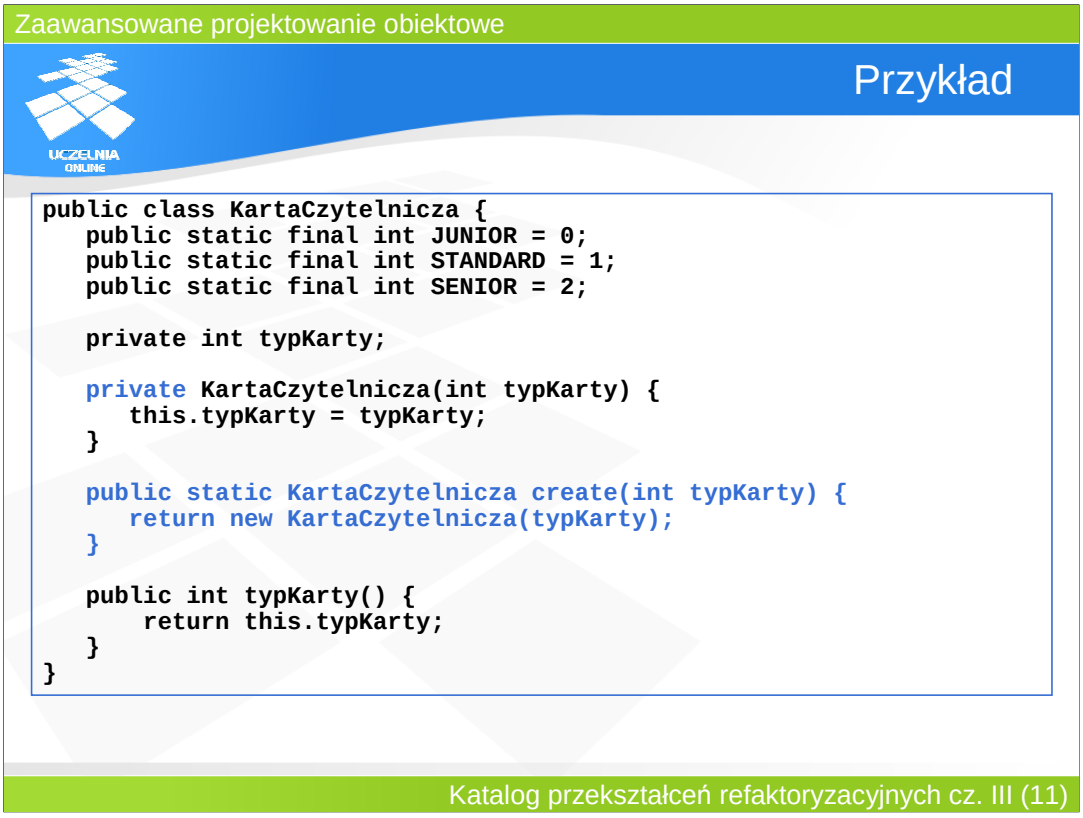
Drugie przekształcenie związane z polem reprezentującym stan obiektu dotyczy sytuacji, w której zmiana stanu oznacza zmianę zachowania niektórych metod obiektu. W tym wypadku użycie pojedynczej klasy jest niewystarczające: konieczne jest pokrycie metod, tak aby ich zachowanie odpowiadało stanowi, w jakim znajduje się obiekt.

Przekształcenie polega na wydzieleniu każdej wartości pola reprezentującego stan obiektu do oddzielnej podklasy. W ten sposób powstaje możliwość pokrywania metod wymagających zmiany zachowania, a jednocześnie wykorzystane jest dziedziczenie jako mechanizm powtórnego użycia kodu.

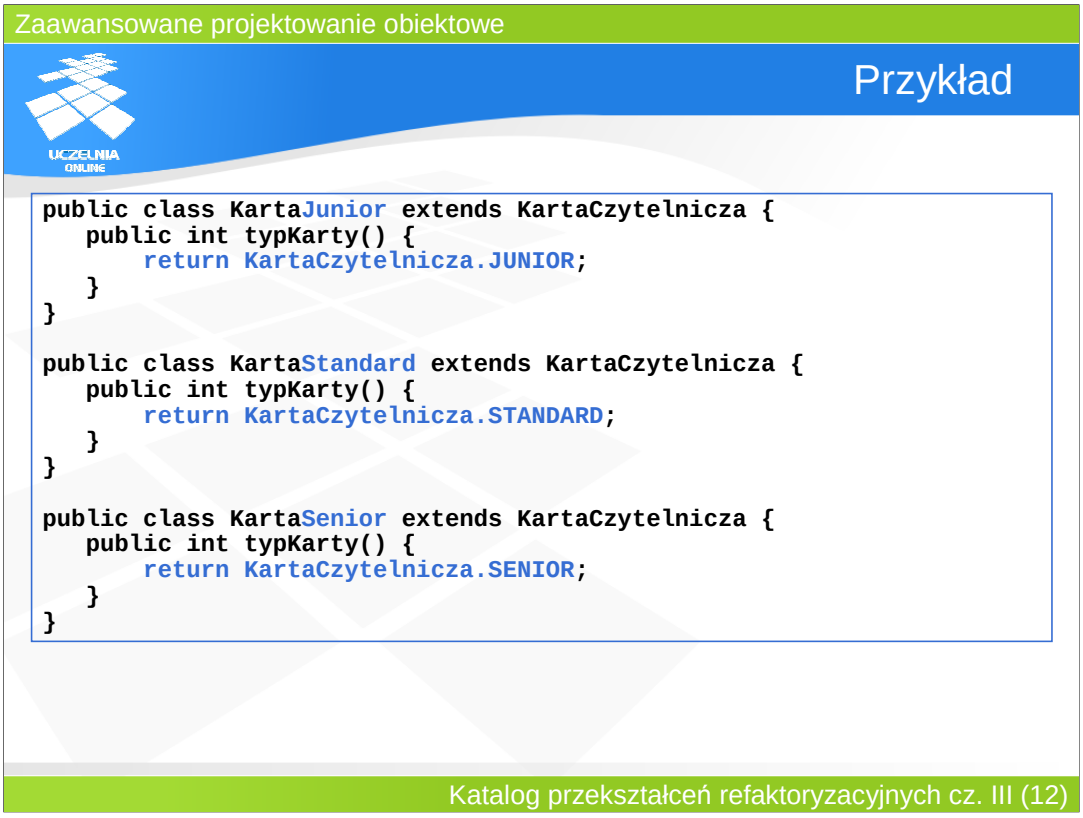
Pierwszym krokiem przekształcenia jest hermetyzacja pola stanu, tak aby było dostępne we własnej klasie i jej podklasach. Następnie dla każdej dopuszczalnej wartości tego pola należy utworzyć nową podklasę, pokrywając w niej metodę zwracającą wartość pola stanu w nadklasie na reprezentowany przez tę podklasę stan. Po przeniesieniu wszystkich wartości pola ono samo może zostać usunięte z nadklasy, a metody dostępowe do niego (*set* i *get*) – zadeklarowane jako abstrakcyjne.



Przykładem w dalszym ciągu pozostaje klasa KartaCzytelnicza. Przyjmijmy na potrzeby tego przykładu, że zmiana stanu obiektu (czyli w tym przypadku typu karty JUNIOR, STANDARD i SENIOR) wpływa na zachowanie niektórych metod w tej klasie, przez co niemożliwe jest wykonanie przekształcenia pola *typKlasy* jedynie w klasę.



Pierwszy krok polega na zastąpieniu konstruktora metodą-fabryką (czyli wykonaniu przekształcenia Replace Constructor with Factory Method): oznaczeniu konstruktora jako prywatnego oraz stworzeniu statycznej metody *create()*.

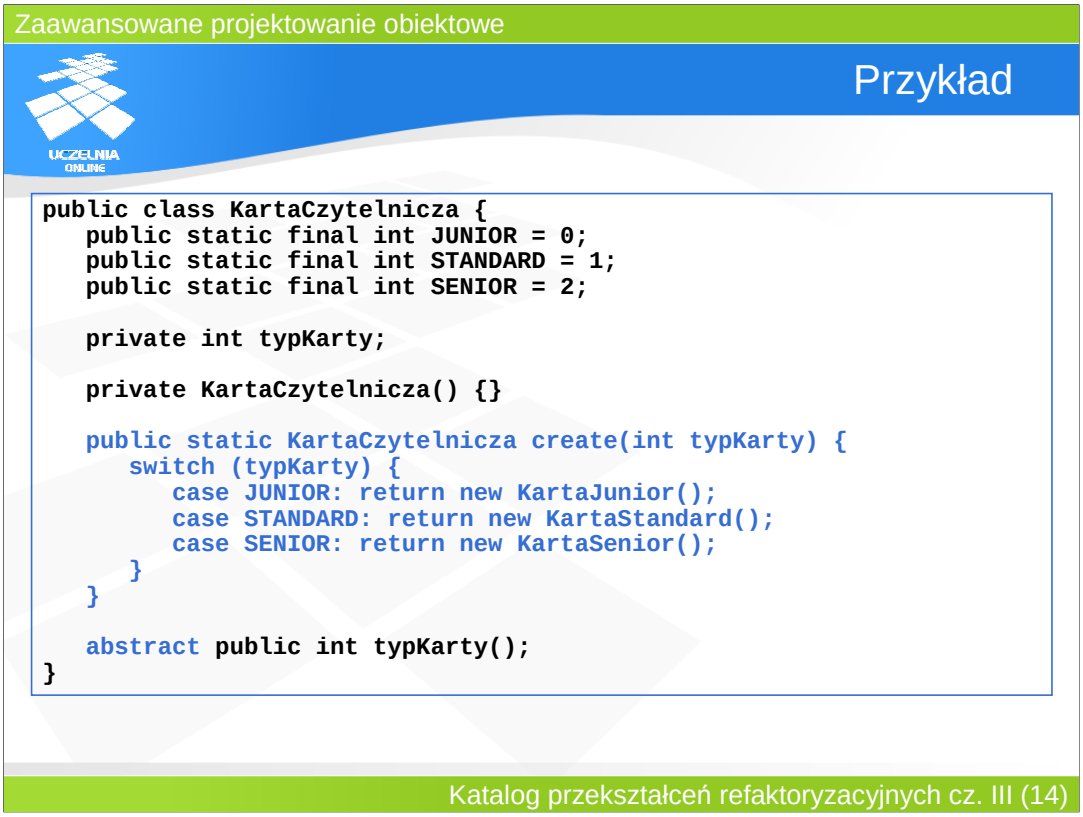


Następnie należy utworzyć podklasy klasy KartaCzytelnicza odpowiadające poszczególnym wartościom pola stanu: KartaJunior, KartaStandard i KartaSenior. Każda z tych klas pokrywa odziedziczoną z klasy KartaCzytelnicza metodę *typKarty()* i zwraca reprezentowany przez siebie typ karty.

Przykład

```
public class KartaCzytelnicza {
   public static final int JUNIOR = 0;
   public static final int STANDARD = 1;
   public static final int SENIOR = 2;

   private int typKarty;

   private KartaCzytelnicza(int typKarty) {
      this.typKarty = typKarty;
   }

   public static KartaCzytelnicza create(int typKarty) {
      if (typKarty == JUNIOR)
         return new KartaJunior();
      return new KartaCzytelnicza(typKarty);
   }

   public int typKarty() {
       return this.typKarty;
   }
}
```

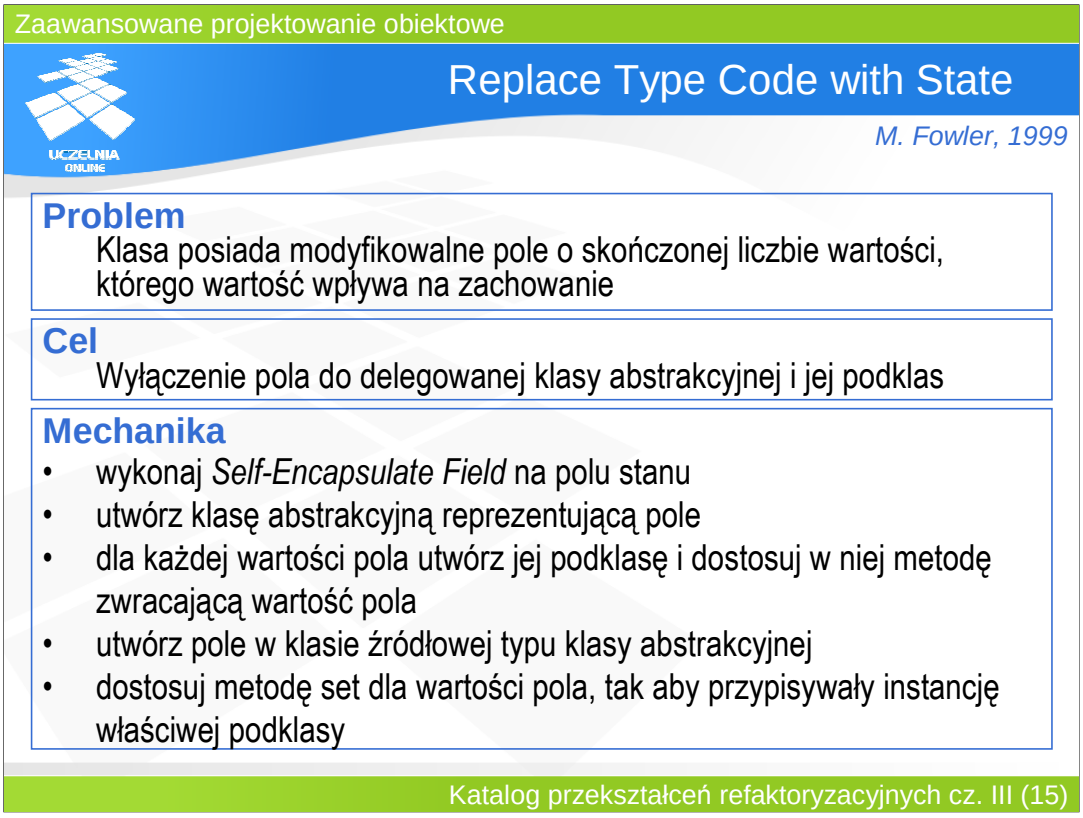

Teraz trzeba stworzone podklasy kolejno zintegrować z klasą KartaCzytelnicza poprzez rozszerzenie metody-fabryki. Jej funkcja polega na rozpoznaniu żądanego typu obiektu i utworzeniu instancji odpowiedniej podklasy. Klient za pomocą metody-fabryki może utworzyć obiekty żądanego typu.



Po zakończeniu integracji należy usunąć możliwość wywołania konstruktora klasy KartaCzytelnicza. W tym celu, zamiast fizycznie go usuwać, najlepiej stworzyć pusty prywatny konstruktor, ponieważ w przeciwnym przypadku kompilator i tak wygeneruje domyślny konstruktor bezparametrowy.

Drugą czynnością jest usunięcie ciała metody *typKarty()* w tej samej klasie i oznaczenie jej jako prywatnej. Od tego momentu klasa KartaCzytelnicza służy jedynie do przechowywania metody-fabryki i jako nadklasa dla klas reprezentujących poszczególne stany. Z punktu widzenia klienta jednak podklasy te są widoczne jedynie poprzez interfejs KartaCzytelnicza.

Efektem tego przekształcenia była zamiana pola reprezentującego stan klasy w grupę podklas, z których każda reprezentuje klasę w określonym stanie.



Przekształcenie to jest podobne do poprzedniego, a różnica polega na zastąpieniu mechanizmu dziedziczenia delegacją. Stosuje się je w tych przypadkach, gdy użycie podklas jest niemożliwe, np. w sytuacji, gdy obiekt musi zmienić swój stan (w przypadku podklas wymagałoby to utworzenia nowego obiektu, co często jest nie do przyjęcia z uwagi na różnicę referencji starego i nowego obiektu).

Celem przekształcenia jest wyłączenie sfery związanej ze stanami i zmiennym zachowaniem do oddzielnej hierarchii dziedziczenia, złożonej z klasy abstrakcyjnej (lub interfejsu) oraz jej podklas. Obiekt jest związany z obiektem reprezentującym określony stan relacją kompozycji, i poprzez nią deleguje wywołania do obiektu stanu.

Mechanika przekształcenia przebiega w następujących krokach. Na początku należy zahermetyzować pole stanu i udostępnić je przez metody dostępowe set/get. Następnym krokiem jest zdefiniowanie klasy abstrakcyjnej, której implementacje będą reprezentować stany obiektu, wraz z metodami set/get zapewniającymi dostęp do starego pola stanu. Inne metody zadeklarowane w tej klasie, które następnie będą pokrywane w podklasach, umożliwiają określenie, jakie zachowanie będzie zależało od stanu. Po stworzeniu klasy abstrakcyjnej należy zdefiniować jej podklasy, pokrywając odpowiednio metody set/get dostępu do pola stanu dziedziczone z nadklasy. W tym momencie następuje zdefiniowanie nowego mechanizmu opisu stanu: z pola przechowywanego bezpośrednio w klasie źródłowej do nowej, niezależnej hierarchii klas reprezentujących stany. Polega to na stworzeniu pola w klasie źródłowej, wskazującego na obiekt typu klasy abstrakcyjnej, a następnie zmianie metod set/get, tak aby odwoływały się do nowego pola reprezentującego stan. Ostatnim krokiem jest usunięcie starego pola stanu.

Przykład

```
public class KartaCzytelnicza {
   public static final int JUNIOR = 0;
   public static final int STANDARD = 1;
   public static final int SENIOR = 2;

   private int kodTypuKarty;

   public KartaCzytelnicza(int kodTypuKarty) {
      this. kodTypuKarty = kodTypuKarty;
   }
   public int kodTypuKarty() {
       return this. kodTypuKarty;
   }
   public double oplata() {
      switch (kodTypuKarty()) {
         case JUNIOR: return 0;
         case STANDARD: return 100 – (10 * okresCzytelnictwa);
         case SENIOR: return 50;
      }
   }
}
```


Przykład w dalszym ciągu jest oparty na klasie KartaCzytelnicza, której stan jest opisywany polem *typKarty*. W zależności od wartości tego pola metoda *oplata()* inaczej oblicza wysokość opłaty za dostęp do biblioteki.

Przykład

```
abstract public class TypKarty {
   abstract public int kodTypuKarty();
}

public class TypKartyJunior extends TypKarty {
   public int kodTypuKarty() {
       return KartaCzytelnicza.JUNIOR;
   }
}

public class TypKartyStandard extends TypKarty {
   public int kodTypuKarty() {
       return KartaCzytelnicza.STANDARD;
   }
}

public class TypKartySenior extends TypKarty {
   public int kodTypuKarty() {
       return KartaCzytelnicza.SENIOR;
   }
}
```


Pierwszym krokiem jest stworzenie klasy abstrakcyjnej TypKarty, w którym zdefiniowana jest metoda zwracająca wartość starego pola stanu. Metoda ta jest pokrywana w klasach potomnych TypKartyJunior, TypKartyStandard i TypKartySenior.

Przykład

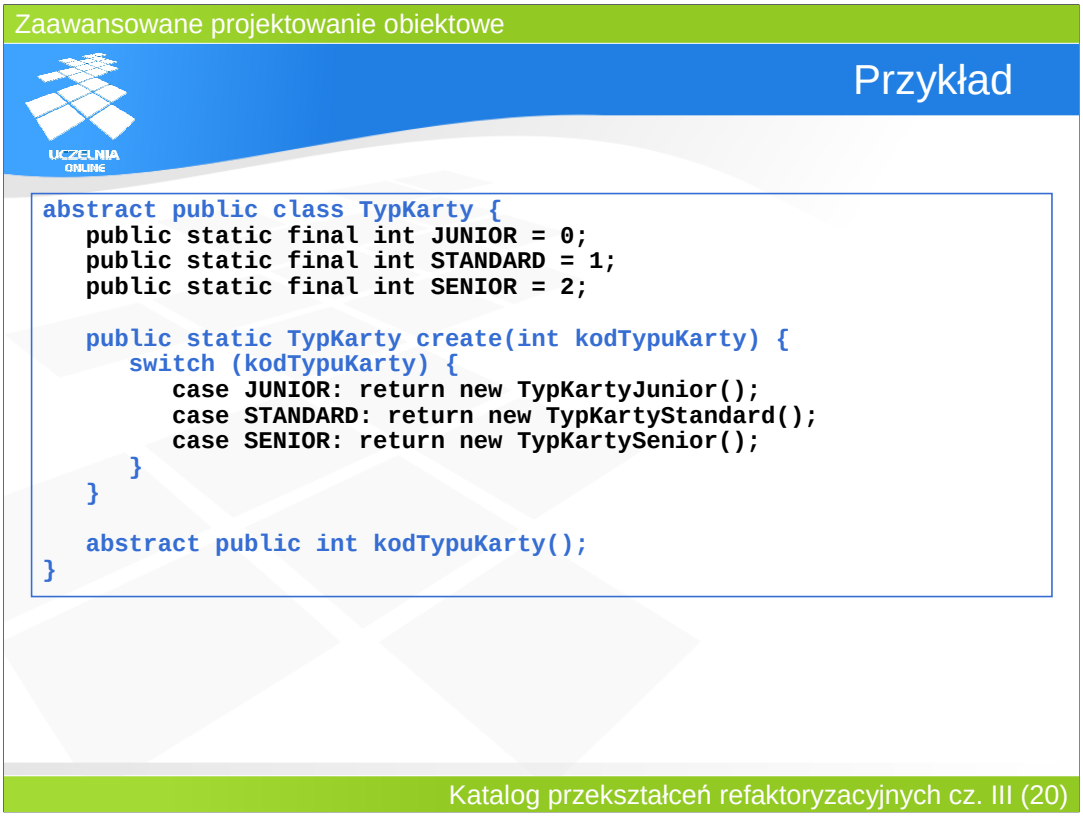
```
public class KartaCzytelnicza {
   public static final int JUNIOR = 0;
   public static final int STANDARD = 1;
   public static final int SENIOR = 2;

   private TypKarty typKarty;

   public KartaCzytelnicza(TypKarty typKarty) {
      this.typKarty = typKarty;
   }
   public int kodTypuKarty() {
       return this.typKarty.kodTypuKarty();
   }
   public double oplata() {
      switch (kodTypuKarty()) {
         case JUNIOR: return 0;
         case STANDARD: return 100 – (10 * okresCzytelnictwa);
         case SENIOR: return 50;
      }
   }
}
```


Teraz można utworzyć w klasie źródłowej KartaCzytelnicza pole typu TypKarty. Będzie ono decydowało o stanie obiektu. Wszystkie metody, które dotychczas posługiwały się starym polem stanu, są modyfikowane, tak aby korzystały z nowego pola stanu. Ilustruje to metoda *kodTypuKarty()*, która nie zwraca kodu typu karty bezpośrednio, ale jako wynik delegacji przez pole *typKarty*.

Przykład

```
public class KartaCzytelnicza {
   private TypKarty typKarty;

   public KartaCzytelnicza(TypKarty typKarty) {
      this.typKarty = typKarty;
   }
   public int kodTypuKarty() {
       return this.typKarty.kodTypuKarty();
   }
   public double oplata() {
      switch (kodTypuKarty()) {
         case JUNIOR: return 0;
         case STANDARD: return 100 – (10 * okresCzytelnictwa);
         case SENIOR: return 50;
      }
   }
}
```


W tym kroku można usunąć stałe związane z typami z klasy źródłowej, przenosząc je do klas reprezentujących poszczególne stałe.



Klasa TypKarty przechowuje stałe reprezentujące poszczególne stany obiektu, jak również definiuje metodę fabrykę tworzącą obiekty jednej z jej podklas w zależności od przekazanego parametru. Zasada działania tego mechanizmu jest identyczna jak w przypadku poprzedniego przekształcenia, które reprezentowało stany jako podklasy.

W wyniku przekształcenia pole typu prymitywnego, przechowywane w klasie źródłowej, zostało przetransformowane do postaci oddzielnej klasy i jej podklas. Mechanizm ten pozwala swobodnie modyfikować wartość pola stanu w trakcie wykonywania programu (należy w tym celu utworzyć instancję innej z podklas), i rozróżniać stany na podstawie klasy. Jednak rozwiązanie to nie wykorzystuje do końca możliwości oferowanych przez polimorfizm, dlatego przekształcenie to można kontynuować.

Replace Type Code with Polymorphism

M. Fowler, 1999

Problem

Wyrażenie warunkowe wykonuje różne akcje w zależności od stanu obiektu

Cel

Wyłącz każdą gałąź instrukcji warunkowej do metody w podklasie reprezentującej stan obiektu

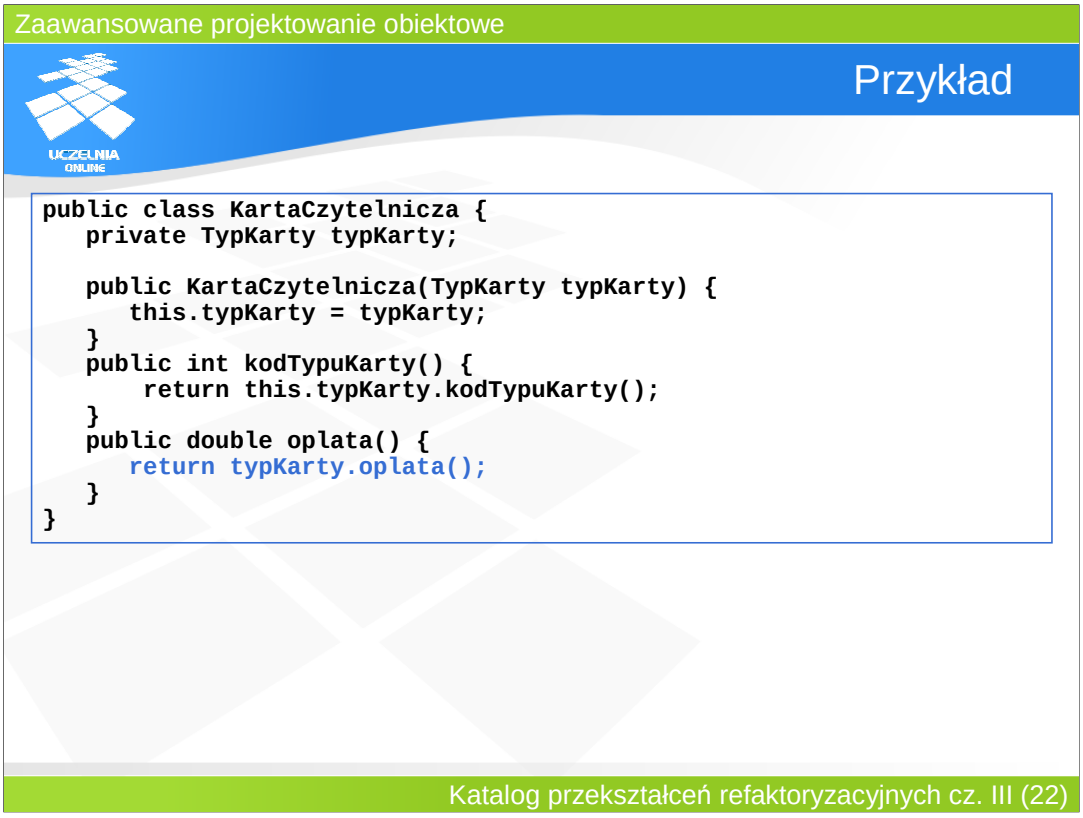
Mechanika

- wyłącz wyrażenie warunkowe do nowej metody
- przenieś metodę do klasy abstrakcyjnej reprezentującej stan
- pokryj metodę w podklasach, kopiując do nich odpowiednią gałąź wyrażenia warunkowego
- skompiluj i przetestuj
- usuń przeniesioną gałąź w klasie abstrakcyjnej

Przekształcenie to stanowi ciąg dalszy poprzedniego, stosującego do obsługi pola stanu zgodne z wzorcem State. Tym razem ten sam wzorzec zostanie wykorzystany także do usunięcia warunkowego wyboru określonego zachowania obiektu i przeniesienia związanych z nim metod do podklasy reprezentującej stan.

Pierwszym krokiem przekształcenia jest wyłączenie wyrażenia warunkowego do osobnej metody, tak aby było możliwe przeniesienie jej do klasy abstrakcyjnej reprezentującej stan obiektu (zob. przekształcenie Replace Type Code with State). Następnym krokiem jest przeniesienie tej metody do tej klasy i pozostawienie w obiekcie źródłowym delegacji do tej instancji tej klasy. W ten sposób wywołanie metody, której zachowanie zależy od stanu, jest zawsze delegowane do bieżącego obiektu stanu.

Gdy metoda znajdzie się w klasie abstrakcyjnej, wówczas należy pokryć ją we wszystkich podklasach, kopiując do nich właściwą gałąź wyrażenia warunkowego, związaną z danym stanem. Po przeniesieniu wszystkich gałęzi metoda w klasie abstrakcyjnej może zostać zadeklarowana jako abstrakcyjna.

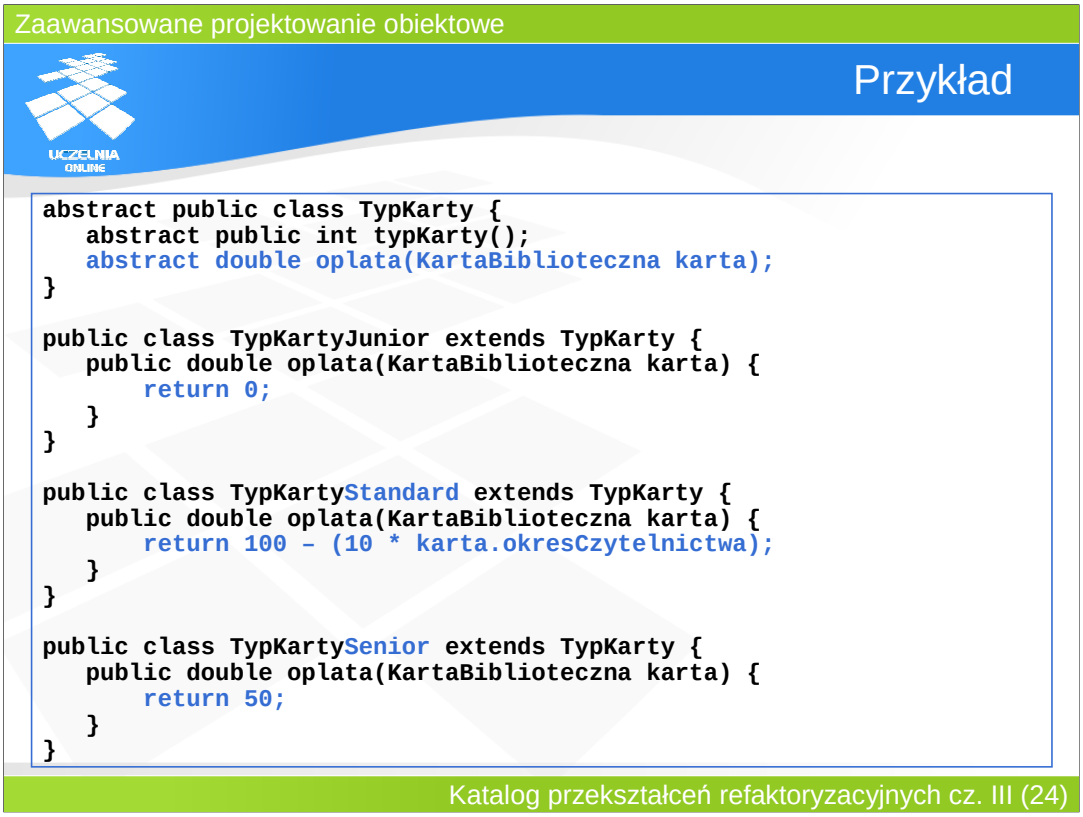


Ponownie jako przykład posłuży klasa KartaCzytelnicza. Zdefiniowana w niej metoda *oplata()* jest delegacją do klasy TypKarty, a zatem pominięty zostaje pierwszy krok przekształcenia (zwykle jest on realizowany przez przekształcenie Replace Type Code with State).

Przykład

```
abstract public class TypKarty {
   public static final int JUNIOR = 0;
   public static final int STANDARD = 1;
   public static final int SENIOR = 2;

   public static TypKarty create(int kodTypuKarty) {
      switch (kodTypuKarty) {
         case JUNIOR: return new TypKartyJunior();
         case STANDARD: return new TypKartyStandard();
         case SENIOR: return new TypKartySenior();
      }
   }
   abstract public int typKarty();

   public double oplata(KartaCzytelnictwa karta) {
      switch (kodTypuKarty) {
         case JUNIOR: return 0;
         case STANDARD: return 100 - (10 * okresCzytelnictwa);
         case SENIOR: return 50;
      }
   }
}
```

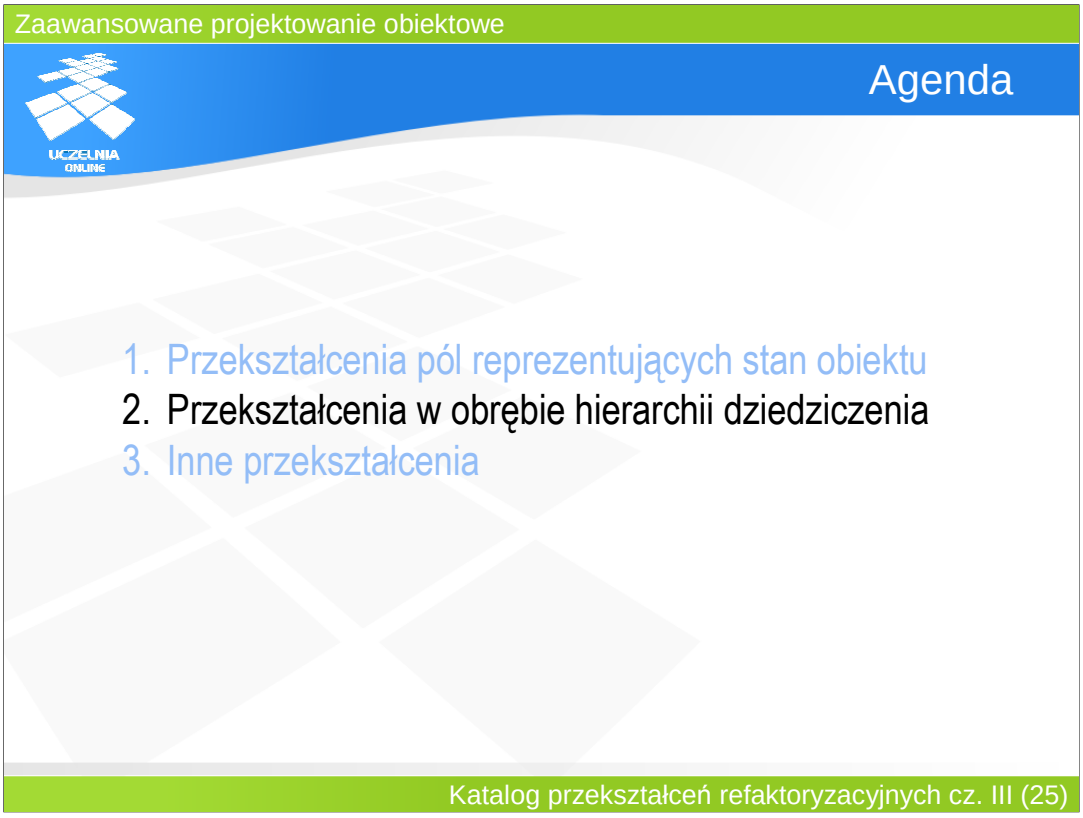

Cała logika metody *oplata()* zawarta jest w klasie TypKarty. Metoda ta dokonuje wyboru typu karty na podstawie stanu karty i oblicza właściwą wartość metody dla poszczególnych stanów: JUNIOR, STANDARD i SENIOR. Jednak rozwiązanie to powoduje, że dodanie nowego stanu (czyli TypuKarty) związane jest z modyfikacją metody *oplata()* w klasie TypKarty.



Dlatego do wykonania właściwego kodu wykorzystywane są podklasy klasy TypKarty, które reprezentują poszczególne stany obiektu. Metoda *oplata()* jest pokrywana w tych podklasach poprzez skopiowanie do nich tylko wybranej gałęzi wyrażenia warunkowego, związanej z określonym stanem.

W efekcie tego przekształcenia w miejsce zmiennego zachowania obiektu, zależnego od jego stanu, powstała niezależna hierarchia dziedziczenia zawierająca definicje poszczególnych stanów zamknięte w postaci oddzielnych klas. Każda klasa definiuje własne implementacje metod, których zachowanie zmienia się wraz ze stanem.

Dzięki temu możliwe stało się także usunięcie wyrażenia warunkowego. Przekształcenie to może zatem służyć nie tylko do restrukturyzacji problemu pola stanu, ale również złożonych wyrażeń warunkowych.



Drugą grupą przekształceń omawianych podczas tego wykładu są refaktoryzacje wewnątrz hierarchii dziedziczenia, związane z przenoszeniem metod, pól i konstruktora do pod- i nadklas.

Pull Up Method

Problem

Podklasy posiadają identyczną lub podobną metodę

Cel

Przesunięcie metody do nadklasy

Mechanika

- ujednolić metody w podklasach
- zadeklaruj metodę w nadklasie
- skopiuj do niej ciało metody w jednej z podklas
- usuń metodę z jednej z podklas
- skompiluj i przetestuj
- powtórz dla pozostałych podklas

M. Fowler, 1999

Pierwszym z przekształceń jest przeniesienie w górę hierarchii dziedziczenia metody, która występuje w wielu podklasach posiadających wspólną nadklasę. Celem jest zatem przeniesienie metody do nadklasy, tak aby była ona dziedziczona, a nie definiowana w poszczególnych podklasach.

Krokiem wstępnym tego przekształcenia jest ujednolicenie metod w poszczególnych podklasach. Następnie należy zadeklarować metodę w nadklasie i skopiować do niej ciało metody w jednej z podklas. Jeżeli wszystkie podklasy definiowały tę metodę, wówczas nadal nie powinna wystąpić zmiana zachowania programu.

W kolejnych krokach metoda jest usuwana z podklas (wówczas jest ona dostępna wyłącznie poprzez dziedziczenie z nadklasy).

W efekcie metoda jest usuwana z wszystkich podklas i przeniesiona do nadklasy. Jeżeli niektóre podklasy definiują metodę odmiennie od pozostałych, wówczas mogą pozostawić metodę pokrytą.

Pull Up Constructor Body

Problem

Konstruktory w podklasach są bardzo podobne do siebie

Cel

Przesunięcie wspólne elementy konstruktora do nadklasy

Mechanika

- zadeklaruj konstruktor w nadklasie
- przenieś do niego wspólne elementy konstruktorów z podklas
- wywołaj konstruktor nadklasy na początku konstruktorów w podklasach
- skompiluj i przetestuj

M. Fowler, 1999

Nieco innym problemem jest przeniesienie do nadklasy części konstruktora. Podstawową różnicą pomiędzy metodą i konstruktorem jest fakt, że konstruktor nie może być odziedziczony ani wywołany polimorficznie. Wywołanie konstruktora zawsze tworzy instancję określonej klasy, na rzecz której został wywołany. W przypadku dziedziczenia podklasa musi zawsze wywołać konstruktor nadklasy jako pierwszą instrukcję swojego konstruktora – i dlatego przenoszenie do nadklasy jego części różni się od zwykłego przeniesienia metody.

Przekształcenie zaczyna się od zadeklarowania konstruktora w nadklasie i przeniesienie do niego wspólnych fragmentów kodu konstruktorów z podklas. Następnie w konstruktorach podklas należy wywołać konstruktor nadklasy jako pierwszą instrukcję (poprzez kwalifikator *super*).

Przykład 1

```
public class Wydawnictwo {
    protected String tytul;
    protected String id;

    protected Wydawnictwo() {
    }
}

public class Ksiazka extends Wydawnictwo {
    private int liczbaStron;

    public Ksiazka (String tytul, String id, int liczbaStron){
        this.tytul = tytul;
        this.id = id;
        this.liczbaStron = liczbaStron;
    }
}
```


Jako przykład rozpatrzymy relację pomiędzy klasami Wydawnictwo (nadklasą) oraz Książka (podklasą). Obecnie klasa Wydawnictwo posiada pusty konstruktor chroniony, natomiast klasa Książka definiuje swój własny, pełny konstruktor. Warto zwrócić uwagę, że w tym przykładzie konstruktor klasy Wydawnictwo musi być bezparametrowy, aby mógł być niejawnie wywołany w konstruktorze klasy Książka.

Przykład 1

```
public class Wydawnictwo {
    protected String tytul;
    protected String id;

    protected Wydawnictwo(String tytul, String id) {
        this.tytul = tytul;
        this.id = id;
    }
}

public class Ksiazka extends Wydawnictwo {
    private int liczbaStron;

    public Ksiazka (String tytul, String id, int liczbaStron){
        super(tytul, id);
        this.liczbaStron = liczbaStron;
    }
}
```


Wybrane elementy konstruktora klasy Książka zostały przeniesione do konstruktora klasy Wydawnictwo, natomiast sam konstruktor musi być wywołany (tym razem już jawnie) w klasie Książka.

Przykład 2

```
public class Wydawnictwo {
    protected String tytul;
    protected String id;

    public void wyznaczRegal() {...}
    public boolean czyDuze() {...}
}

public class Ksiazka extends Wydawnictwo {
    private int liczbaStron;

    public Ksiazka (String tytul, String id, int liczbaStron) {
        super(tytul, id);
        this.liczbaStron = liczbaStron;
        if (czyDuze())
            wyznaczRegal();
    }
    public boolean czyDuze() {
       return liczbaStron > 150;
    }
}
```


Drugi przykład jest bardziej skomplikowany i pokazuje, że realizacja tego przekształcenia może napotykać na problemy.

Tym razem w klasie Wydawnictwo są zdefiniowane dwie metody: *wyznaczRegal(),* która wskazuje, na którym regale powinno znaleźć się Wydawnictwo, oraz *czyDuze(),* określające rozmiar Wydawnictwa. Metody te są wykorzystane w konstruktorze klasy Książka, w którym wyznaczenie regału jest wykonywane tylko dla dużych Wydawnictw. Należy jednak zauważyć, że metoda *czyDuze()* jest pokryta w klasie Książka, co wskazuje, że metoda *wyznaczRegal()* jest wywoływana w nadklasie, natomiast metoda *czyDuze()* – lokalnie.

Taki graf wywołań powoduje, że nie można w prosty sposób przenieść fragmentu konstruktora Książki do klasy Wydawnictwo, ponieważ klasa ta posiada własną definicję metody *czyDuze(),* której nie można w takim wypadku zastosować.

Przykład 2

```
public class Wydawnictwo {
    protected String tytul;
    protected String id;
    public void wyznaczRegal() {...}
    public boolean czyDuze() {...}
    protected inicjuj() {
        if (czyDuze())
            wyznaczRegal();
    }
}

public class Ksiazka extends Wydawnictwo {
    private int liczbaStron;
    public Ksiazka (String tytul, String id, int liczbaStron) {
        super(tytul, id);
        this.liczbaStron = liczbaStron;
        inicjuj();
    }
    public boolean czyDuze() {
       return liczbaStron > 150;
    }
}
```


Rozwiązaniem takiego problemu jest przeniesienie fragmentów kodu nie do konstruktora, ale do osobnej metody *inicjuj()*, która zostanie wywołana w konstruktorze. Metoda ta ta może zdefiniować szkielet (zob. wzorzec Template Method), którego elementy – metody *czyDuze()* i *wyznaczRegal()* – mogą zostać pokryte w podklasach. Wywołanie metody *inicjuj()* w klasie Książka powoduje zatem wywołanie właściwych metod *czyDuze()* i *wyznaczRegal().*

Push Down Method

Problem

Metoda ma znaczenie tylko dla niektórych podklas

Cel

Przesunięcie metody do tych podklas

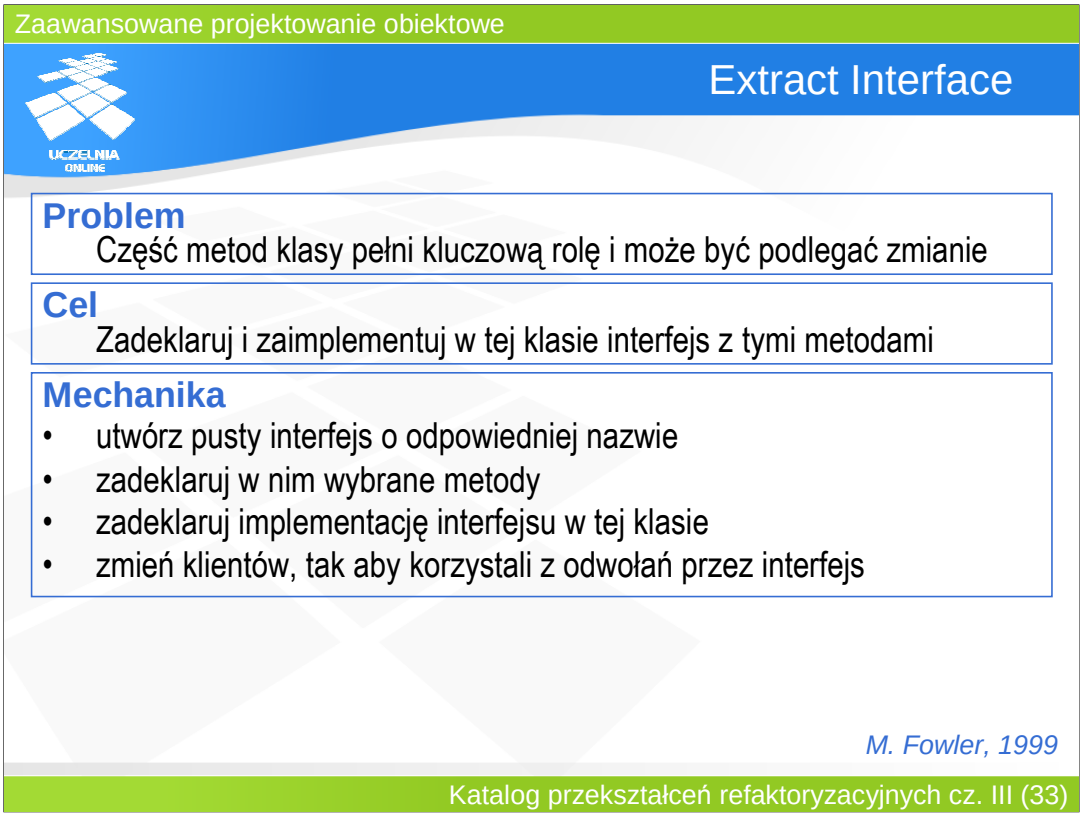
Mechanika

- zadeklaruj metodę w podklasach
- skopiuj do nich ciało metody z nadklasy
- usuń metodę z nadklasy
- skompiluj i przetestuj
- usuń metodę z podklas, które jej nie potrzebują

M. Fowler, 1999

To przekształcenie jest komplementarne w stosunku do przeniesienia metody do nadklasy. Jeżeli metoda zdefiniowana w nadklasie ma znaczenie jedynie dla wybranych podklas, wówczas bardziej uzasadnione może okazać się przeniesienie jej do tych podklas i usunięcie z nadklasy.

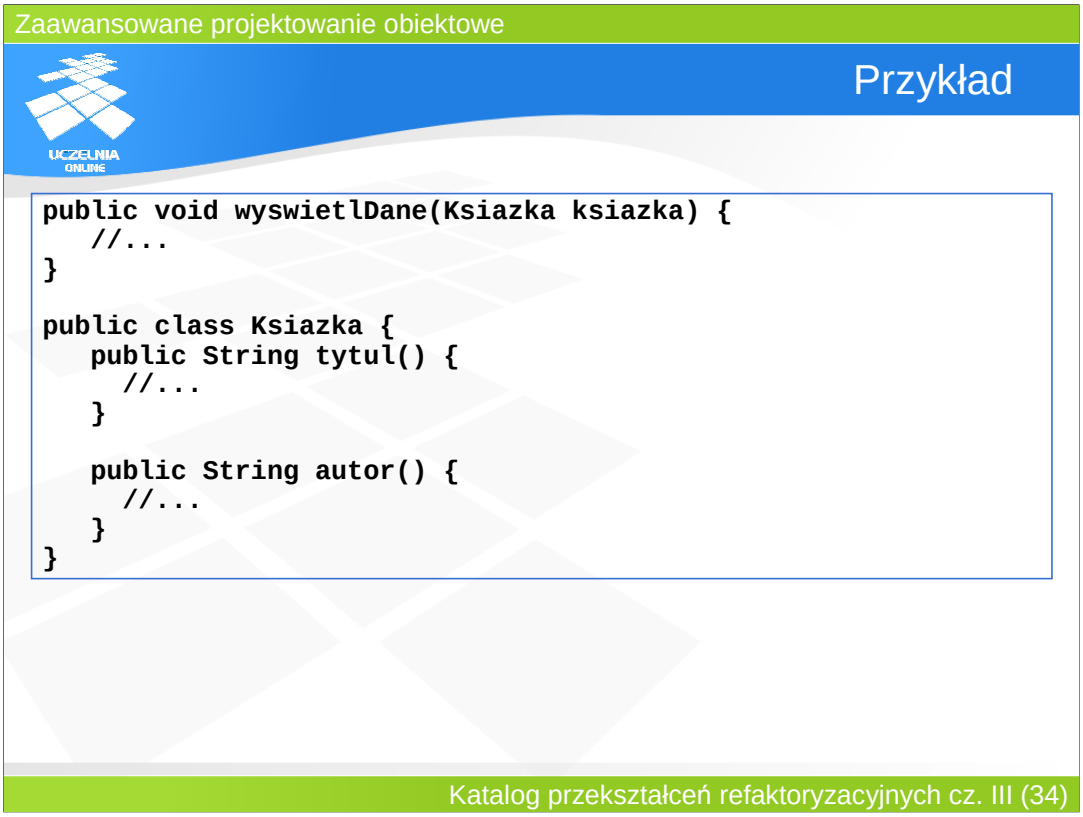
Przekształcenie rozpoczyna się od zadeklarowania metody we wszystkich podklasach poprzez skopiowanie do nich ciała z nadklasy. Następnie metoda z nadklasy jest usuwana, jednak z uwagi na uprzednie przeniesienie jej do wszystkich podklas, zachowanie programu nie zmienia się. Ostatnim krokiem jest usunięcie metody z tych podklas, które jej nie potrzebują.



Wyłączenie interfejsu jest przekształceniem czysto syntaktycznym, które nie może wprowadzić zmian w zachowaniu programu. Jednak pozwala poprawić jakość projektu przez zastąpienie powiązań bezpośrednio z klasami powiązaniami przez interfejs.

Przekształcenie to stosowane jest zwykle w sytuacji, gdy grupa metod pełni kluczową rolę w klasie, a ich zmiana musi być dokonana jednocześnie. Celem przekształcenia jest zadeklarowanie tych metod w nowym interfejsie, oraz zaimplementowanie tego interfejsu w klasie. W efekcie wszystkie klasy klienckie mogą odwoływać się do tej klasy poprzez interfejs, co pozwala zmienić jej implementację.

Pierwszym krokiem jest utworzenie interfejsu i zadeklarowanie w nim wybranych metod (bez zmiany ich sygnatur). Następnie należy zadeklarować implementację tego interfejsu w klasie. Zmiana ta powoduje jedynie rozdzielenie typu i jego implementacji, a więc nie zmienia zachowania systemu. Ostatnim krokiem jest zmiana klientów, tak aby zależały od interfejsu, a nie bezpośrednio od klasy go implementującej.



Dla przykładu rozpatrzmy sposób wyświetlania informacji o książce. Metoda *wyswietlDane()* przyjmuje jako argument obiekt klasy Książka. Obiekt ten posiada dwie metody: *tytul()* i *autor()*. Aby zmniejszyć zależność metody *wyswietlDane()* od klasy Książka, a jednocześnie umożliwić zastąpienie tej klasy inną klasą, można wyłączyć ją do nowego interfejsu.

Przykład

```
public void wyswietlDane(Wydawnictwo ksiazka) {
   //...
}

public class Ksiazka implements Wydawnictwo {
   public String tytul() {
     //...
   }

   public String autor() {
     //...
   }
}

public interface Wydawnictwo {
   public String tytul();
   public String autor();
}
```


Po wykonaniu przekształcenia powstał nowy interfejs – Wydawnictwo, który deklaruje identyczne dwie metody co klasa Książka. Zadeklarowanie tego interfejsu w klasie Książka nie wymaga zatem żadnych innych zmian w kodzie. Dzięki tej zmianie możliwe jest rozszerzenie typu argumentu metody *wyswietlDane()* do interfejsu Wydawnictwo, co umożliwia użycie w przyszłości innych implementacji tego interfejsu (np. czasopisma, książki wielotomowej etc.).

W efekcie w miejsce klasy, która definiowała jednocześnie typ i implementację powstały dwa odrębne elementy: interfejs określający zakres odpowiedzialności oraz klasa będąca jego realizacją.

Extract Superclass

Problem

Kilka klas posiada bardzo podobne metody i zakres odpowiedzialności

Cel

Połączenie ich poprzez wspólną nadklasę

Mechanika

- utwórz pustą nadklasę abstrakcyjną
- podziel metody na identyczne i całkowicie różne; identyczne są przenoszone do nadklasy, a całkowicie różne – pozostają w podklasach (ewentualnie są zadeklarowane w nadklasie jako abstrakcyjne)
- wykonaj *Pull Up Fields, Pull Up Methods, Pull Up Constructor Body*
- skompiluj i przetestuj po każdej zmianie

M. Fowler, 1999

Przekształcenie to pozwala ujednolicić grupę niespokrewnionych klas o podobnych sygnaturach metod i podobnym zakresie odpowiedzialności. W efekcie powstaje ich nadklasa, do której zostają przeniesione wspólne metody i pola.

Mechanika rozpoczyna się od utworzenia pustej nadklasy i zadeklarowania dziedziczenia po niej w każdej z analizowanych klas (oczywiście, w języku Java warunkiem poprawności takiego przekształcenia jest, aby nie były one już podklasami jakiejkolwiek innej klasy). Następnie, przeprowadzając zmiany w sygnaturach metod, należy doprowadzić do sytuacji, w której sygnatury te (i zakresy odpowiedzialności) będą albo identyczne, albo całkowicie różne. Metody identyczne należy przekopiować do nadklasy, a następnie kolejno usuwać z podklas. W przypadku metod całkowicie różnych można zostawić je w podklasach, w uzasadnionych przypadkach deklarując je w nadklasie jako abstrakcyjne (wówczas jednak wszystkie podklasy muszą je zaimplementować) lub o pustej domyślnej implementacji.

W efekcie przekształcenia powstaje nadklasa, która definiuje w jednym miejscu wspólne metody i pola grupy klas.

Extract Subclass

Problem

Część metod klasy jest wykorzystywana tylko przez niektóre obiekty

Cel

Utworzenie nowej podklasy

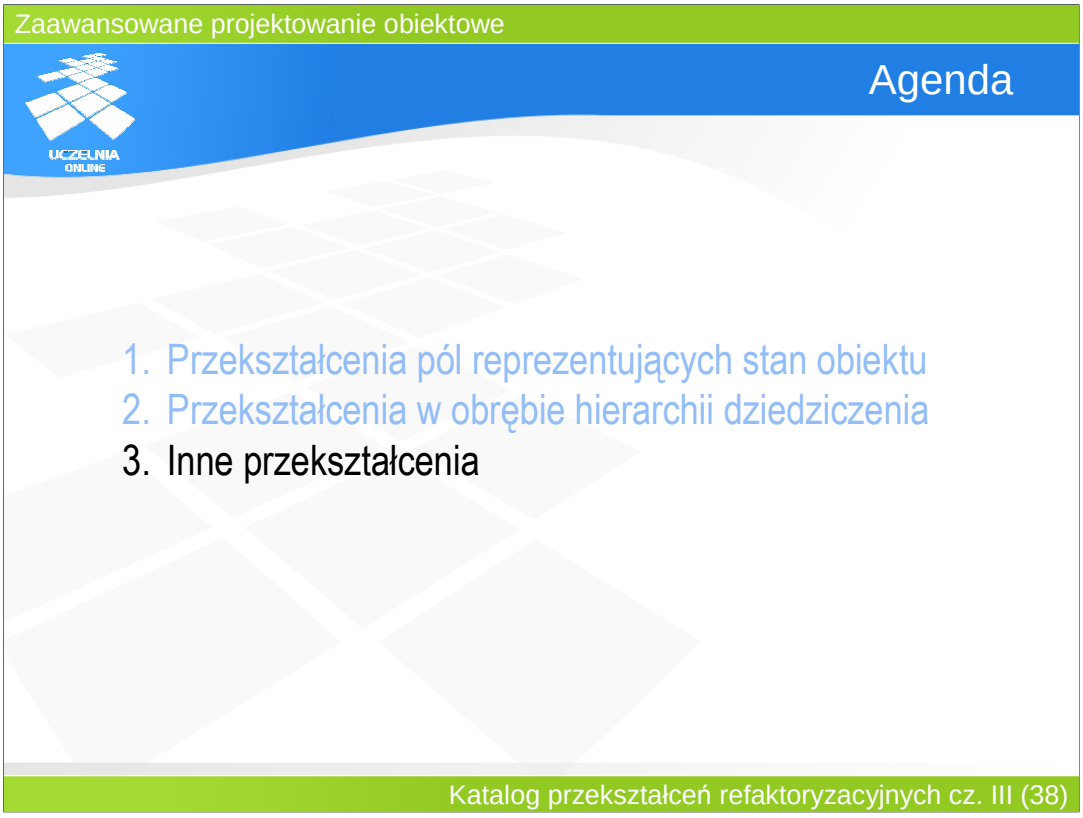
Mechanika

- utwórz pustą podklasę i zdefiniuj w niej konstruktor
- zastąp odpowiednie wywołania konstruktora nadklasy wywołaniami konstruktora podklasy
- przenieś wybrane pola i metody do podklasy
- skompiluj i przetestuj

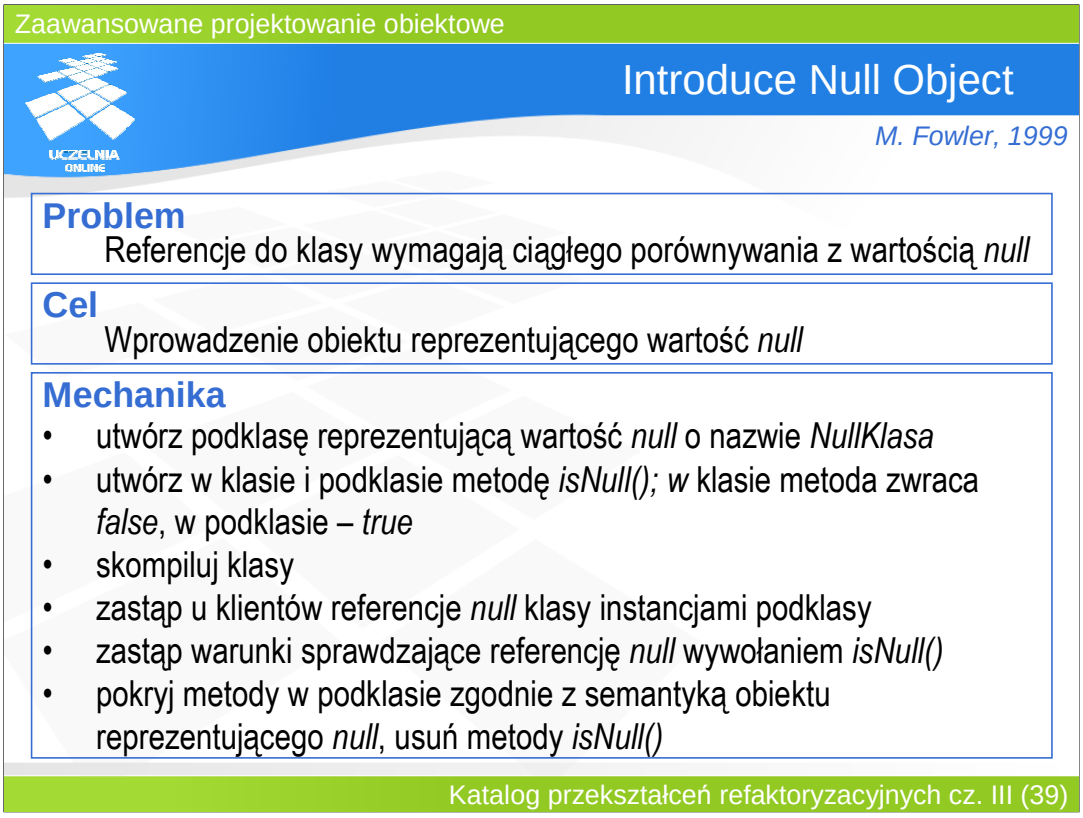
M. Fowler, 1999

Przekształceniem analogicznym do poprzedniego jest wyłączenie podklasy. Stosuje się je w sytuacjach, gdy część metod należących do klasy w niektórych sytuacjach nie jest wykorzystywana. Oznacza to, że klasa w rzeczywistości powinna być podzielona na dwie części – ogólniejszą i bardziej szczegółową, czyli nadklasę i podklasę, połączone relacją dziedziczenia.

Przekształcenie rozpoczyna się od zdefiniowania nowej podklasy i jej konstruktora, który zwykle odwołuje się bezpośrednio do konstruktora nadklasy. Następnie należy podzielić instancje klasy utworzone przez klientów na te, które wymagają pełnego zbioru metod, i wymagające jedynie części z nich. Te pierwsze powinny być zastąpione przez instancje podklasy (jednak na tym etapie różnica ta nie odgrywa żadnej roli, ponieważ obie klasy posiadają identyczny zbiór metod). Ostatnim krokiem jest przeniesienie metod i pól wymaganych jedynie przez niektóre instancje do podklasy (stosując przekształcenie Push Down Field/Method).



Ostatnia grupa przekształceń obejmuje pozostałe refaktoryzacje, które trudno zakwalifikować do pozostałych kategorii.

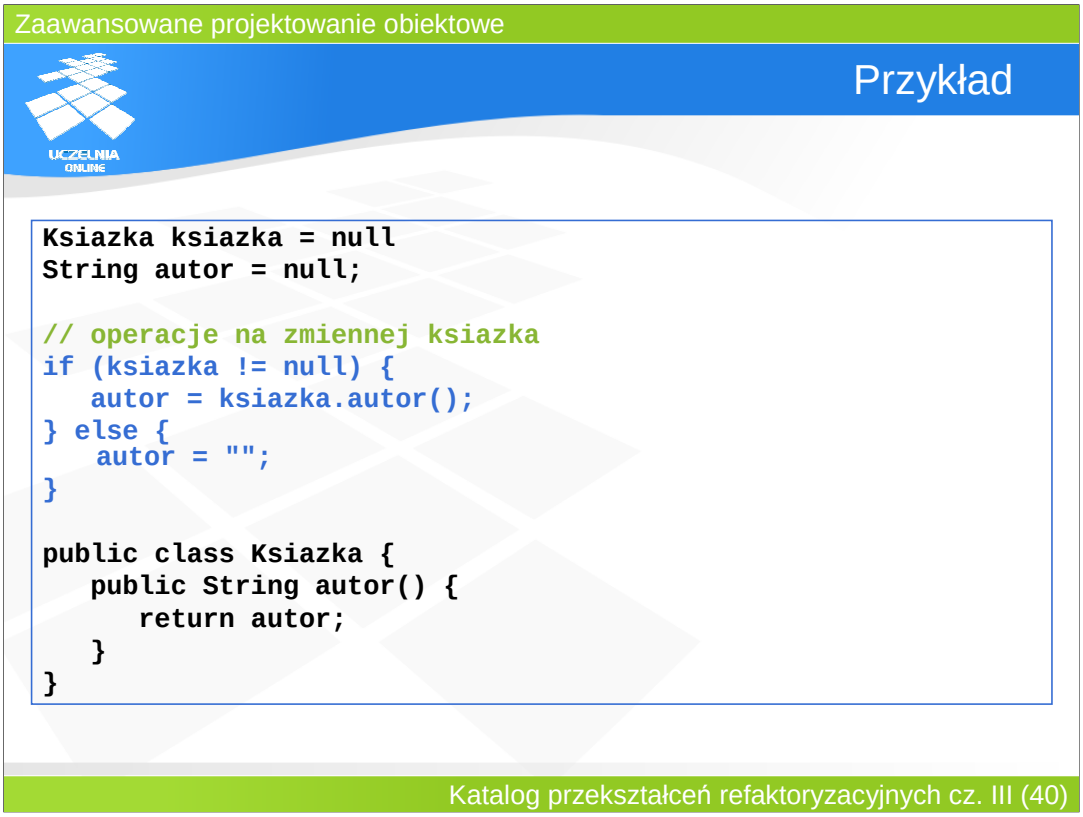


Przekształcenie to służy do usunięcia porównań referencji do obiektu z wartością *null*. Porównania takie często pojawiają się w kodzie, ponieważ zapobiegają pojawieniu się wyjątku *NullPointerException*. Jednak duża ich liczba powoduje ograniczenie czytelności kodu.

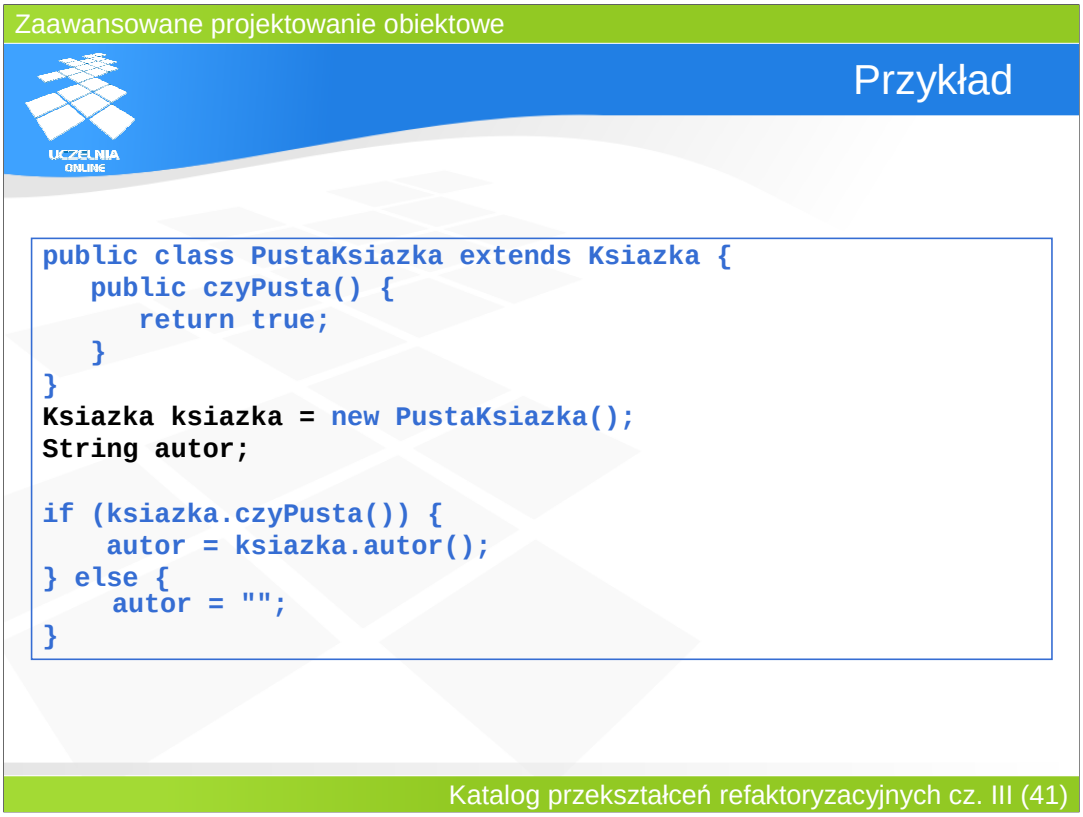
Celem tego przekształcenia jest wykorzystanie polimorfizmu (podobnie jak w przypadku innych przekształceń dotyczących wyrażeń warunkowych) i stworzenie podklasy analizowanego obiektu reprezentującej wartość *null*.

Przekształcenie zaczyna się od stworzenia podklasy, której nazwa – zgodnie z konwencją – zaczyna się od słowa *Null* i która będzie reprezentowała niezainicjowane zmienne analizowanej klasy. W obu klasach (nadklasie i podklasie) należy zdefiniować metodę, która pozwoli odróżnić ich instancje – np. *isNull()*. W podklasie metoda ta zwraca wartość *true*, a w nadklasie – *false.* Następnie wszystkie przypisania wartości *null* należy zastąpić utworzeniem instancji podklasy. Dzięki temu można po kolei zmieniać występujące u klientów porównania z wartością *null* wywołaniem metody *isNull().*

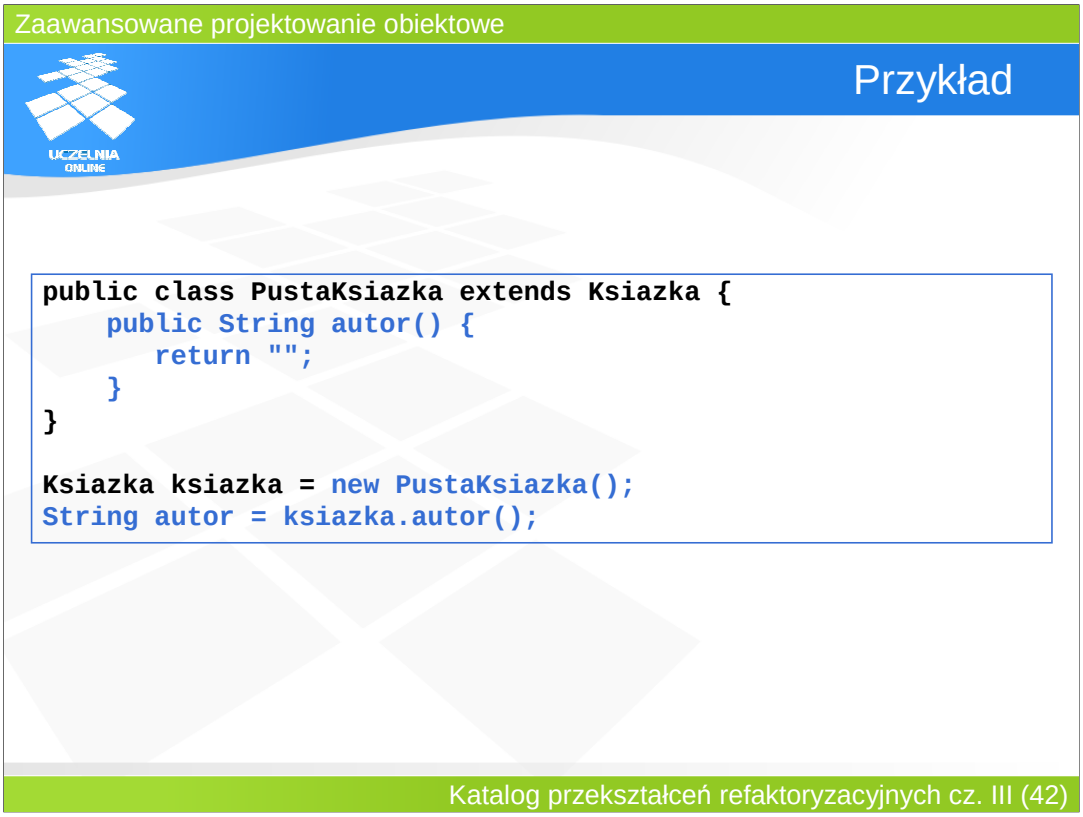
Dopiero po zakończeniu tej operacji jest możliwe pełne wykorzystanie polimorfizmu: należy pokryć w podklasie metody zwracające wartości odziedziczone po nadklasie, tak aby ich wyniki odpowiadały tym, których można się spodziewać po obiektach "pustych" (np. nazwisko nieistniejącego studenta może być napisem pustym). To pozwoli usunąć metody *isNull()* i warunkowe ich sprawdzanie, ponieważ wynik pokrytych metod wywoływanych na obiektach będzie zależał od klasy, do której należą.



W klasie Książka jest zdefiniowana metoda *autor()*. Jednak przed jej wywołaniem należy się upewnić, czy ma ona wartość różną od *null*, gdyż pozwala to uniknąć zgłoszenia wyjątku. Celem przekształcenia jest usunięcie instrukcji warunkowej, która poprzedza wywołanie dowolnej metody w klasie Książka.



Pierwszy krok przekształcenia polega na utworzeniu podklasy PustaKsiążka i zdefiniowaniu w niej oraz w nadklasie metody *czyPusta()*. W podklasie zwraca ona wartość *true*, natomiast w nadklasie – *false.* Wszystkie przypisania do zmiennych typu Książka, które dotychczas przyjmowały wartość *null*, muszą zostać zastąpione instancjami podklasy PustaKsiążka. Ponieważ obecnie żadna zmienna nie może posiadać wartości *null*, dlatego instrukcje warunkowe w klasach klienckich, sprawdzające czy referencja ma taką wartość, muszą korzystać z metody *czyPusta()*.



Ostatnim etapem jest pokrycie w podklasie metod dziedziczonych po nadklasie: metoda *autor(),* która w nadklasie zwraca nazwisko autora, w podklasie zwraca wartość *"".* Dzięki temu można całkowicie usunąć wywołania metody *czyPusta()* oraz samą metodę, ponieważ wybór właściwej metody *autor()* jest określony przez instancję klasy, na rzecz której metoda ta jest wywoływana.

Efektem przekształcenia jest stworzenie nowej podklasy, która reprezentuje obiekt "pusty", czyli równoważny referencji *null*. Należy jednak pamiętać, że obiekty takie mogą zachowywać się różnie w różnych kontekstach (np. metoda *autor()* w niektórych przypadkach może zwracać wartość "(nieznany)"), co wymaga utworzenia wielu takich klas. W takiej sytuacji przekształcenie to może okazać się nieefektywne.

Replace Constructor with Factory Method

Problem

Konstruktor nie pozwala na wybór sposobu utworzenia obiektu

Cel

Zastąpienie bezpośredniego wywołania konstruktora metodą-fabryką

Mechanika

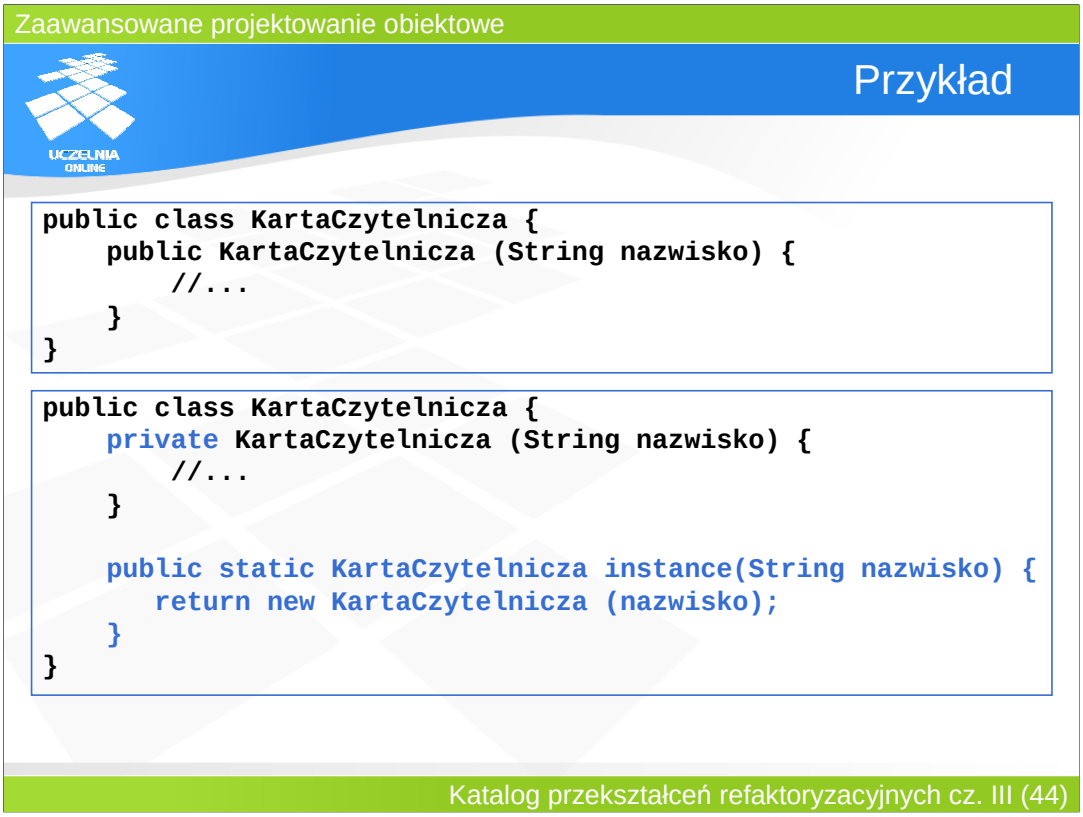
- utwórz metodę-fabrykę (zob. wzorzec projektowy Factory Method) wywołującą konstruktor
- zamień wywołania konstruktora na wywołania metody-fabryki
- skompiluj
- oznacz konstruktor jako prywatny
- skompiluj

M. Fowler, 1999

Przekształcenie to pozwala zmienić sposób tworzenia obiektu pewnej klasy (i jej podklas) z wywołania konstruktora na użycie wzorca Factory Method. Dzięki niemu możliwe jest stworzenie logicznego konstruktora – metody, która, w odróżnieniu od "zwykłego" konstruktora może zachowywać się polimorficznie, a także dokonywać wyboru konkretnej klasy (wśród klas posiadających ten sam typ, np. dziedziczonych). Zwykły konstruktor w momencie wywołania operuje na utworzonym już obiekcie własnej klasy, dlatego takiej możliwości nie posiada. Zatem celem przekształcenia jest zastąpienie konstruktora jako metody tworzenia obiektów przez klientów – wywołaniem metody-fabryki.

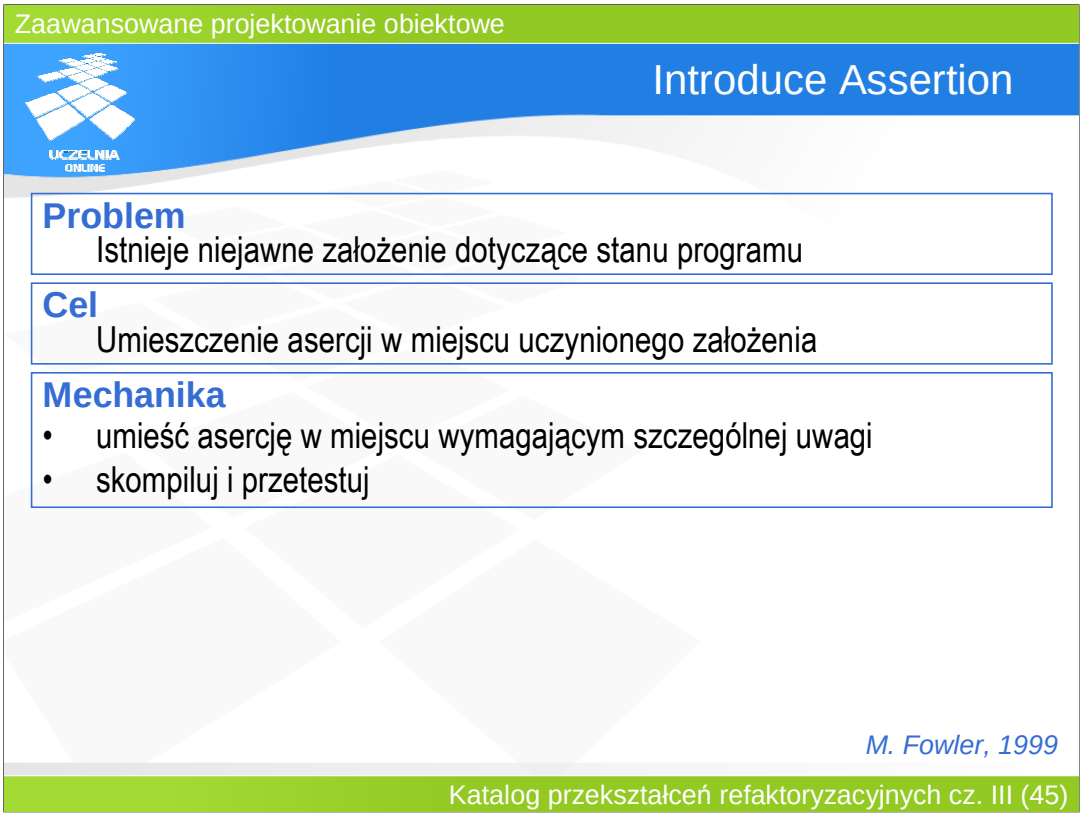
Przekształcenie składa się z trzech kroków. Pierwszym jest stworzenie metody-fabryki (czyli zwykle metody statycznej przyjmującej parametry pozwalające dokonać wyboru klasy, i które można przekazać konstruktorowi), która jedynie wywołuje konstruktor. Następnie wywołania konstruktora w kodzie klientów są zastępowane wywołaniami metody-fabryki. Aby uniemożliwić tworzenie obiektów bez pośrednictwa metody-fabryki, należy zadeklarować konstruktor jako prywatny.

Po wykonaniu przekształcenia można rozwijać logikę metody-fabryki: wyposażyć ją w pamięć utworzonych obiektów lub dokonywać w niej wyboru klasy tworzonego obiektu.



W tym przykładzie przekształceniu ulega obiekt KartaCzytelnicza. W miejsce publicznego konstruktora tworzona jest metoda-fabryka, która wywołuje ten sam konstruktor, ale już zadeklarowany jako prywatny.

Dzięki temu przekształceniu tworzenie instancji klasy KartaCzytelnicza stało się procesem bardziej abstrakcyjnym, niepowiązanym z konkretną klasą. Pozwala to na dalszą rozbudowę i ewolucję metody-fabryki.



Wprowadzenie asercji do kodu programu właściwie nie stanowi refaktoryzacji, ponieważ w wyniku tego przekształcenia nie zmienia się struktura kodu, natomiast może zmienić się jego zachowanie (jeżeli asercja okaże się fałszywa). Jednak warto wspomnieć o tej operacji, ponieważ pozwala ona poprawić czytelność i zrozumienie kodu poprzez uwypuklenie pewnych poczynionych niejawnie założeń.

Introduce Foreign Method

Problem

Klasa wymaga nowej metody, jednak nie może być uzupełniona

Cel

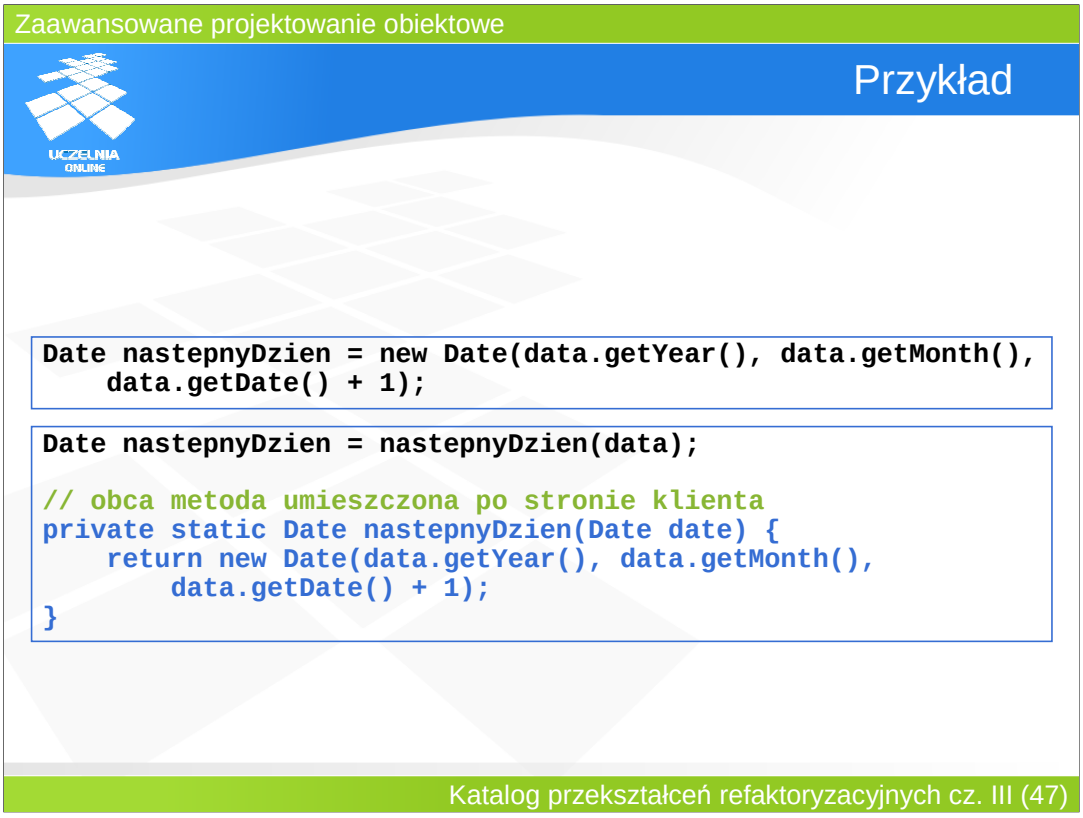
Wprowadzenie metody po stronie klienta

Mechanika

- utwórz nową metodę w klasie klienta, przekazując jej referencję do klasy serwera i inne niezbędne dane
- skomentuj metodę jako 'obcą'
- skompiluj

M. Fowler, 1999

Kolejne dwa przekształcenia dotyczą problemu związanego z niewystarczającą funkcjonalnością oferowaną przez importowaną bibliotekę, której nie można bezpośrednio uzupełnić. Ten przypadek opisuje sytuację, w której w bibliotece brakuje zaledwie kilku metod. Wówczas sugerowanym rozwiązaniem jest umieszczenie ich w kodzie klienta, chociaż obniża to spójność kodu. Sytuacja taka wymaga komentarza umieszczonego przy każdej metodzie, który uzasadni umiejscowienie jej właśnie w tej klasie.



Przykład autorstwa M. Fowlera dotyczy klasy Date, w której brakuje metody obliczającej następny dzień. Aby uniknąć obliczania tej wartości ręcznie, można stworzyć metodę *nastepnyDzien()* i umieścić ją w pobliżu miejsca jej wywołania.

Introduce Local Extension

Problem

Klasa wymaga kilku nowych metod, jednak nie może być uzupełniona

Cel

Utworzenie rozszerzenia (podklasy lub opakowania)

Mechanika

- utwórz nową klasę będącą rozszerzeniem (podklasą lub opakowaniem) klasy serwera
- utwórz konstruktor konwertujący w klasie rozszerzenia
- dodaj nowe funkcje w klasie rozszerzenia
- zmień wybranych klientów, tak aby korzystali z klasy rozszerzenia

M. Fowler, 1999

Innym rozwiązaniem problemu niekompletnej biblioteki jest utworzenie jej rozszerzenia w postaci podklasy lub opakowania. Dzięki temu możliwe jest (w przeciwieństwie do poprzedniego przekształcenia) umieszczenie metod we właściwym miejscu. Wybór pomiędzy dziedziczeniem a delegacją często podyktowany jest możliwościami zastosowania jednej albo drugiej techniki: dziedziczenie wymaga, aby klasa biblioteczna nie była sfinalizowana, natomiast delegacja – publicznego dostępu do najważniejszych metod.

Mechanika przekształcenia polega na utworzeniu klasy-rozszerzenia, zdefiniowaniu w niej konstruktora (w przypadku delegacji musi to być konstruktor konwertujący – zob. wzorzec Decorator) i dodaniu nowych funkcji. Ostatnim krokiem jest aktualizacja klientów, którzy wymagają dostępu do rozszerzonej klasy.

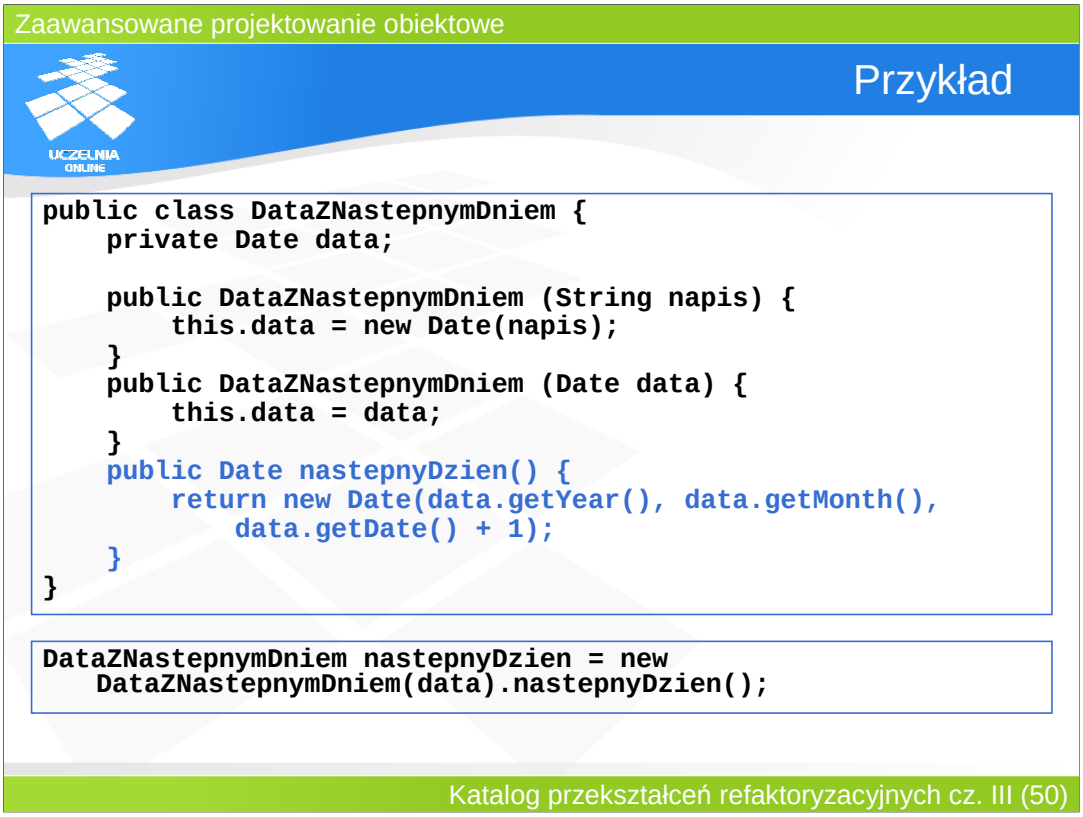
Przykład

```
Date nastepnyDzien = new Date(data.getYear(), data.getMonth(),
    data.getDate() + 1);
```

```
public class DataZNastepnymDniem extend Date {
    public DataZNastepnymDniem (String napis) {
        super(napis);
    }

    public DataZNastepnymDniem (Date data) {
        super(data.getTime());
    }

    public Date nastepnyDzien() {
        return new Date(getYear(), getMonth(), getDate() + 1);
    }
}
```

```
Date nastepnyDzien = new
   DataZNastepnymDniem(data).nastepnyDzien();
```


Slajd ten przedstawia ten sam przykład co poprzednio, jednak zaimplementowany w postaci lokalnego rozszerzenia. Jest nim klasa DataZNastępnymDniem, będąca podklasą klasy Date. Definiuje ona metodę *nastepnyDzien(),* która realizuje wymaganą funkcjonalność. Wywołanie tej metody wymaga utworzenia obiektu klasy-rozszerzenia.



Drugie rozwiązanie polega na stworzeniu klasy-opakowania, która deleguje wywołania metod do pola będącego obiektem klasy Date. Należy jedynie pamiętać, że w tej postaci (o ile nie istnieje interfejs, który implementuje klasa źródłowa) rozszerzenie jest niezgodne pod względem typu z klasą źródłową. Oznacza to, że nie może być do niego stosowana reguła podstawiania, np. nie może być użyty jako argument zamiast klasy źródłowej.

Alternatywnym rozwiązaniem jest zastosowanie w roli opakowania podklasy klasy źródłowej. Wówczas problem niezgodności typów jest rozwiązany.

Replace Exception with Test

Problem

Wyjątek jest zgłaszany w sytuacji, gdy można dokonać sprawdzenia

Cel

Zastąpienie zgłaszania wyjątku sprawdzeniem

Mechanika

- umieść warunek poprawności na początku metody
- skopiuj ciało klauzuli *catch* do klauzuli tego warunku
- umieść asercję z warunkiem poprawności w klauzuli *catch*
- skompiluj i przetestuj
- usuń klauzule *try* i *catch*
- skompiluj

M. Fowler, 1999

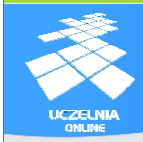
Motywacja dla tego przekształcenia wynika z obserwacji, że wielu sytuacji wyjątkowych w systemie można uniknąć, jeżeli możliwe byłoby wykluczenie okoliczności, które je powodują. Celem jest zatem zastąpienie zgłoszenia wyjątku sprawdzeniem, czy może on się pojawić.

Pierwszym krokiem przekształcenia jest umieszczenie warunku sprawdzającego warunki wystąpienia wyjątku na początku metody, i skopiowanie ciała klauzuli catch obsługi wyjątku jako akcji wykonywanej jeżeli ten warunek jest spełniony. W celu zabezpieczenia przed wykonaniem klauzuli *catch* (po przekształceniu wyjątek nie powinien już się pojawić, zatem jego obsługa również jest zbędna) można umieścić w niej asercję dotyczącą warunku poprawności. Ostatnim krokiem jest usunięcie instrukcji obsługi wyjątku i klauzul *try* oraz *catch*.

Przykład

```
Stack<Connection> pula;
Stack<Connection> zaalokowane;

public Connection polaczenie() {
    Connection polaczenie;
    try {
        polaczenie = pula.pop();
    } catch (EmptyStackException ex) {
        polaczenie = new DriverManager.getConnection(
            url, uzytkownik, haslo);
    }
    zaalokowane.push(polaczenie);

    return polaczenie;
}
```


Jako przykład rozpatrzmy fragment klasy obsługującej pulę połączeń z bazą danych. Metoda *polaczenie()* próbuje pobrać połączenie z dostępnej puli. Jeżeli pula jest pusta i zostanie zgłoszony wyjątek, wówczas metoda tworzy nowe połączenie i umieszcza je w puli połączeń zaalokowanych.



Przykład

```
Stack<Connection> pula;
Stack<Connection> zaalokowane;

public Connection polaczenie() {
    Connection polaczenie;
    if (pula.isEmpty()) {
        polaczenie = new DriverManager.getConnection(
            url, uzytkownik, haslo);
    } else {
        polaczenie = pula.pop();
    }
    zaalokowane.push(polaczenie);

    return polaczenie;
}
```


Warunkiem powodującym zgłoszenie wyjątku jest *pula.isEmpty()*, dlatego to właśnie on jest umieszczony na początku metody. Jeżeli warunek ten jest spełniony, wówczas wykonywana jest akcja realizowana obecnie w klazuli *catch* obsługi wyjątku, a w przeciwnym przypadku – instrukcja pobrania połączenia pochodząca z klauzuli *try*.

Taka zmiana pozwoliła na zapobieżenie wyjątkowi zamiast podjęcia próby jego obsłużenia.

Replace Recursion with Iteration

Problem

Rekrusja jest trudno zrozumiała

Cel

Przekształcenie rekursji w iteracje

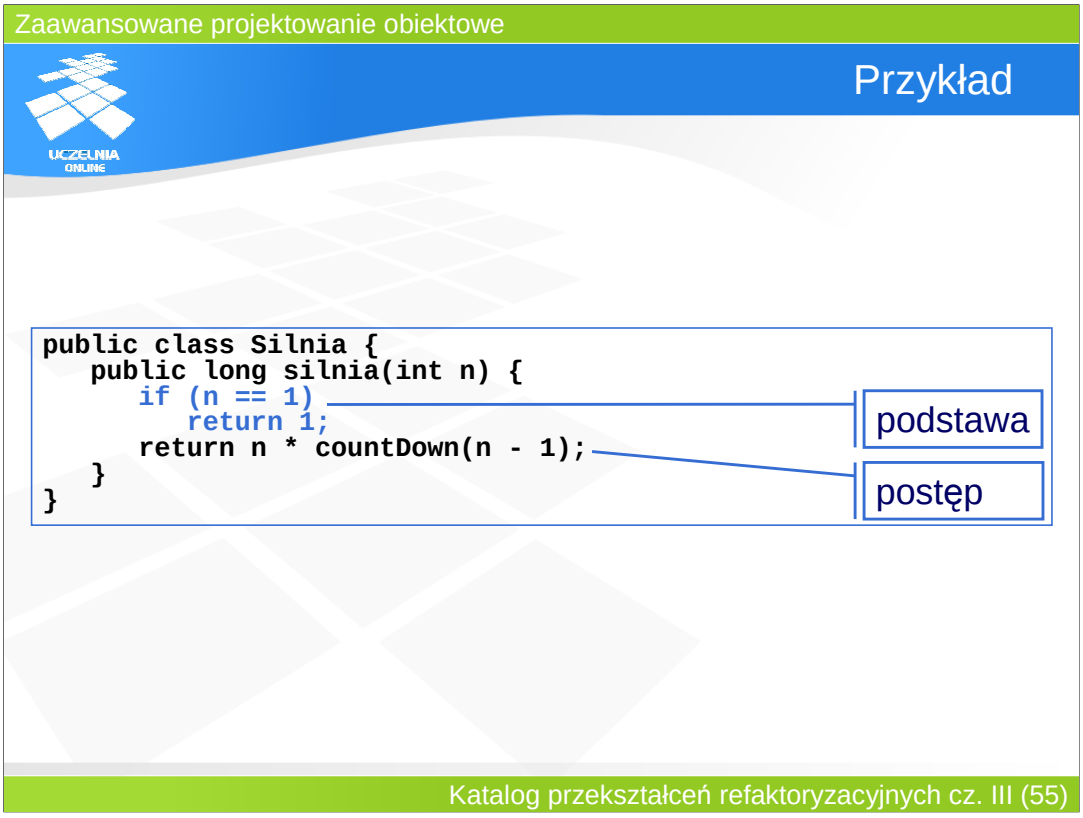
Mechanika

- wyznacz wartość bazową rekursji
- stwórz pętlę, która będzie kończyła się przy wartości bazowej
- wykonuj wewnątrz pętli postęp w kierunku wartości bazowej
- przekaz zmodyfikowane parametry na początek pętli

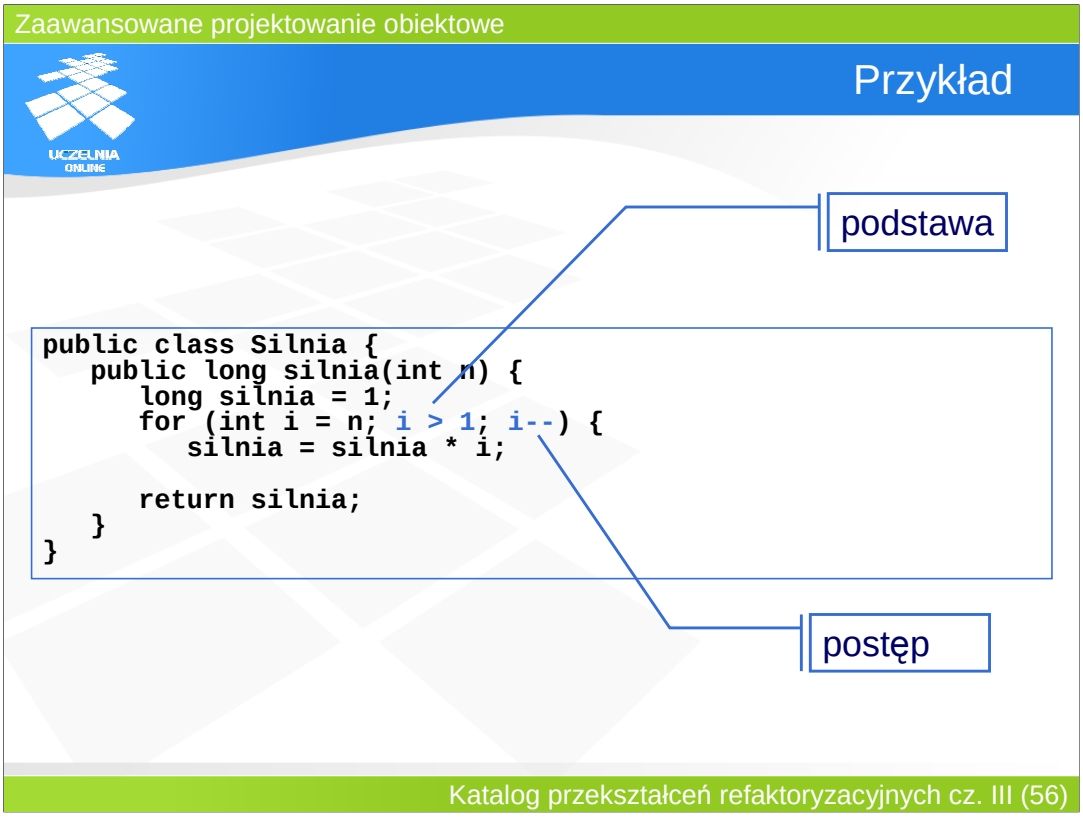
I. Mitrovic, 2001

Dwa ostatnie przekształcenia dotyczą problemu przekształcania rekurencji w iteracje i odwrotnie.

Zastąpienie rekurencji przez iteracje rozpoczyna się od wyznaczenia wartości bazowej rekurencji, tzn. wartości, przy której metoda nie wywołuje siebie rekurencyjnie. Należy wówczas stworzyć pętlę, która przestanie się wykonywać w momencie osiągnięcia wartości bazowej, oraz która będzie wykonywała postęp w kierunku tej wartości, zgodnie z parametrami przekazywanymi dotychczas przy rekurencyjnym wywołaniu metody. Zmodyfikowane parametry rekurencji są obecnie przekazywane na początek pętli.



Najprostszym przykładem rekurencyjnej metody jest obliczanie silni. W obecnej postaci podstawą jest warunek n==1, natomiast postęp polega na odejmowaniu 1 od n.



Po przekształceniu obliczenia zostały umieszczone w pętli *for*. Podstawa jest zapisana w postaci i > 1, a postęp – i--;

Replace Iteration with Recursion

I. Mitrovic, 2001

Problem

Trudno zrozumień semantykę pętli

Cel

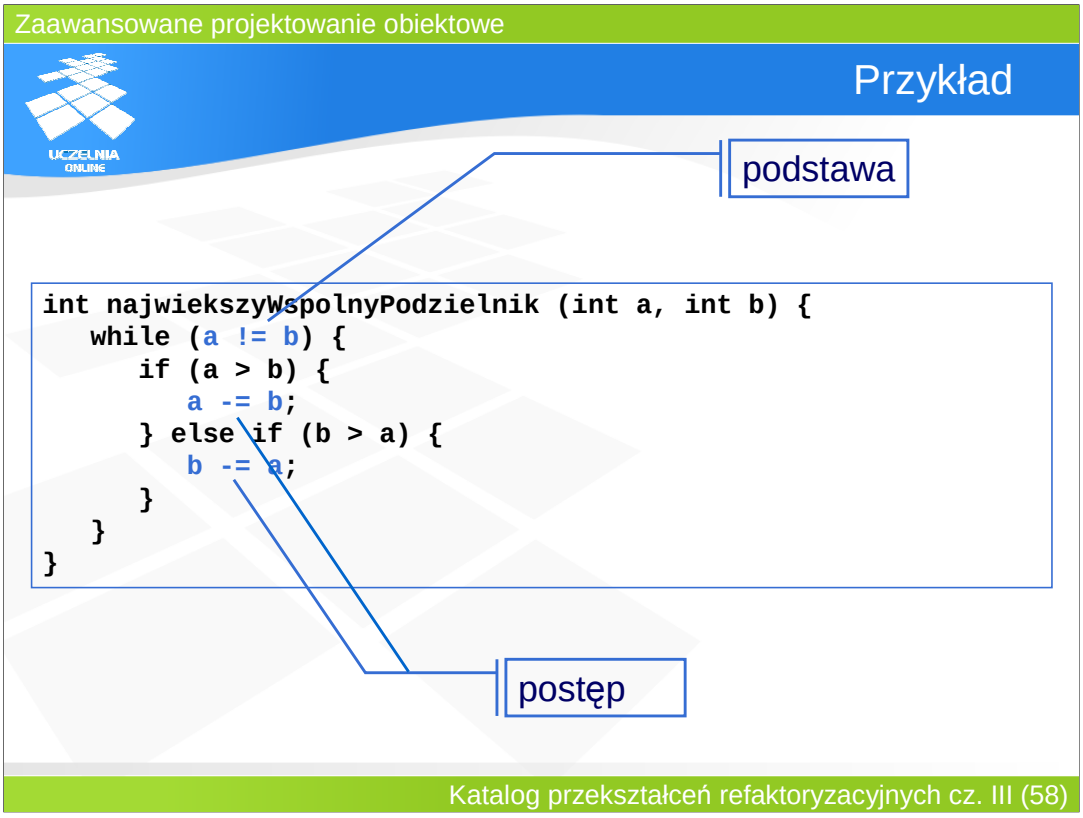
Przekształcenie iteracji w rekursję

Mechanika

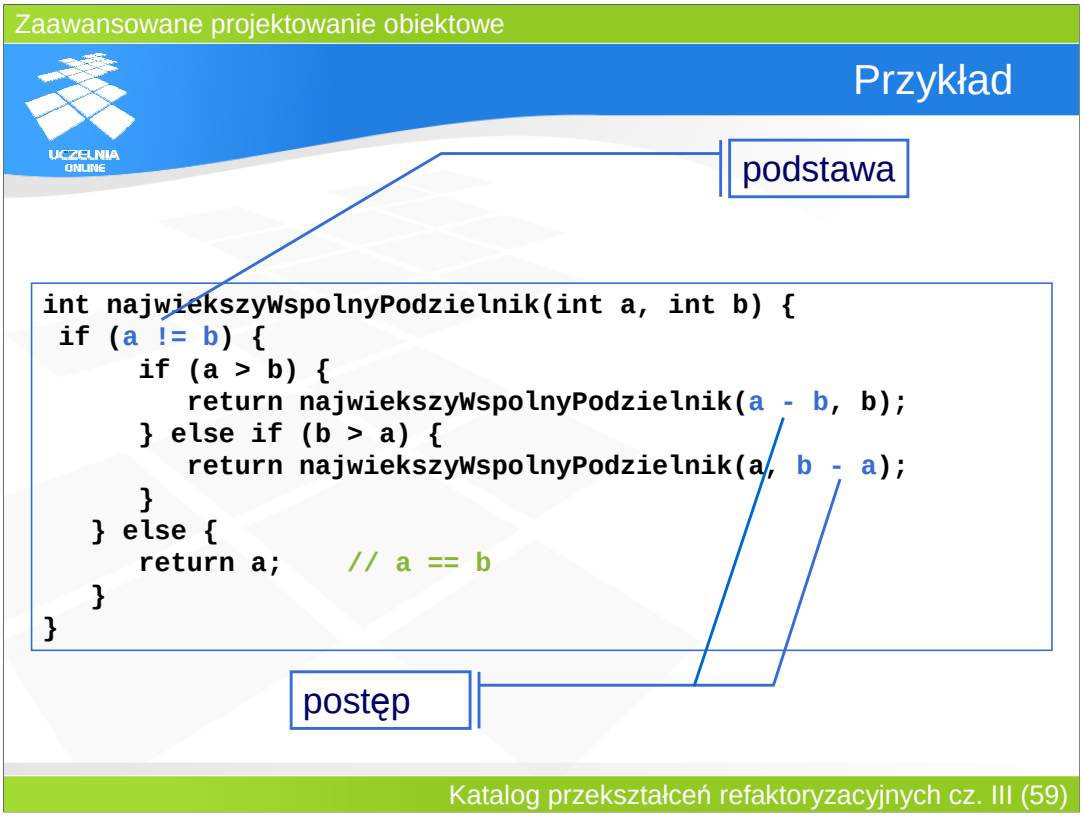
- znajdź pętlę modyfikującą jedną lub kilka zmiennych lokalnych, które potem wpływają na wynik metody; wyłącz pętlę do nowej metody
- zastąp pętlę algorytmem przyjmującym te same parametry i dającym ten sam wynik:
  - klauzula *if* sprawdza warunek kontynuacji pętli
  - akcja *then* zwraca wynik końcowy
  - akcja *else* wywołuje metodę rekurencyjnie ze zmienionymi param.
- skompiluj i przetestuj

Przekształcenie odwrotne jest wykonywane w nielicznych przypadkach, w których zapis iteracyjny jest trudniej zrozumiały niż rekurencyjny. Celem jest zatem zamiana iteracji w rekurencję.

Pierwszym krokiem przekształcenia jest identyfikacja głównej pętli, wewnątrz której modyfikowana jest jedna lub kilka zmiennych lokalnych wpływających na wynik metody. Warto wówczas rozważyć wyłączenie pętli do odrębnej metody. Następnie pętla jest zastępowana wyrażeniem warunkowym, które zachowuje się identycznie jak pętla i sprawdza warunek kontynuacji pętli. Jeżeli warunek ten jest spełniony, wówczas zwracany jest wynik końcowy metody, natomiast w przeciwnym przypadku wywoływana jest rekurencyjnie ta sama metoda z parametrami zmienionymi zgodnie z postępem pętli.



Przykład podał autor przekształcenia, I. Mitrovic. Metoda *największyWspólnyPodzielnik()* jest realizowana w postaci iteracyjnej. Podstawą pętli jest warunek a != b, natomiast postępem są dwie alternatywne instrukcje a-=b i b-=a;



W wyniku przekształcenia niespełnienie warunku kontynuacji pętli (a != b) spowoduje zwrócenie wyniku metody. W przeciwnym przypadku nastąpi rekurencyjne wywołanie tej metody, która przekazuje zmodyfikowane parametry zgodnie z zapisem jaki istniał wewnątrz pętli.

Podsumowanie

- Przekształcenie refaktoryzacyjne, podobnie jak wzorzec projektowy, jest opisane szablonem
- Katalog Fowlera zawiera praktyczne przekształcenia refaktoryzacyjne dla języka Java
- Obejmują one najważniejsze elementy programowania w tym języku
- Katalog jest sukcesywnie rozszerzany przez innych autorów

Podczas trzech ostatnich wykładów przedstawiono katalog przekształceń refaktoryzacyjnych, oparty przede wszystkim na katalogu autorstwa M. Fowlera. Każde przekształcenie jest opisane atrybutami nadającymi mu nadać postać zbliżoną do wzorca projektowego. Przekształcenia te obejmują najważniejsze mechanizmy języka oraz problemy projektowe, dlatego warto z nich korzystać w praktyce programowania.