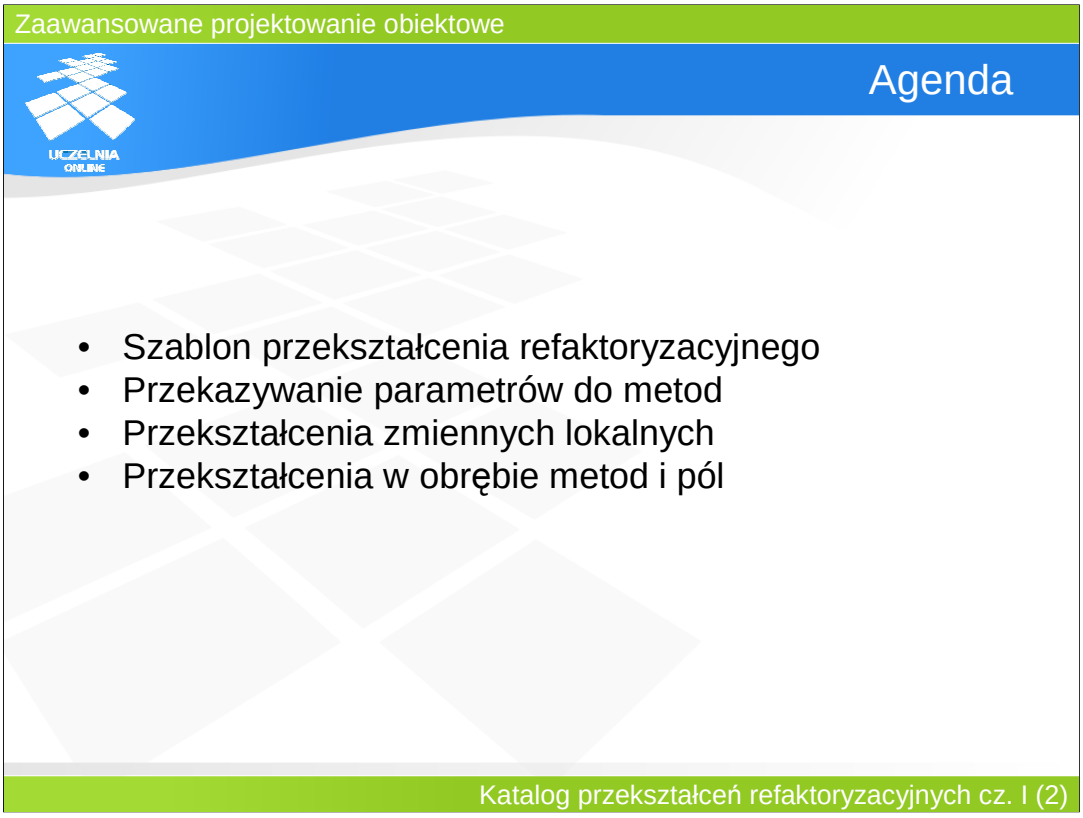
Zaawansowane projektowanie obiektowe

Katalog przekształceń refaktoryzacyjnych cz. I

Prowadzący: Bartosz Walter

UCZELNIA
ONLINE



Wykład ten jest pierwszym z serii trzech poświęconych prezentacji katalogu przekształceń refaktoryzacyjnych opartego na książce M. Fowlera.

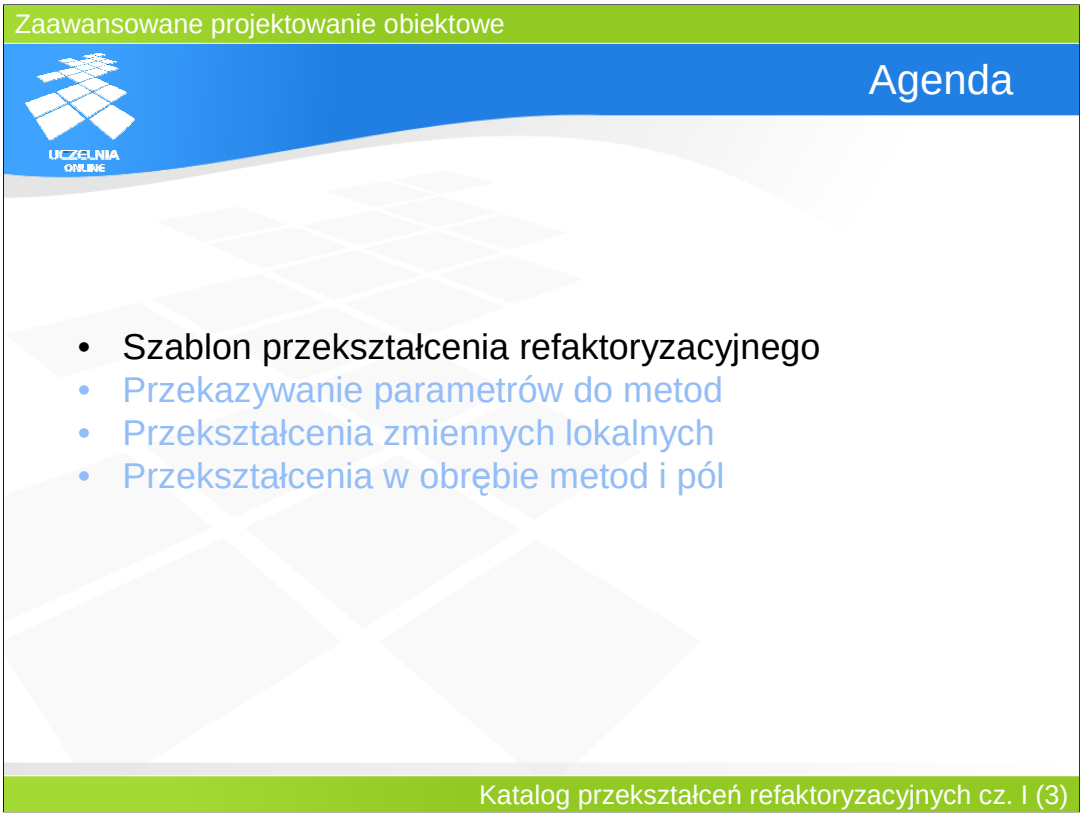
W tej części zaprezentowane zostaną następujące zagadnienia:

•szablon refaktoryzacji, czyli wzorzec, według którego każde przekształcenie jest opisywane;

•grupa przekształceń poświęconych przekazywaniu parametrów do metod,

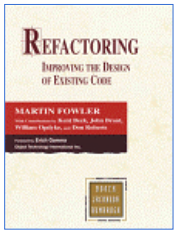
•przekształcenia związane ze zmiennymi lokalnymi i tymczasowymi;

•przekształcenia w obrębie pól i metod w obiekcie.



Pierwsza część wykładu dotyczy opisu przekształcenia refaktoryzacyjnego.

Szablon przekształcenia refaktoryzacyjnego



Problem

Cel

Mechanika

Przykład

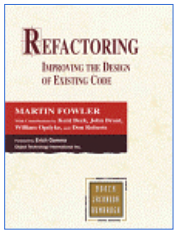
Szablon przekształcenia refaktoryzacyjnego jest odpowiednikiem szablonu wzorca projektowego. Przedstawiony podczas wykładu szablon jest wzorowany na szablonie zastosowanym przez Martina Fowlera w jego katalogu.

Ograniczony szablon stosowany podczas niniejszego wykładu składa się z następujących elementów:

- **problemu**, jaki przekształcenie próbuje rozwiązać;
- **celu**, jaki należy za pomocą przekształcenia osiągnąć;
- **mechaniki**, opisującej kolejne kroki, jakie należy podjąć, aby przekształcenie zostało z powodzeniem i poprawnie zakończone;
- **przykładu**, który przybliża sposób jego przeprowadzenia

Szablon przekształcenia refaktoryzacyjnego

Extract Method



Problem

Metoda wykonuje wiele funkcji

Cel

Wyłączenie części kodu do osobnych metod

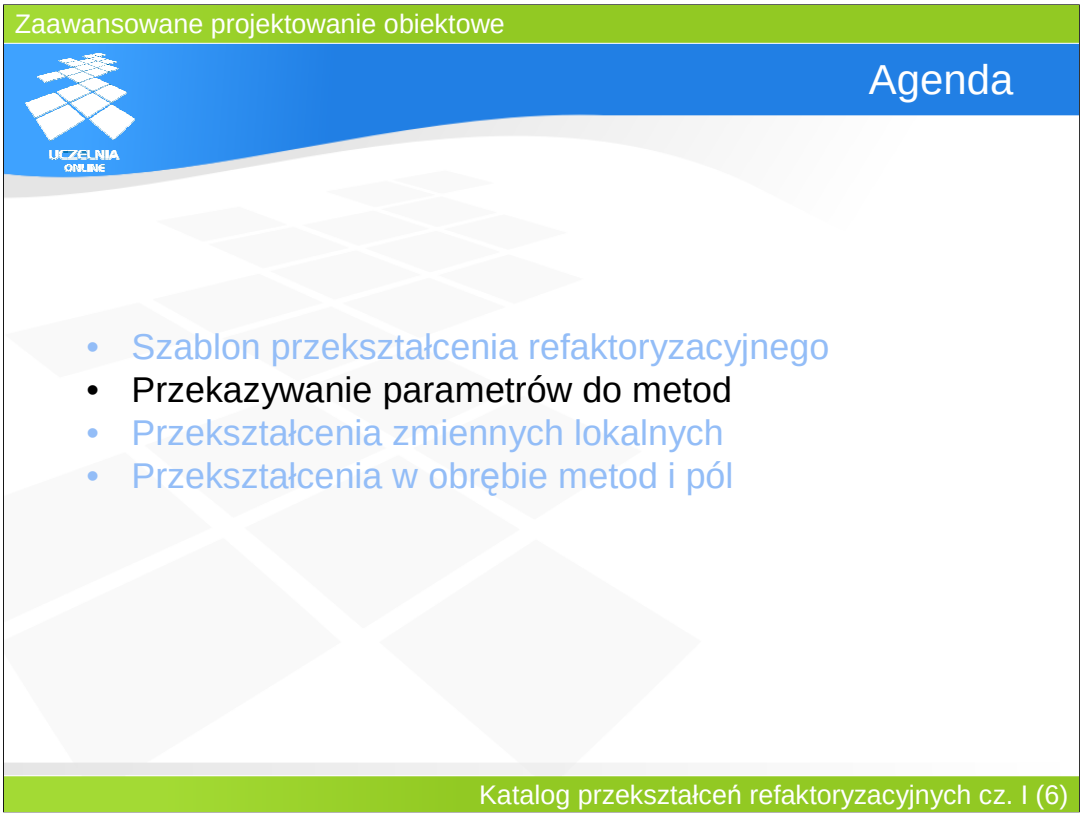
Mechanika

- wybierz fragment kodu
- poszukaj w nim modyfikowanych zmiennych lokalnych
- utwórz nową metodę i skopiuj do niej wybrany fragment kodu
- przekaż zmienne lokalne do nowej metody jako parametry
- zastąp stary kod wywołaniem nowej metody
- skompiluj i przetestuj

Przykład

Na slajdzie przedstawiony został oparty na szablonie opis przekształcenia refaktoryzacyjnego Extract Method, czyli wyłączenia fragmentu kodu do nowej metody. Dotyczy ono problemu długich, rozbudowanych metod realizujących wiele funkcji. Celem przekształcenia jest wyłączenie części kodu do osobnych, nowoutworzonych metod.

Mechanika polega na wybraniu fragmentu kodu, który ma być wyłączony, następnie – wyszukaniu w nim modyfikowanych zmiennych lokalnych (zmienne takie przenoszą informacje pomiędzy tym fragmentem kodu a szerszym kontekstem, w jakim ten fragment się znajduje) oraz utworzeniu nowej metody, do której wybrany fragment kodu jest kopiowany. Zmienne lokalne, które zostały zidentyfikowane, są przekazywane do nowej metody jako jej parametry. Wówczas stary kod, który na początku został zaznaczony, może zostać zastąpiony wywołaniem nowej metody.



Pierwsza grupa przekształceń dotyczy sposobu przekazywania parametrów do metod, ich dodawania i usuwania.

Add Parameter

Problem

Metoda potrzebuje więcej informacji od klienta

Cel

Dodanie parametru do sygnatury

Mechanika

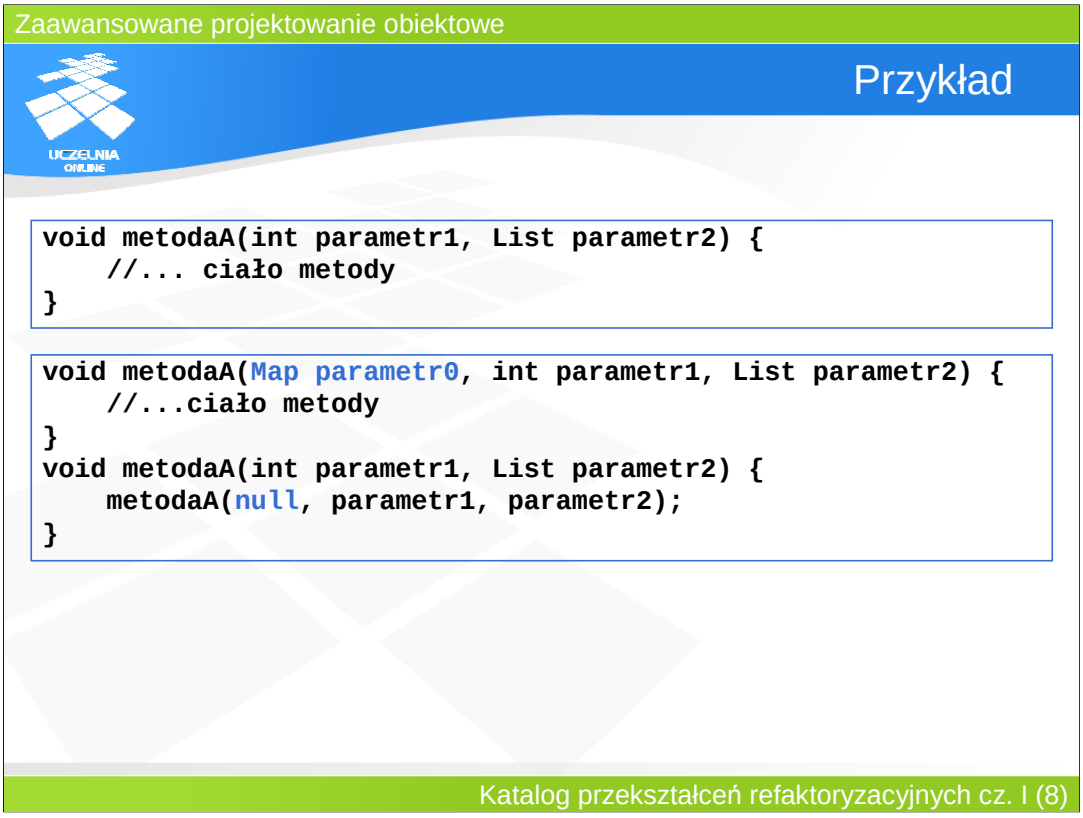
- sprawdź czy metoda nie jest polimorficzna
- zadeklaruj nową metodę z dodanym parametrem
- skopiuj ciało starej metody do nowej
- skompiluj
- zmień starą metodę tak, aby delegowała wywołania do nowej
- skompiluj i przetestuj
- zastąp wywołania starej metody wywołaniami nowej
- opcjonalnie: usuń starą metodę
- skompiluj i przetestuj

M. Fowler, 1999

Przekształcenie Add Parameter dotyczy sytuacji, w której metoda potrzebuje od klienta więcej informacji, niż otrzymuje w tej chwili. Zatem konieczne jest przekazanie jej nowego parametru.

Mechanika tej refaktoryzacji przebiega w taki sposób, aby maksymalnie wydłużyć okres, w którym istnieją jednocześnie dwie wersje metody, dotychczasowa i nowa, z dodanym parametrem. Jest to charakterystyczny sposób postępowania występujący w wielu przekształceniach, pozwala bowiem zapewnić drogę odwrotu w przypadku niepowodzenia.

Pierwszym krokiem (poza sprawdzeniem czy metoda jest polimorficzna, co uniemożliwiłoby prawidłowe zakończenie przekształcenia) jest stworzenie nowej metody z dodanym parametrem, do której skopiować należy ciało starej metody (warto zauważyć, że nowy parametr pozostaje w niej niewykorzystany). Następnie należny zmienić starą metodę w ten sposób, aby delegowała przychodzące wywołania do nowej wersji, przekazując dowolną wartość w miejsce nowego parametru. W kolejnym kroku należy zmodyfikować klientów metody, tak aby odwoływali się do nowej wersji metody. Ostatnim etapem może być usunięcie starej metody, o ile nie jest ona potrzebna z innych względów (np. uczestnictwa w interfejsie).



Jako przykład posłuży *metodaA()*, posiadająca pierwotnie dwa parametry. W trakcie przekształcenia tworzona jest wersja metody z dodatkowym parametrem *parametr0*, do której delegowane są wywołania z oryginalnej metody. W miejsce dodatkowego parametru przekazywana jest wartość *null*.

Remove Parameter

Problem

Parametr jest nieużywany w ciele metody

Cel

Usunięcie parametru z sygnatury metody

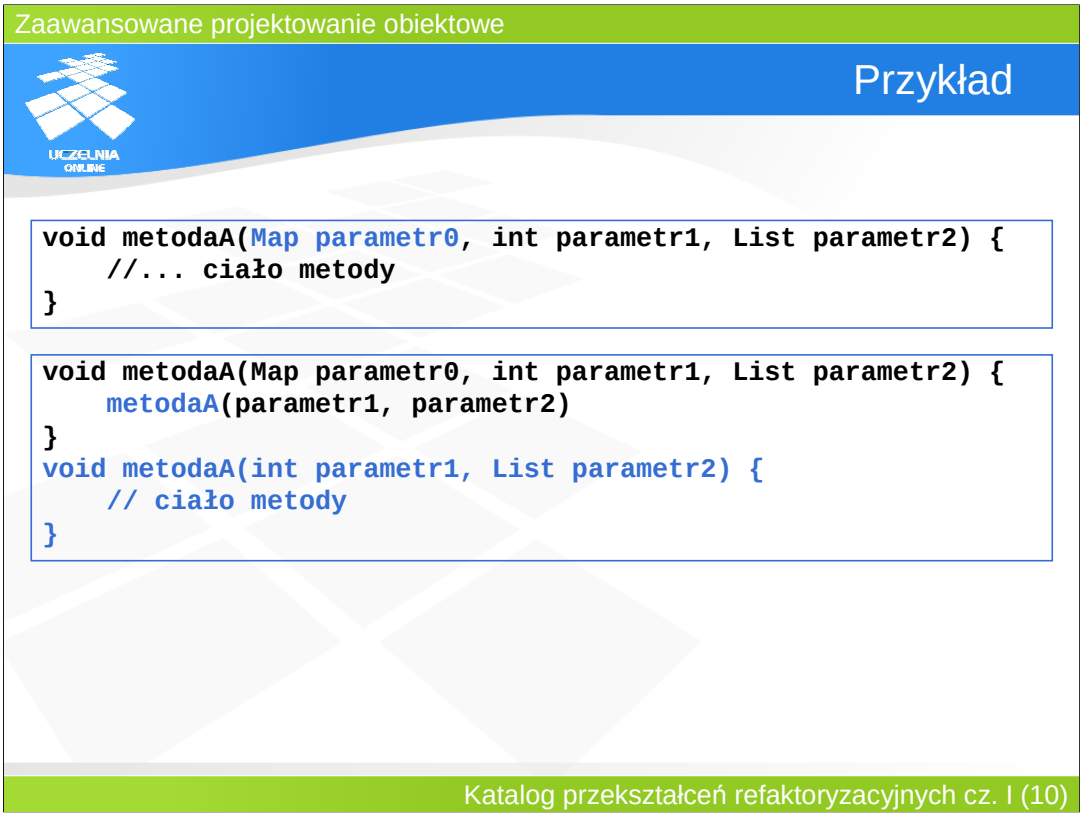
Mechanika

- sprawdź, czy metoda nie jest polimorficzna
- zadeklaruj nową metodę bez zbędnego parametru
- skopiuj ciało starej metody do nowej
- skompiluj
- zmień starą metodę, tak aby delegowała wywołania do nowej
- skompiluj i przetestuj
- zastąp wywołania starej metody wywołaniami nowej
- opcjonalnie: usuń starą metodę
- skompiluj i przetestuj

M. Fowler, 1999

Przekształceniem odwrotnym do poprzedniego jest usunięcie parametru z sygnatury metody. Potrzeba jego wykonania pojawia się, gdy parametr taki jest niewykorzystywany wewnątrz metody.

Mechanika przebiega w sposób odwrotny do dodania parametru. Pierwszym krokiem jest sprawdzenie, czy metoda ta nie jest dziedziczona lub pokrywana w innych klasach. Następnie należy zadeklarować nową metodę bez zbędnego parametru i skopiować do niej ciało oryginalnej metody. Kolejnym krokiem jest umieszczenie delegacji z metody oryginalnej do nowej, i kolejne zastępowanie w ten sam sposób odwołań do tej metody, jakie istnieją po stronie klientów. Na koniec również można usunąć starą metodę.



Przykład polega na usunięciu parametru *parametr0* dodanego w wyniku poprzedniego przekształcenia. Metoda *metodaA()* jest przeciążana, tak że nowa wersja metody jest pozbawiona parametru, a oryginalna wersja deleguje do niej wywołania.

W efekcie wszystkie odwołania do starej metody są usunięte i zastąpione odwołaniami do nowej, bez zbędnego parametru.

Preserve Whole Object

Problem

Pola obiektu są osobno przekazywane jako parametry

Cel

Przekazanie referencji do obiektu zamiast pojedynczych parametrów

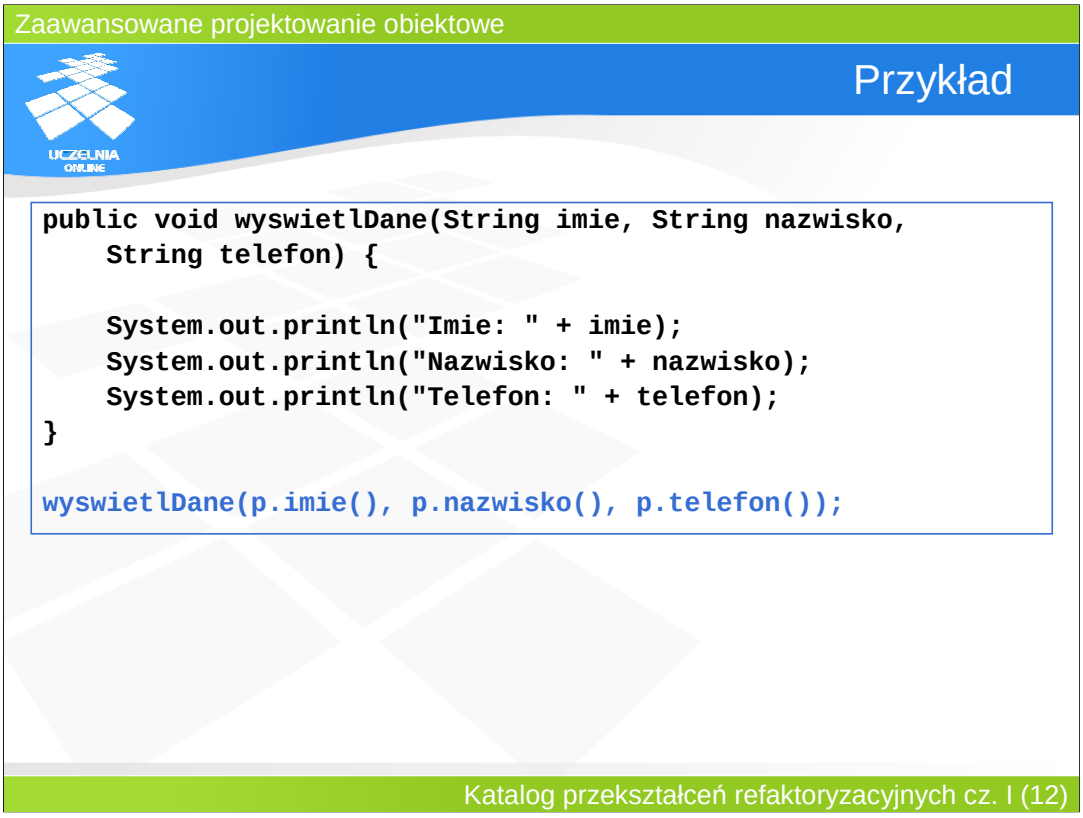
Mechanika

- wprowadź obiekt jako nowy parametr metody
- skompiluj
- zastąp odwołania do kolejnych parametrów metody wywołaniami odpowiednich metod w obiekcie
- usuń parametr
- skompiluj i przetestuj

M. Fowler, 1999

Kolejne trzy przekształcenia służą do ograniczenia liczby parametrów metody. W przypadku tej refaktoryzacji – Preserve Whole Object, czyli zachowaniu całego obiektu – rozwiązywany przez nią problem polega na przekazywaniu do metody jako oddzielnych parametrów pól lub wyników wykonania metod jednego obiektu. Przekształcenie pozwala zastąpić grupę takich parametrów jednym – obiektem będącym ich źródłem.

Refaktoryzacja rozpoczyna się od dodania nowego parametru – referencji do całego obiektu. Następnie odwołania do parametrów występujące wewnątrz metody są kolejno zastępowane wywołaniami metod lub odwołaniami do pól obiektu – nowego parametru. Gdy wszystkie pozostałe parametry staną się bezużyteczne, mogą zostać usunięte z klasy.



Jako przykład rozpatrzmy metodę *wyswietlDane()*, przyjmującą trzy parametry: imię, nazwisko i telefon osoby. W praktyce wywołanie tej metody wymaga pobrania tych trzech wartości z obiektu Osoba, zatem wskazane jest stworzenie wersji metody umożliwiającej przekazanie tylko referencji do tego obiektu.

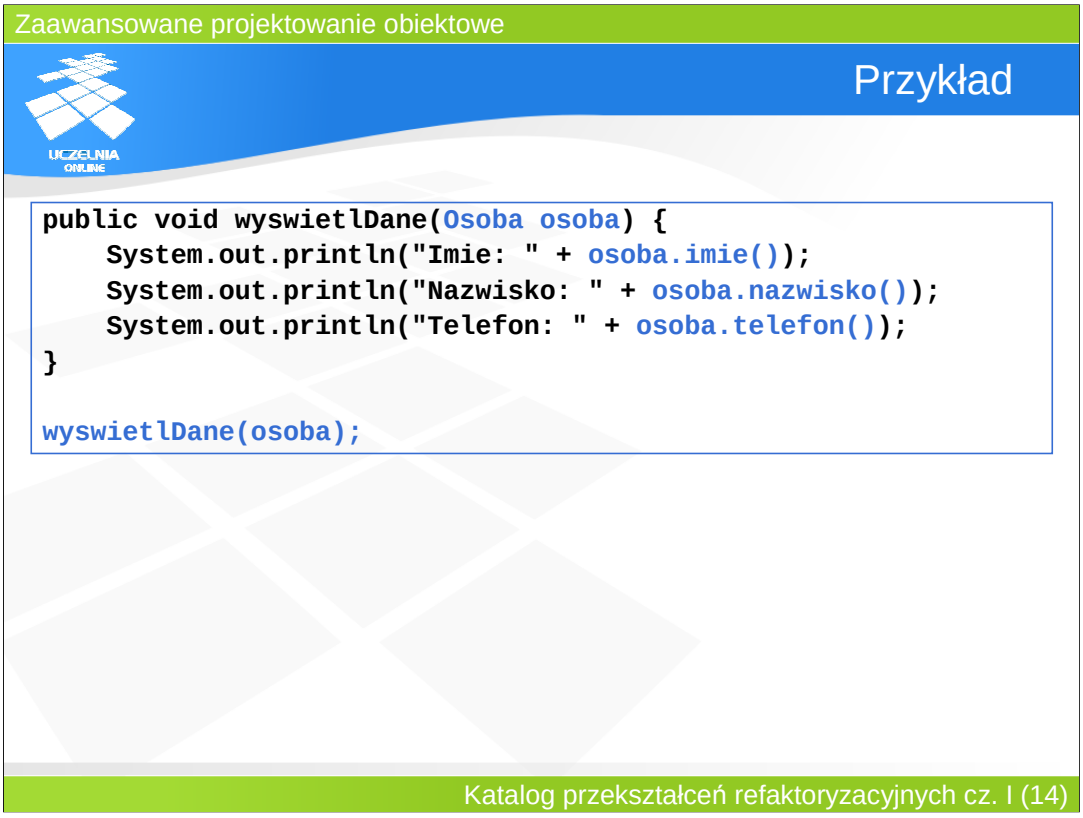
Przykład

```
public void wyswietlDane(Osoba osoba, String imie,
    String nazwisko, String telefon) {

    System.out.println("Imie: " + osoba.imie());
    System.out.println("Nazwisko: " + osoba.nazwisko());
    System.out.println("Telefon: " + osoba.telefon());
}

wyswietlDane(person, null, null, null);
```


Pierwszym krokiem jest dodanie do sygnatury metody nowego parametru typu Osoba, a następnie zmiana odwołań do pozostałych parametrów metody wywołaniami odpowiednich metod nowego parametru. Po zakończeniu zmiany wywołanie metody wymaga przekazania instancji klasy Osoba oraz nieznaczących pozostałych wartości parametrów (w tym przypadku wartości *null*).



Ostatnim krokiem jest usunięcie zbędnych parametrów, zgodnie z mechaniką przekształcenia Remove Parameter.

W efekcie przekształcenia grupa parametrów metody została zastąpiona jednym, który jest źródłem tych parameterów.

Replace Parameter with Explicit Method

Problem

Zachowanie metody zależy od wartości jednego z parametrów

Cel

Utworzenie oddzielnych metod dla każdej wartości parametru

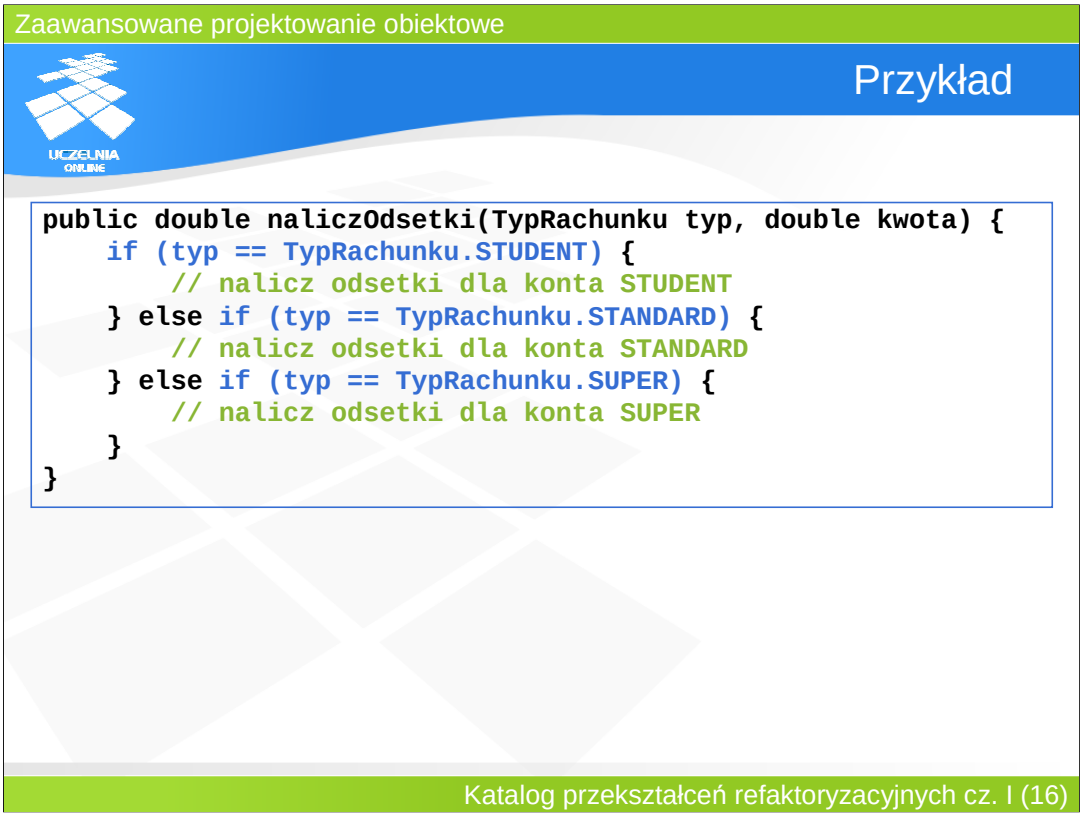
Mechanika

- utwórz osobne metody dla każdej wartości parametru
- w każdej gałęzi instrukcji warunkowej wywołaj odpowiednią nową metodę
- skompiluj i przetestuj
- zastąp wywołania starej metody wywołaniami właściwej nowej metody
- opcjonalnie: usuń starą metodę

M. Fowler, 1999

Problem rozwiązywany przez to przekształcenie dotyczy metod, których zachowanie zależy od wartości jednego z parametrów. Rozwiązanie polega na utworzeniu osobnych wersji metody dla każdej wartości tego parametru (jeżeli jest ich kilka) i właściwym ich nazwaniu.

Przekształcenie zaczyna się od utworzenia nowych metod, które w nazwach będą zawierały jedną z wartości usuwanego parametru. Następnie z instrukcji warunkowej należy skopiować zawartość właściwej gałęzi, a w jej miejscu umieścić wywołanie nowej metody. Kolejnym krokiem jest aktualizacja klientów, aby korzystali z nowych wersji metod, i usunięcie nieużywanej obecnie oryginalnej wersji metody.



Przekształcenie to zostało przedstawione na przykładzie metody *naliczOdsetki().* Czynność wykonana przez tę metodę zależy od wartości parametru TypRachunku.

Przykład

```
public double naliczOdsetki(TypRachunku typ, double kwota) {
    if (typ == TypRachunku.STUDENT) {
        naliczOdsetkiStudent(kwota);
    } else if (typ == TypRachunku.STANDARD) {
        naliczOdsetkiStandard(kwota);
    } else if (typ == TypRachunku.SUPER) {
        naliczOdsetkiSuper(kwota);
    }
}
public double naliczOdsetkiStudent(double kwota) {
    // nalicz odsetki dla konta STUDENT
}
public double naliczOdsetkiStandard(double kwota) {
    // nalicz odsetki dla konta STANDARD
}
public double naliczOdsetkiSuper(double kwota) {
    // nalicz odsetki dla konta SUPER
}
```


W miejsce fragmentów kodu wewnątrz oryginalnej metody umieszczane są wywołania nowoutworzonych metod naliczających odsetki dla każdego typu rachunku osobno.

W efekcie przekształcenia metoda posiadająca parametr decydujący o jej zachowaniu została przekształcona w rodzinę metod pozbawionych tego parametru.

Replace Parameter with Method

Problem

Wynik wykonania metody jest parametrem innej metody tego obiektu

Cel

Obliczenie wartości metody wewnątrz innej metody

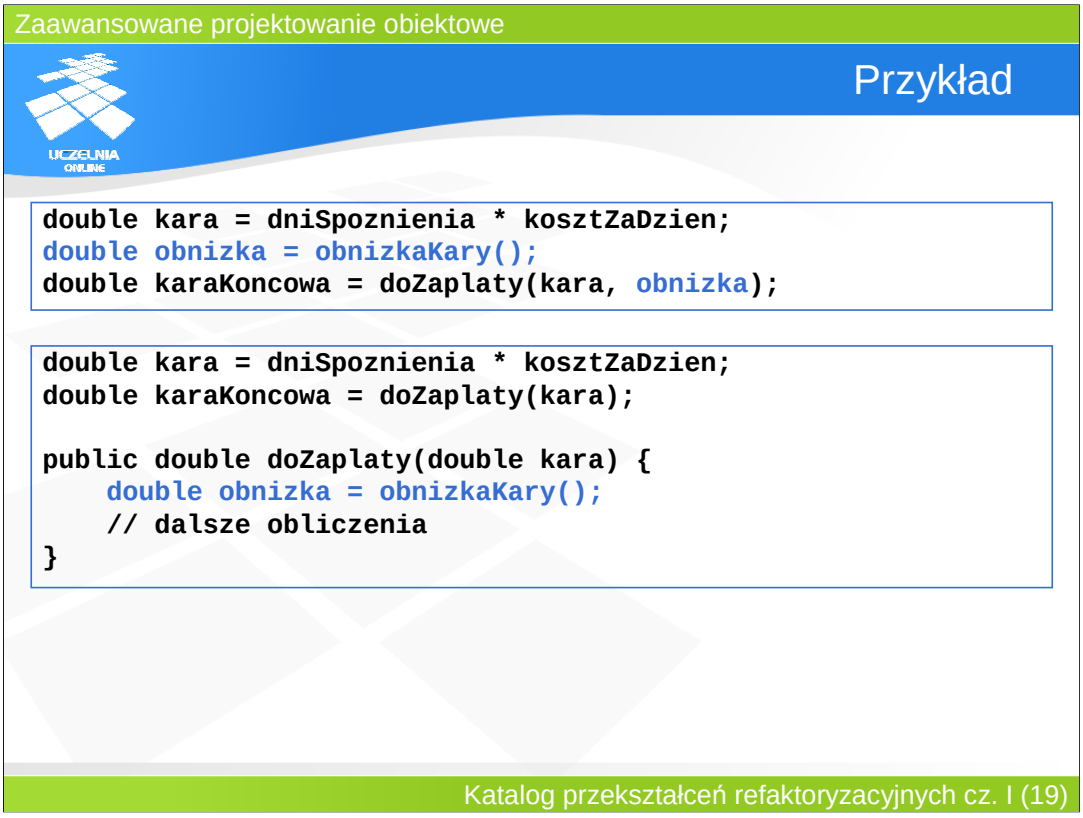
Mechanika

- opcjonalnie: wyłącz obliczenie parametru do nowej metody
- po kolei zastąp odwołania do parametru wywołaniami metody obliczającej go (uwaga na efekty uboczne!)
- skompiluj i przetestuj
- wykonaj przekształcenie *Remove Parameter* na nieużywanym parametrze

M. Fowler, 1999

Ostatnim przekształceniem należącym do kategorii związanej z przekształceniami w obrębie sygnatury metody jest zastąpienie parametru wywołaniem metody. Stosuje się je w sytuacjach, gdy wynik wykonania metody staje się parametrem innej metody tego obiektu. Zamiast obliczać tę wartość na zewnątrz metody i przekazywać ją jako parametr, można dokonać niezbędnych obliczeń wewnątrz niej.

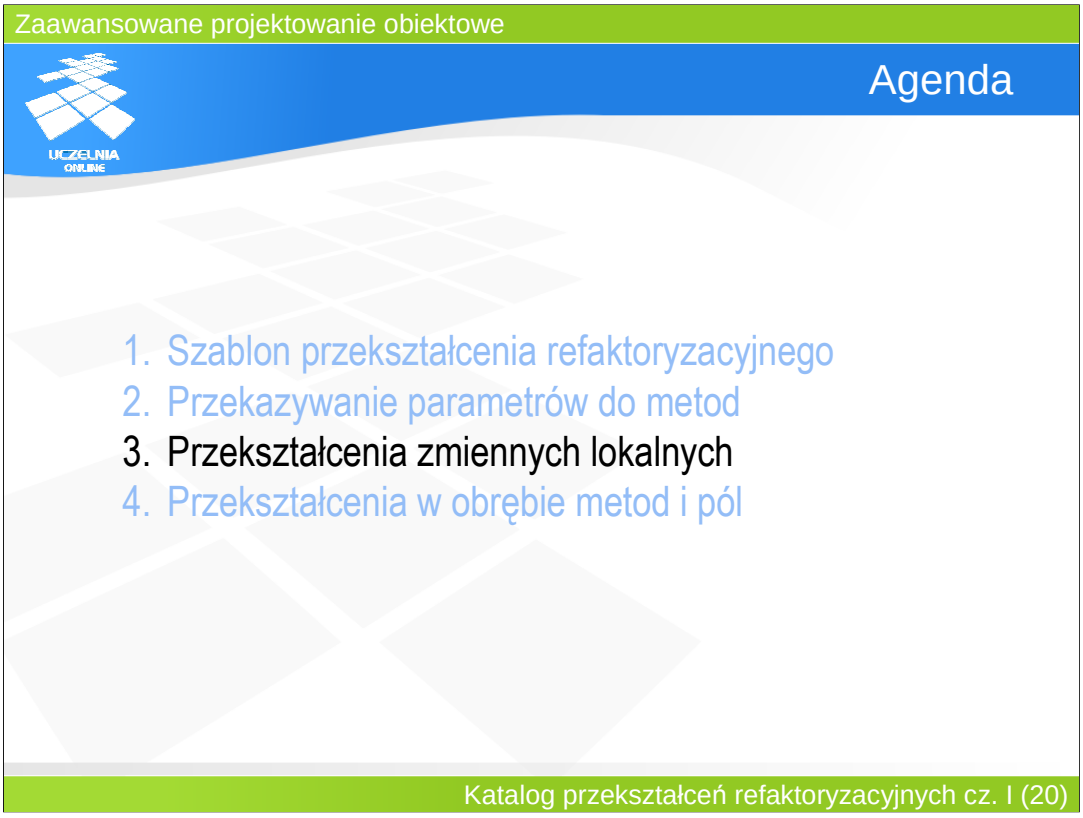
Przekształcenie rozpoczyna się od wyłączenia polecenia obliczenia wartości parametru do nowej metody (o ile nie jest już obliczany przez dedykowaną metodę). Następnie należy wewnątrz metod zastąpić odwołania do parametru wywołaniami metody, która go oblicza. Trzeba jednak zwrócić uwagę na potencjalne efekty uboczne, związane z wielokrotnym obliczaniem wartości parametru (np. jeżeli obliczanie tej wartości modyfikuje jakąś zmienną). W ostatnim kroku nieużywany parametr może zostać usunięty.



Na przykład, na kwotę opłaty karnej za nieterminowy zwrot książek do biblioteki składa się kilka elementów: czas spóźnienia, koszt za jeden dzień i ewentualna obniżka. Metoda obliczająca tę wartość, *doZaplaty(),* przyjmuje dwa parametry: wysokość naliczonej kary i wysokość obniżki, będącej efektem wykonania innej metody.

Po przekształceniu metoda *doZaplaty()* przyjmuje tylko jeden parametr, ponieważ pozostałe ważne dla siebie informacje może zdobyć samodzielnie.

Dzięki temu przekształceniu udało się usunąć niepotrzebny parametr poprzez obliczenie jego wartości wewnątrz metody.



Druga grupa przekształceń dotyczy zmiennych lokalnych. Stanowią one istotny problem dla wielu przekształceń, np. Extract Method, ponieważ wymagają przekazywania ich w postaci parametrów, co niepotrzebnie zwiększa stopień powiązań między metodami.

Inline Temp

Problem

Tymczasowa zmienna przechowuje wynik prostego wyrażenia

Cel

Zastąpienie zmiennej każdorazowym obliczeniem wyrażenia

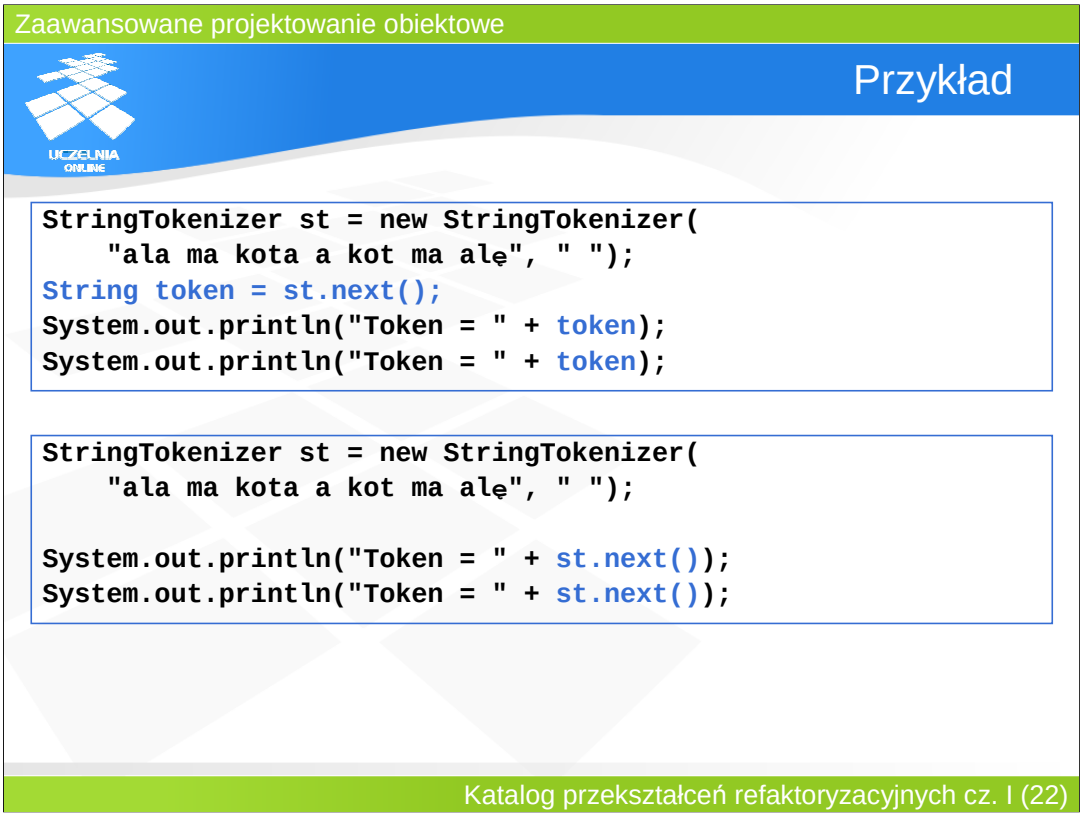
Mechanika

- oznacz zmienną jako sfinalizowaną
- kolejno zastąp odwołania do zmiennej obliczeniem wartości wyrażenia
- skompiluj i przetestuj (uwaga na efekty uboczne!)
- usuń deklarację zmiennej tymczasowej
- skompiluj

M. Fowler, 1999

Pierwsze przekształcenie było wcześniej omawiane podczas dyskusji na temat skutków ubocznych niektórych refaktoryzacji. Służy ono do zastąpienia zmiennej lokalnej użytej w prostym obliczeniu, którego wynik jest przechowywany w zmiennej lokalnej, wyrażeniem obliczającym jej wartość.

Pierwszym krokiem przekształcenia jest zadeklarowanie zmiennej jako sfinalizowanej, co zapobiega wielokrotnemu przypisywaniu wartości do niej. Następnie należy kolejno zastąpić odwołania do zmiennej obliczeniem wartości wyrażenia, testując po każdej zmianie. Po zakończeniu tego etapu można z klasy usunąć deklarację zmiennej lokalnej.



Przykładem błędnego przeprowadzenia takiej zmiany jest następujący fragment kodu. Dotyczy on obiektu StringTokenizer, dzielącego napisy na wyrazy i zwracającego je w postaci pojedynczych tokenów.

Zamiast zapamiętywania wartości metody *st.next()* po przekształceniu metoda ta jest wywoływana w każdym miejscu, w którym dotychczas następowało odwołanie do zmiennej. Jednak takie przekształcenie wprowadza błąd do programu, ponieważ metoda *st.next(),* która zmienia stan obiektu StringTokenizer, zostanie obecnie wywołana wielokrotnie, co wpłynie na zwracane przez nią wartości. Dlatego w przypadku tego przekształcenia należy zwrócić szczególną uwagę na problem efektów ubocznych.

Introduce Explaining Variable

Problem

Obliczane wyrażenie jest długie i złożone

Cel

Podzielenie wyrażenia na zmienne tymczasowe

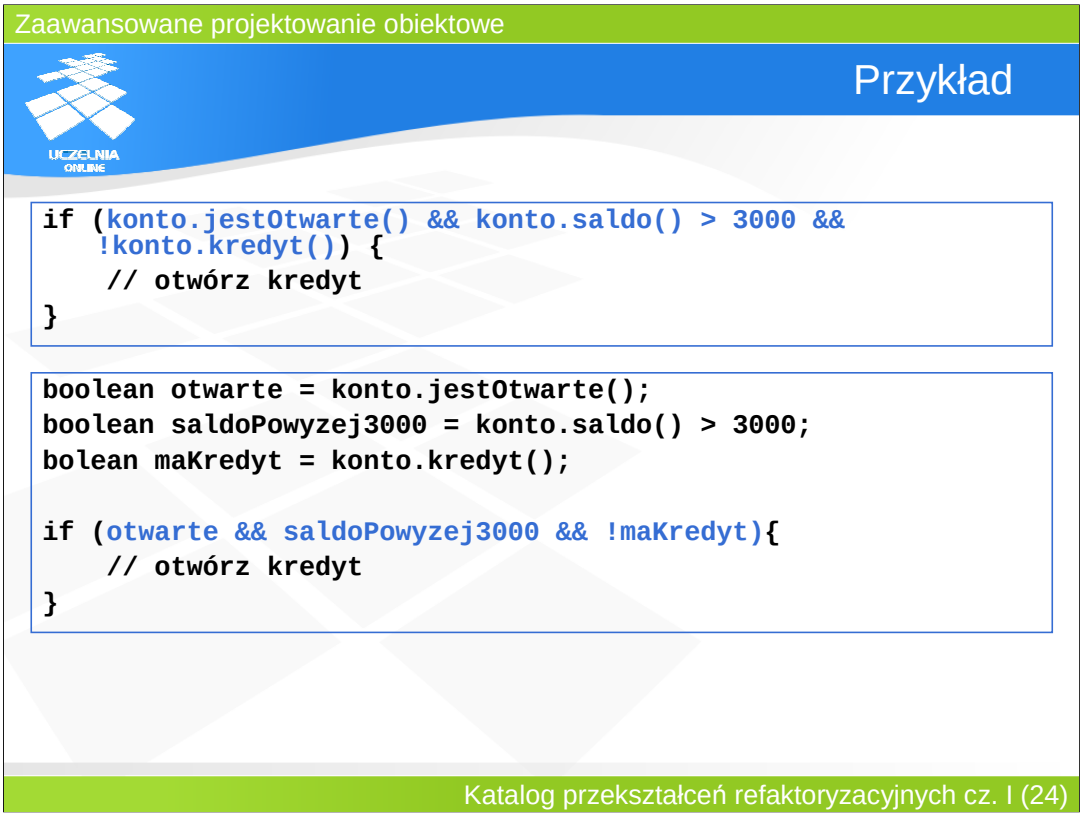
Mechanika

- stwórz sfinalizowaną zmienną i przypisz do niej wartość części wyrażenia oryginalnego
- zastąp fragment wyrażenia oryginalnego odwołaniem do nowej zmiennej
- skompiluj

M. Fowler, 1999

To przekształcenie jest dość proste i intuicyjne. Gdy wyrażenie jest długie, złożone i nieczytelne, można podzielić je na mniejsze fragmenty i ich wartości zapamiętać w zmiennych tymczasowych.

Refaktoryzacja rozpoczyna się od stworzenia sfinalizowanej zmiennej i przypisania do niej wartości wybranego fragmentu wyrażenia. Następnie fragment ten w oryginalnym wyrażeniu jest zastępowany odwołaniem do nowej zmiennej. Ponieważ zmienna ta z założenia jest wykorzystana tylko raz, dlatego efekty uboczne podczas tego przekształcenia nie występują.



Przykładem jest warunek uzależniający otwarcie kredytu posiadaniem otwartego konta, stanem salda przekraczającym 3000 PLN oraz brakiem innego otwartego kredytu. To wyrażenie jest dość złożone, dlatego można podzielić je na mniejsze fragmenty i zapamiętać ich wartości w nowych zmiennych lokalnych. Zapis wyrażenia oryginalnego stał się zatem znacznie prostszy.

W efekcie przekształcenia fragmenty wyrażenia zostały wydzielone do zmiennych lokalnych. Jednak zmienne te utrudniają dalsze operacje na metodzie (m.in. jej podział), dlatego będą musiały być w kolejnych refaktoryzacjach przekształcane do postaci metod.

Replace Temp with Query

Problem

Zmienna tymczasowa uniemożliwia uproszczenie kodu

Cel

Zastąpienie zmiennej metodą obliczającą jej wartość

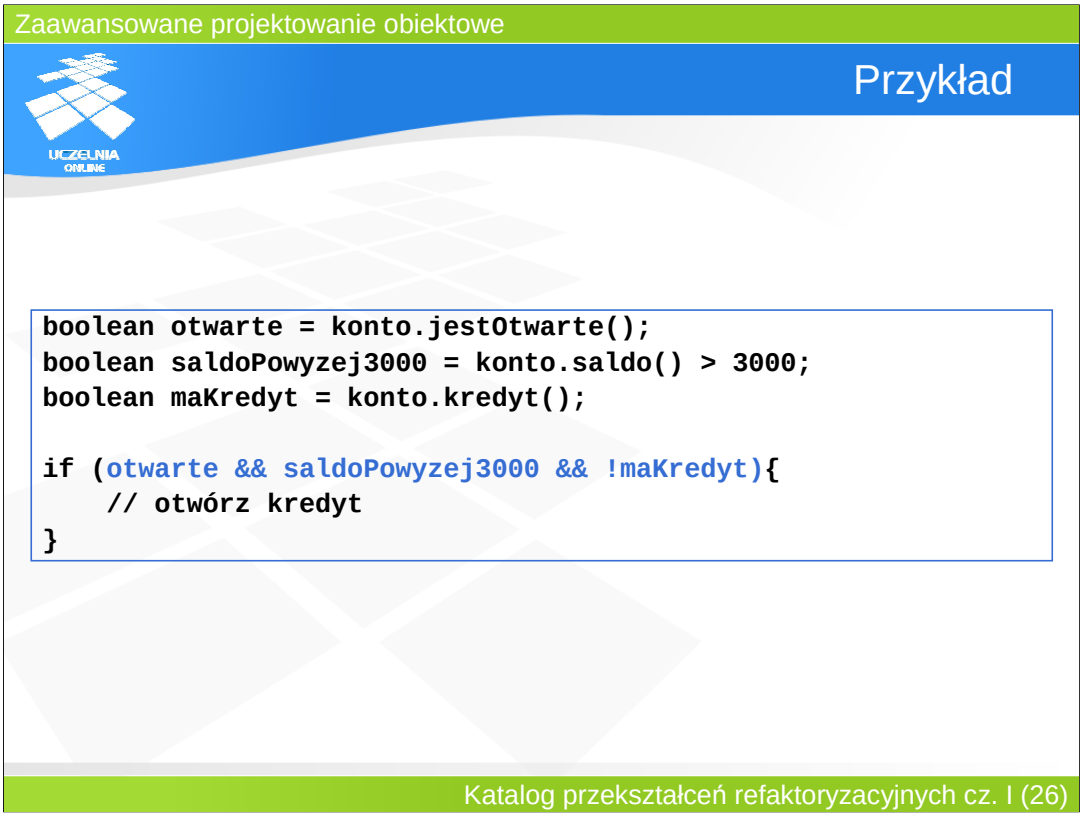
Mechanika

- oznacz zmienną jako sfinalizowaną
- skompiluj
- zastosuj *Extract Method* wobec wyrażenia będącego wartością zmiennej
- przypisz zmiennej wartość utworzonej metody
- skompiluj i przetestuj
- usuń zmienną stosując *Inline Temp*

M. Fowler, 1999

Pierwszym krokiem do tego celu jest zastąpienie zmiennej metodą obliczającą jej wartość.

Podobnie jak w wielu innych operacjach na zmiennych lokalnych, pierwszym zadaniem jest zadeklarowanie zmiennej jako sfinalizowanej. Następnie wobec wyrażenia będącego wartością zmiennej należy wykonać przekształcenie Extract Method, a wynik nowej metody przypisać do zmiennej. W ostatnim etapie zmienna ta może zostać usunięta za pomocą przekształcenia Inline Temp.



Przykład zaczyna się od momentu zakończenia poprzedniego przekształcenia: wyrażenie jest zbudowane ze zmiennych lokalnych.

Przykład

```
private boolean otwarte() {
    return konto.jestOtwarte();
}
private boolean saldoPowyzej3000() {
    return konto.saldo() > 3000;
}
private boolean maKredyt() {
    return konto.kredyt();
}

if (otwarte() && saldoPowyzej3000() && !maKredyt()){
    // otwórz kredyt
}
```


Każda z tych zmiennych jest przekształcana w metodę o tym samym typie, która zwraca tę samą wartość.

W efekcie w wyrażeniu zamiast zmiennych pojawiają się metody, które nie utrudniają w takim stopniu dalszych operacji refaktoryzacyjnych.

Split Temporary Variable

Problem

Zmienna lokalna jest wielokrotnie używana w różnych celach

Cel

Utworzenie nowej zmiennej dla każdego przypisania wartości

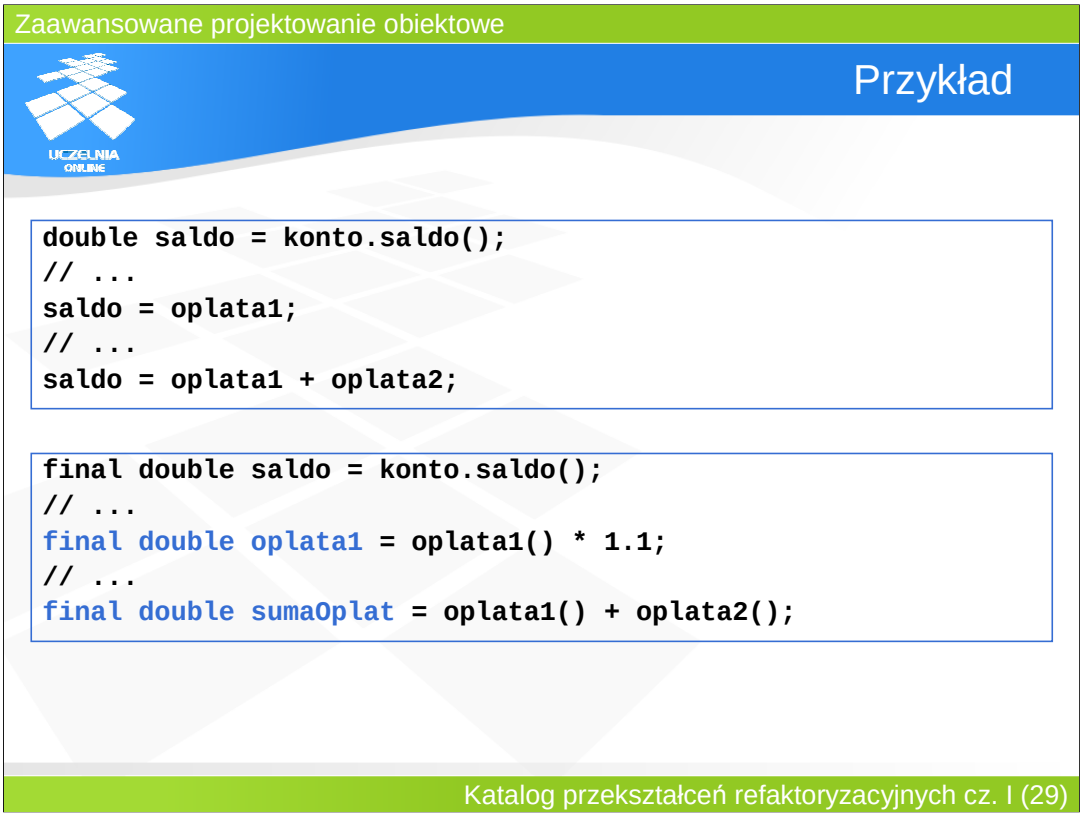
Mechanika

- oznacz zmienną jako sfinalizowaną
- przy pomocy kompilatora znajdź kolejne przypisania do zmiennej
- zadeklaruj nową zmienną sfinalizowaną o intuicyjnej nazwie dla każdego przypisania
- skompiluj i przetestuj

M. Fowler, 1999

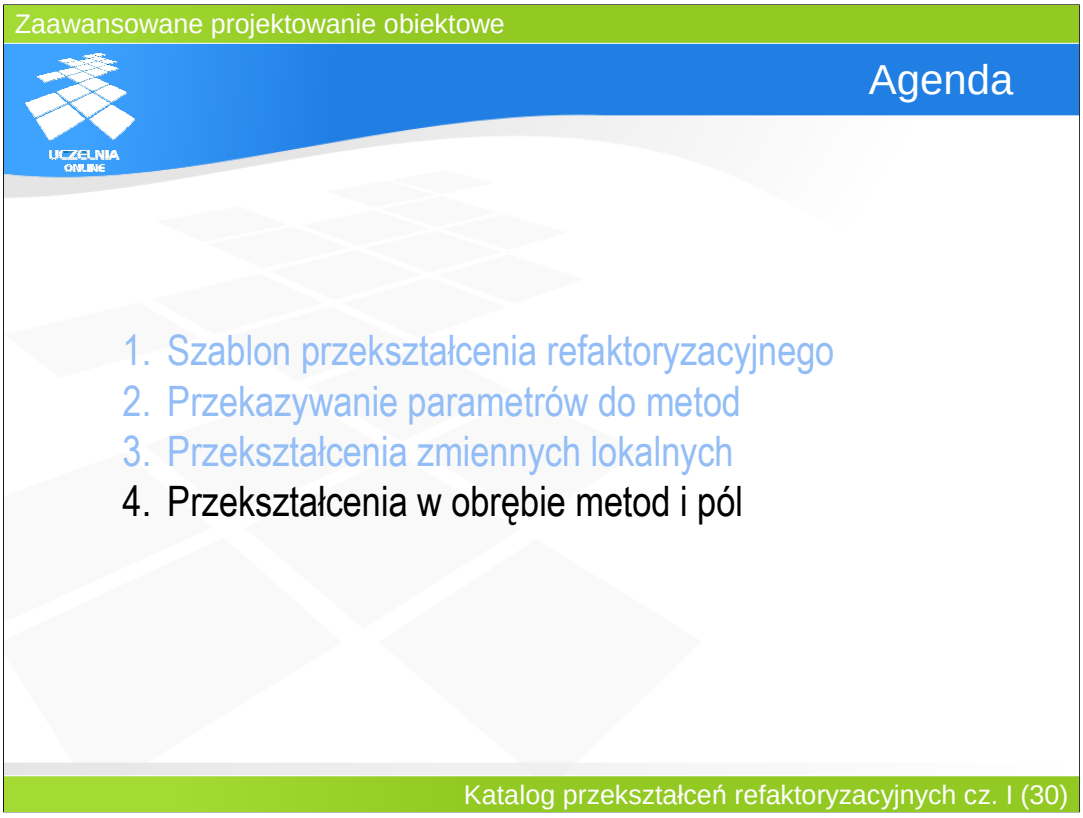
To przekształcenie dotyczy problemu wielokrotnego użycia zmiennych lokalnych do przechowywania nie związanych ze sobą danych. W efekcie nazwa użytej w ten sposób zmiennej nie oznacza już swego pierwotnego przeznaczenia, co pogarsza czytelność i obniża zrozumienie kodu. Celem refaktoryzacji jest podział jej na nowe zmienne, tak aby przypisanie wartości zawsze dotyczyło nowej zmiennej lokalnej.

Przekształcenie jest realizowane przy istotnym wsparciu ze strony kompilatora. Pierwszym krokiem jest zadeklarowanie zmiennej jako sfinalizowanej. To powoduje, że próba kompilacji automatycznie wskazuje miejsce ponownego przypisania wartości do tej zmiennej. W tym miejscu należy zadeklarować nową zmienną o nazwie odpowiadającej jej przeznaczeniu, i kontynuować pracę aż do usunięcia wszystkich przypisań.

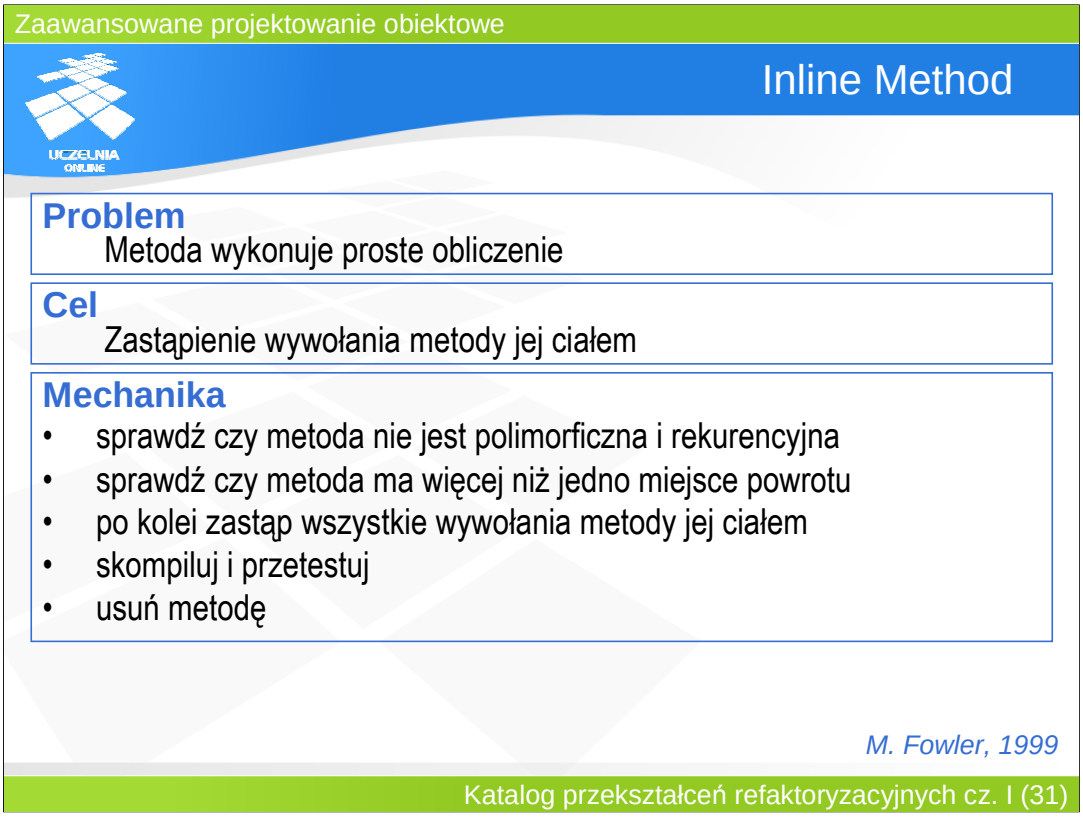


Przedstawiona na przykładzie zmienna saldo przechowuje kilka różnych wartości. W efekcie przekształcenia jest ona rozbita na kilka innych zmiennych, do których przypisania następują tylko raz. Oczywiście, przekształcenie to zwiększa liczbę zmiennych lokalnych, jednak można je w dalszej części refaktoryzacji usunąć, stosując m.in. przekształcenie Replace Temp with Query.

Dzięki wykonaniu przekształcenia każda zmienna lokalna ma przypisaną wartość tylko raz.

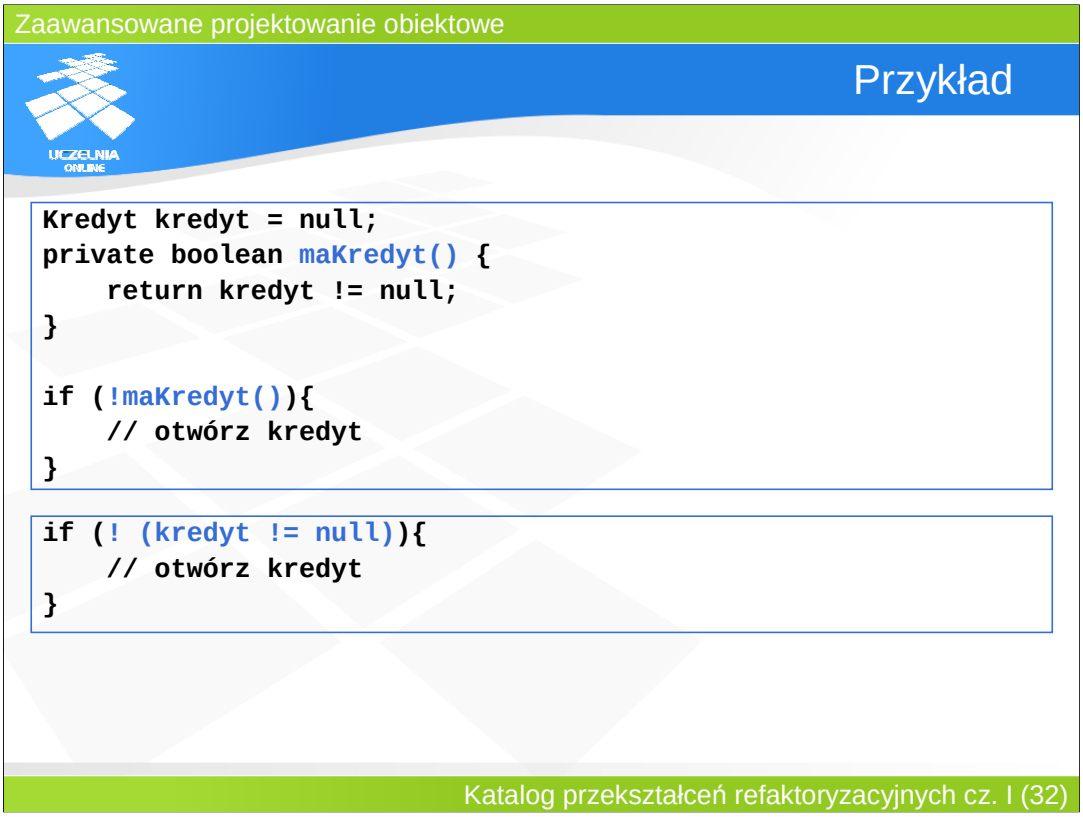


Ostatnia, najobszerniejsza grupa przekształceń obejmuje refaktoryzacje wykonywane na poziomie pól i metod wewnątrz klasy.



Najprostszym przekształceniem jest komplementarne do Extract Method rozwinięcie metody, tzn. zastąpienie wywołania metody jej ciałem. Dzięki temu metody wykonujące proste obliczenia, które nie zasługują na umieszczenie w metodzie, mogą zostać usunięte.

Podobnie jak w wielu innych przekształceniach związanych z metodami, najpierw należy upewnić się, że metoda nie jest polimorficznie pokrywana. Dodatkowym warunkiem jest brak jej rekurencyjnych wywołań, które uniemożliwiają (lub bardzo utrudniają) prawidłowe zakończenie refaktoryzacji, oraz jedno miejsce powrotu z metody (choć w niektórych przypadkach wielokrotne instrukcje *return* mogą zostać zastąpione operatorem trójargumentowym). Dalsza część polega na kolejnym zastępowaniu wywołań ciałem metody oraz każdorazowym testowaniu przekształcanego fragmentu. Ostatnim krokiem jest usunięcie zbędnej metody.



Przykładem może być prosta metoda *maKredyt(),* która określa, czy zmienna kredyt została zainicjowana, czy nie. Przekształcenie powoduje zastąpienie wywołania takiej metody jej ciałem. Może to prowadzić do pogorszenia czytelności kodu, dlatego stosowanie tego przekształcenia powinno być dobrze uzasadnione.

Encapsulate Downcast

Problem

Metoda zwraca wynik wymagający rzutowania przez klienta

Cel

Przesunięcie rzutowania do wewnątrz metody

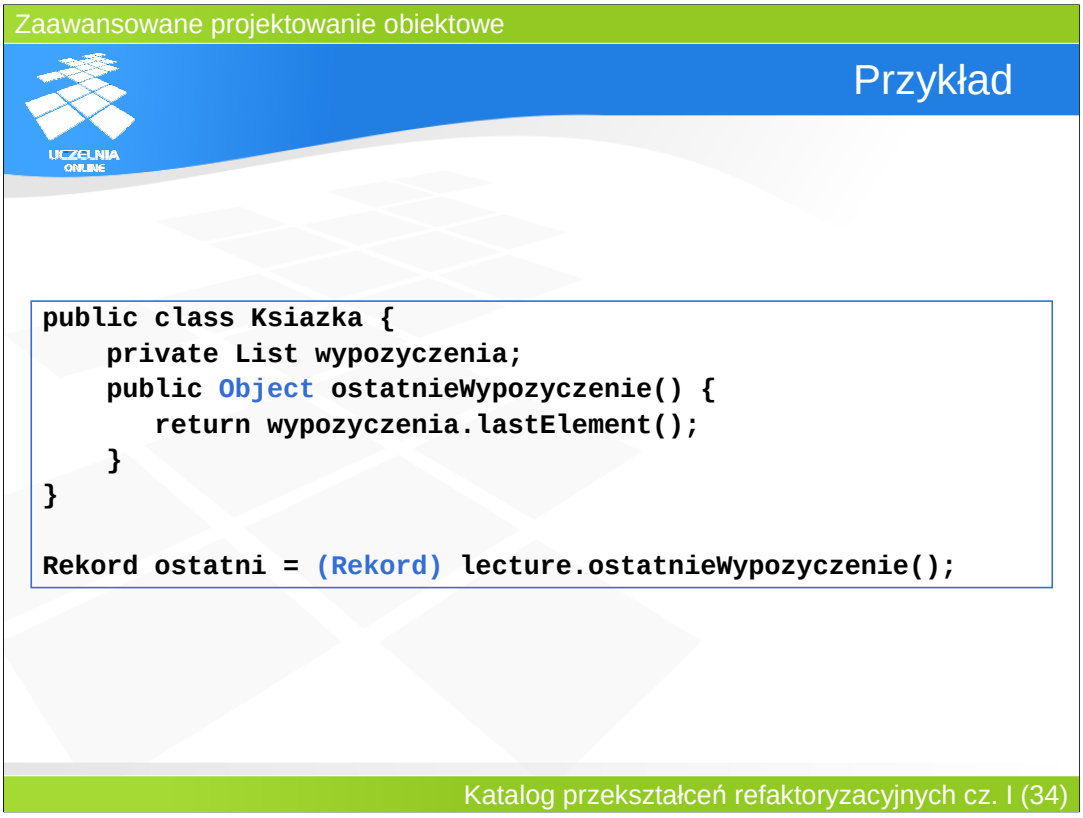
Mechanika

- znajdź wszystkie przypadki rzutowania wyniku metody
- przesuń rzutowanie do najbardziej ogólnego typu do wnętrza metody

M. Fowler, 1999

Przekształcenie to jest prostym mechanizmem hermetyzacji, który pozwala ograniczyć problemy związane z rzutowaniem klas. Występują one jednak tylko w niektórych językach o statycznym systemie typów, dlatego znaczenie tej refaktoryzacji jest ograniczone.

Problem rozwiązywany przez nie dotyczy konieczności rzutowania wyniku metody przez klienta, i zwykle jest związany z operacjami na kolekcjach. Rozwiązaniem jest przesunięcie rzutowania do wewnątrz metody.



Przykładowa klasa Książka przechowuje listę z informacjami (typu Rekord) o jej wypożyczeniach. Metoda *ostatnieWypozyczenie()* zwraca ostatni rekord tej listy. Jednak deklarowany typ tej metody jest zbyt ogólny i wymaga od klienta rzutowania, co może spowodować zgłoszenie wyjątku.

Przykład

```
public class Ksiazka {
    List wypozyczenia;
    Rekord ostatnieWypozyczenie() {
       return (Rekord) wypozyczenia.lastElement();
    }
}
```

```
public class Ksiazka {  // JDK 5.0+
    List<Rekord> wypozyczenia;
    Rekord ostatnieWypozyczenie() {
       return wypozyczenia.lastElement();
    }
}
```


Dlatego konieczne jest usunięcie rzutowania wykonywanego przez klientów. W pierwszym z możliwych rozwiązań operacja rzutowania zostaje przesunięta do wewnątrz metody, dzięki czemu klient otrzymuje obiekt z właściwą informacją o jego typie.

Drugie rozwiązanie wykorzystuje mechanizm typów generycznych, obecny w Javie od wersji 5.0. Lista wypożyczeń została sparametryzowana klasą Rekord, co usuwa konieczność przeprowadzania rzutowania.

Self-Encapsulate Field

Problem

Obliczanie wartości pola wymaga pokrycia w klasach potomnych

Cel

Przeniesienie dostępu do pola do metod set/get

Mechanika

- utwórz niepubliczną parę metod set/get
- zastąp odwołania do pola wywołaniami odpowiedniej metody
- oznacz pole jako prywatne
- skompiluj

M. Fowler, 1999

Fowler wyróżnił dwa przekształcenia dotyczące hermetyzacji pól, w zależności od celu ich stosowania. W pierwszym przypadku jest on związany z koniecznością zmiany sposobu odczytywania wartości pola w podklasach. Ponieważ pola nie są polimorficzne, dlatego konieczne jest hermetyzowanie ich poprzez metody get/set.

Mechanika jest intuicyjnie prosta: należy utworzyć parę metod get/set i zadeklarować w nich niepubliczny poziom widoczności (ponieważ użytkownikami tych metod mają być podklasy, a nie klasy dostępne w inny sposób). Następnie odwołania do pola występujące w podklasach należy zastąpić wywołaniami metod get/set i zadeklarować pole jako prywatne.

Encapsulate Field

Problem

Pole ma dostęp publiczny

Cel

Zastąpienie dostępu do pola metodami set/get

Mechanika

- utwórz publiczną parę metod set/get
- zastąp odwołania do pola wywołaniami odpowiedniej metody
- oznacz pole jako prywatne
- skompiluj

M. Fowler, 1999

Celem drugiego przekształcenia jest hermetyzacja dostępu wobec zewnętrznych klientów klasy.

Mechanika przekształcenia jest identyczna, z tą różnicą, że metody get/set są zadeklarowane jako publiczne.

Encapsulate Collection

M. Fowler, 1999

Problem

Metoda *get()* w klasie właściciela zwraca kolekcję dostępną do modyfikacji

Cel

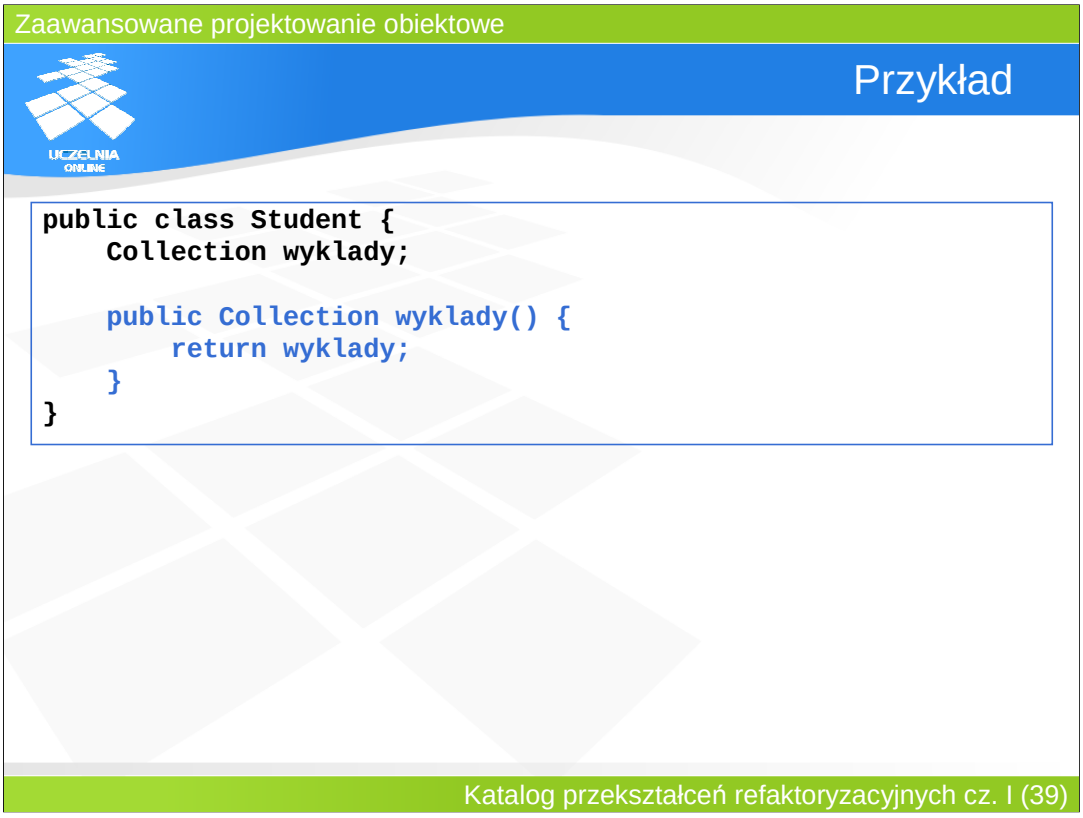
Przeniesienie odpowiedzialności za kolekcję do jej właściciela

Mechanika

- dodaj w klasie właściciela metody *add()* i *remove()* dla kolekcji
- zmień bezpośrednie odwołania do metod *add()* i *remove()* kolekcji odwołaniami do metod jej właściciela
- zmień metodę *get()* zwracającą kolekcję, tak aby zwracała jej widok tylko do odczytu
- skompiluj i przetestuj
- opcjonalnie: zmień metodę *get()* tak, aby zwracała *Iterator*

Problem rozwiązywany przez to przekształcenie został już omówiony w zarysie podczas pierwszego wykładu. Polega on na udostępnieniu do modyfikacji kolekcji poza kontrolą jej właściciela. Właściciel, poprzez metodę typu *get()*, przekazuje klientom referencję do kolekcji, która następnie może być swobodnie modyfikowana. Celem przekształcenia jest przeniesienie odpowiedzialności za modyfikację kolekcji z niej samej do jej właściciela.

Refaktoryzacja rozpoczyna się od zdefiniowania w klasie właściciela metod *add()* i *remove()* dla elementów kolekcji, a następnie przeniesienie wywołań metod kolekcji do nowo utworzonych metod w klasie jej właściciela. Drugim krokiem jest zmiana metody zwracającej referencję do kolekcji w taki sposób, aby nie pozwalała ona na modyfikację tej kolekcji. Istnieje kilka możliwych rozwiązań: użycie specjalizowanego opakowania kolekcji (zob. wzorzec Decorator), który uniemożliwi jej modyfikację (omówionego podczas wykładu nt. kolekcji w języku Java), lub po prostu zwrócenie jej kopii.



Przykład przekształcenia zostanie zaprezentowany na klasie Student, która posiada kolekcję wykładów, na które on uczęszcza. Kolekcja ta jest udostępniona poprzez metodę *wyklady()*.

Przykład

```
public class Student {
    Collection wyklady;
    public boolean dodajWyklad(Wyklad w) {
        return wyklady.add(w);
    }
    public boolean usunWyklad(Wyklad w) {
        return wyklady.remove (w);
    }
    public Collection wyklady() {
        return Collections.unmodifiableCollection(wyklady);
    }
}
```


Po wykonaniu przekształcenia w klasie zaszły trzy zmiany: pojawiły się metody *dodajWyklad()* i *usunWyklad()*, które służą do kontrolowanego modyfikowania kolekcji wykładów, natomiast metoda *wyklady()* zwraca niemodyfikowalną wersję tej kolekcji.

W efekcie wszelkie zmiany w kolekcji mogą być dokonane jedynie poprzez obiekt jej właściciela, a odczyt kolekcji jest możliwy z wykorzystaniem dotychczasowej metody *wyklady()*.

Move Field

Problem

Pole jest częściej używane przez obcą klasę niż przez własną

Cel

Przesunięcie pola do właściwej klasy

Mechanika

- wykonaj *Encapsulate Field*
- utwórz identyczne pole z metodami set/get w docelowej klasie
- skompiluj klasę docelową
- zmień metody set/get w klasie źródłowej, aby delegowały do klasy docelowej
- skompiluj i przetestuj
- usuń pole w klasie źródłowej

M. Fowler, 1999

Przeniesienie pola jest przekształceniem dotyczącym dwóch klas związanych ze sobą relacją asocjacji, agregacji lub kompozycji (ale nie dziedziczenia). Wykonuje się je w sytuacji, gdy pole to jest częściej używane przez obcą klasę niż przez klasę macierzystą. Celem jest zatem zwiększenie spójności klas poprzez przeniesienie go do właściwej klasy.

Pierwszym krokiem jest hermetyzacja pola i udostępnienie metod dostępu do niego get/set (zob. przekształcenie Encapsulate Field). Następnie należy utworzyć identyczne pole wraz z metodami dostępu w klasie docelowej. W tym momencie należy zmienić metody set/get w klasie oryginalnej, tak aby wywoływały ich odpowiedniki w klasie docelowej. Nieużywane pole w klasie źródłowej może zostać w tym momencie usunięte.

Warto zwrócić uwagę, że po przekształceniu do pola można odwołać się zarówno z klasy źródłowej, jak i klasy docelowej, co pozwala na elastyczne dostosowanie klientów do nowego położenia pola. W pewien sposób pozwala to także ukryć strukturę tych dwóch klas, jednak z drugiej strony rozmywa nieco ich odpowiedzialność oraz wiąże je silną zależnością.

Move Method

Problem

Metoda korzysta częściej z metod w obcej klasie niż z własnych

Cel

Przesunięcie metody do właściwej klasy

Mechanika

- sprawdź, czy metoda nie jest polimorficzna w obu klasach
- zadeklaruj metodę w klasie docelowej
- skopiuj ciało metody do klasy docelowej
- skompiluj klasę docelową
- zmień metodę źródłową w delegację do metody w klasie docelowej
- skompiluj i przetestuj

M. Fowler, 1999

Przekształcenie polegające na przeniesieniu metody również dotyczy dwóch klas związanych inaczej niż poprzez dziedziczenie. Jeżeli metoda częściej odwołuje się do metod obcej klasy (jest to oznaka przykrego zapachu Feature Envy), wówczas powinna być przeniesiona do innej klasy.

Refaktoryzacja ta jest szczególnym przypadkiem przekształcenia Move Field, ponieważ pomija krok związany z hermetyzacją pola. Dodatkowo należy zwrócić uwagę, czy przenoszona metoda nie jest wykorzystywana polimorficznie, co uniemożliwiłoby usunięcie jej w klasie źródłowej.

Inline Class

Problem

Odpowiedzialność klasy jest zbyt mała

Cel

Wchłonięcie klasy przez jej klienta

Mechanika

- przenieś metody klasy źródłowej do jej klienta
- zmień metody w klasie źródłowej, tak aby delegowały wywołania do klienta
- zmień innych klientów, tak aby odwoływali się do nowej klasy
- skompiluj i przetestuj
- wykonaj *Move Field*
- usuń klasę źródłową

M. Fowler, 1999

Przekształcenie Inline Class służy do usunięcia klas, których odpowiedzialność jest zbyt mała, żeby uzasadniała ich istnienie. Usunięcie odbywa się poprzez wchłonięcie klasy przez klienta związanego z nią relacją asocjacji.

Mechanika polega na wykonywaniu przekształceń Move Method i Move Field do momentu otrzymania pustej klasy źródłowej, którą można usunąć.

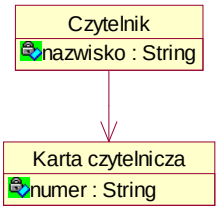
Przykład

```
public class Czytelnik {
    private KartaCzytelnicza karta;
    private String nazwisko;

    public String nazwisko() {
        return nazwisko;
    }
    public KartaCzytelnicza kartaCzytelnicza() {
        return karta;
    }
}

public class KartaCzytelnicza { // do usunięcia
    private String numer;

    public void ustawNumer(String numer) {
        this.numer = numer;
    }
    public String numer() {
        return numer;
    }
}
```

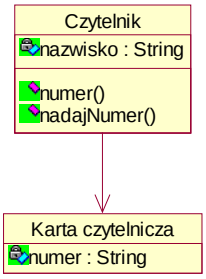


Przykładem takiej klasy jest KartaCzytelnicza, która jest własnością Czytelnika. Odpowiedzialność Karty Czytelniczej jest ograniczona tylko do przechowywania swojego numeru, dlatego zadanie to można przenieść na klasę Czytelnik, a Kartę Czytelniczą – usunąć.

Przykład

```
public class Czytelnik {
    private KartaCzytelnicza karta;
    private String nazwisko;

    public String nazwisko() {
        return nazwisko;
    }
    public KartaCzytelnicza kartaCzytelnicza() {
        return karta;
    }
    public void ustawNumer(String numer) {
        karta.ustawNumer(numer);
    }
    public String numer() {
        return karta.numer();
    }
}
```



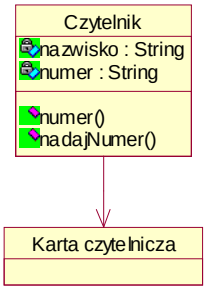
Pierwszy krok polega na stworzeniu w klasie Czytelnik metod, które odpowiadają metodom klasy Karta Czytelnicza, i są do niej delegowane. Dzięki temu można zaktualizować inne klasy klientów, aby odwoływały się do numeru karty z poziomu karty Czytelnik.

Przykład

```
public class Czytelnik {
    private KartaCzytelnicza karta;
    private String nazwisko;
    private String numer;

    public String nazwisko() {
        return nazwisko;
    }
    public KartaCzytelnicza kartaCzytelnicza() {
        return karta;
    }
    public void ustawNumer(String numer) {
        this.numer = numer;
    }
    public String numer() {
        return numer;
    }
}

public class KartaCzytelnicza {
   // pusta - do usunięcia
}
```



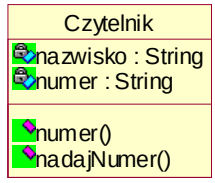
Kolejnym krokiem jest przeniesienie pola *numer* i usunięcie delegacji do jego metod dostępowych w klasie KartaCzytelnicza. W ten sposób pole i metody w tej klasie stają się nieużywane i mogą zostać usunięte.

Przykład

```
public class Czytelnik {

    private String nazwisko;
    private String numer;

    public String nazwisko() {
        return nazwisko;
    }
    public KartaCzytelnicza kartaCzytelnicza() {
        return karta;
    }
    public void ustawNumer(String numer) {
        this.numer = numer;
    }
    public String numer() {
        return numer;
    }
}
```



W efekcie przekształcenia klasa Czytelnik w pełni przejmuje odpowiedzialność klasy Karta Czytelnicza, która zostaje usunięta z systemu.

Replace Method with Method Object

M. Fowler, 1999

Problem

Długa metoda posiada zbyt wiele zmiennych, aby ją podzielić

Cel

Zamiana metody w obiekt z jedną metodą

Mechanika

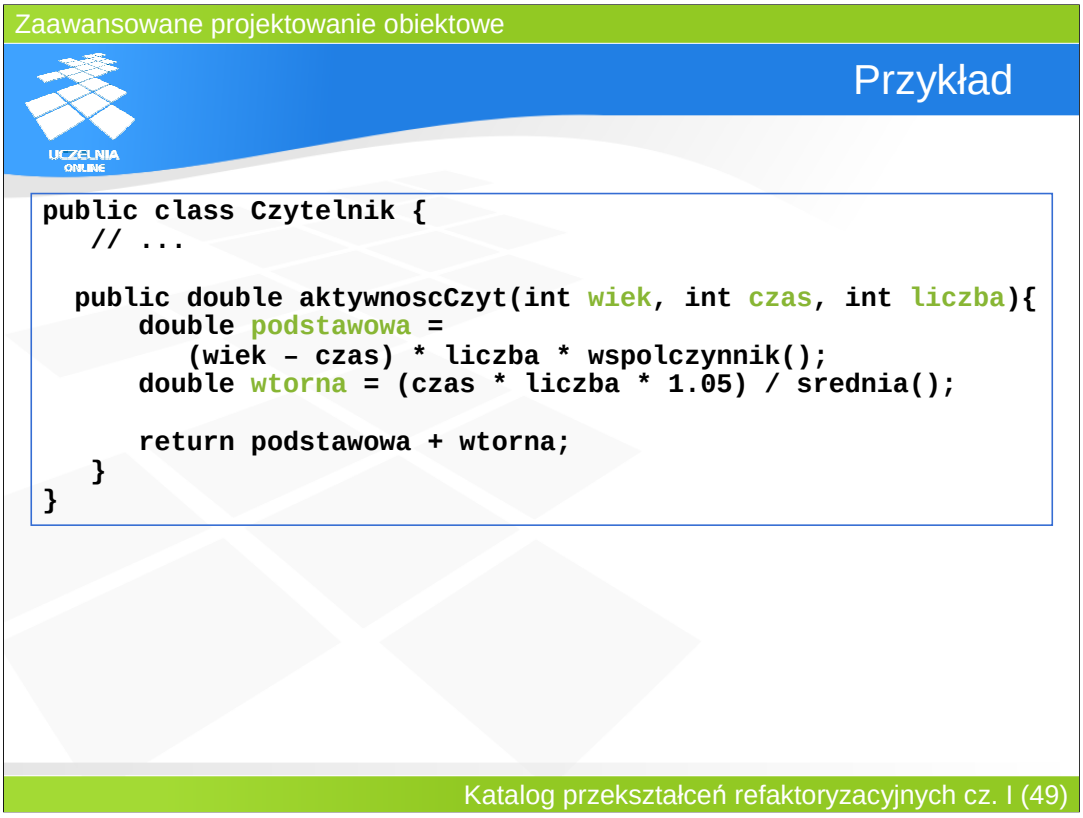
- utwórz nową klasę
- utwórz w nowej klasie sfinalizowane pole typu klasy oryginalnej
- utwórz w nowej klasie konstruktor z pierwszym parametrem typu klasy oryginalnej.
- wykonaj *Move Method*
- skompiluj
- poprzedź wywołanie starej metody utworzeniem obiektu
- podziel metodę w nowej klasie (zmienne i parametry = pola)

Przekształcenie to jest silnie związane z problemem zmiennych tymczasowych. Dotychczas podstawowym sposobem radzenia sobie z długimi metodami był ich podział poprzez wydzielanie nowych metod. Czasem jednak metoda posiada tyle zmiennych lokalnych, że ich przekazywanie do nowych metod jest w zasadzie niemożliwe.

Celem tego przekształcenia jest zmiana metody w obiekt, który umożliwi podział jej na mniejsze jednostki.

Pierwszym krokiem przekształcenia jest utworzenie nowej klasy, która będzie posiadała referencję do klasy oryginalnej (zatem konieczne jest utworzenie w niej pola odpowiedniego typu i konstruktora inicjującego to pole). Kolejną czynnością jest przeniesienie metody do nowego obiektu, a następnie z jej parametrów i zmiennych lokalnych utworzenie pól tego obiektu. Ten krok pozwala na usunięcie wszystkich zależności, które uniemożliwiały podział metody. W tym momencie wszystkie wywołania oryginalnej metody muszą zostać zastąpione utworzeniem obiektu nowej klasy i wywołaniem metody na tym obiekcie.

Dzięki tej operacji metoda o dużej złożoności może być podzielona na mniejsze metody bez konieczności przekazywania im wszystkich zmiennych lokalnych.



Jako przykład prześledźmy klasę Czytelnik, która definiuje metodę *aktywnoscCzyt()*. Posiada ona trzy parametry i definiuje trzy kolejne zmienne lokalne. W tej postaci podział tej metody jest niemożliwy.

Przykład

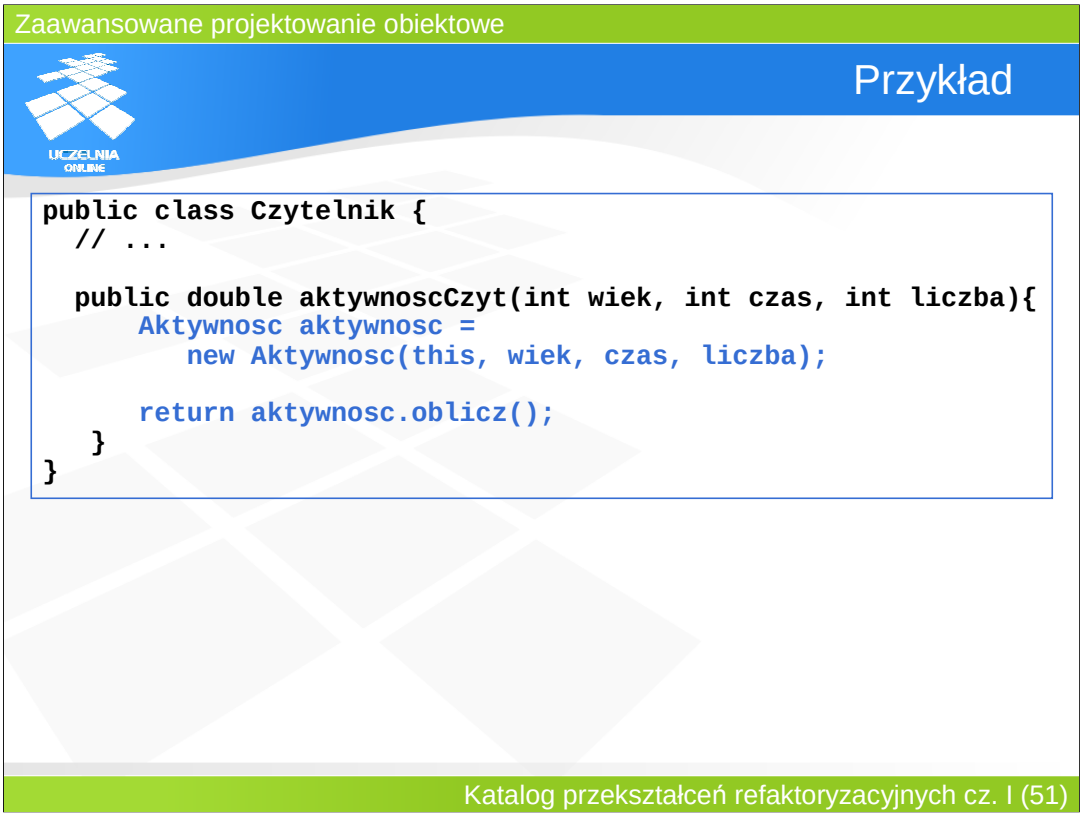
```
public class Aktywnosc {
   Czytelnik czytelnik
   int wiek;
   int czas;
   int liczba;
   double podstawowa;
   double wtorna;

   public Aktywnosc(Czytelnik cz, int w, int c, int l) {
      czytelnik = cz;
      wiek = w;
      czas = c;
      liczba = l;
   }

   public double oblicz() {
      podstawowa = (wiek – czas) * liczba * wspolczynnik();
      wtorna = (czas * liczba * 1.05) / srednia();

      return podstawowa + wtorna;
   }
}
```


Zmiana polega na utworzeniu nowej klasy Aktywność, zawierającej pola odpowiadające parametrom i zmiennym lokalnym tej metody, oraz definiującej jedno pole będące referencją do klasy, która była właścicielem modyfikowanej metody. Konstruktor inicjuje pola, dzięki czemu sama metoda *oblicz()* nie posiada żadnych parametrów i nie definiuje zmiennych lokalnych, zatem może zostać podzielona na mniejsze metody.



Wywołanie oryginalnej metody jest teraz rozszerzone o utworzenie obiektu klasy Aktywność, której przekazywane są niezbędne parametry.

Form Template Method

Problem

Metody w podklasach wykonują podobną sekwencję kroków

Cel

Ujednolić kroki i umieść ich wywołania w szablonie metody

Mechanika

- podziel metody, aż będą identyczne lub całkowicie różne
- identyczne metody przenieś do nadklasy
- różne metody pozostaw w podklasach, ujednolicając ich sygnatury
- utwórz w nadklasie abstrakcyjne metody dla metod różniących się w podklasach
- utwórz w nadklasie szablon metody, składający się z wywołań metod
- skompiluj i przetestuj

M. Fowler, 1999

Ostatnie przekształcenie z tej grupy dotyczy zaimplementowania w kodzie wzorca Template Method. Stosuje się je wówczas, gdy metody w klasach dziedziczących po wspólnej nadklasie wykonują podobną sekwencję kroków. Celem tej refaktoryzacji jest ujednolicenie wykonywanych kroków w postaci metod i umieszczenie ich wywołania w szablonie metody.

Przekształcenie rozpoczyna się od modyfikacji metod w podklasach. Za pomocą przekształceń na poziomie metody (Extract Method, Inline Temp etc.) należy doprowadzić do sytuacji, w której odpowiadające sobie metody w podklasach (o tych samych sygnaturach) będą albo identyczne bądź całkowicie różne. Metody różne należy pozostawić w podklasach i utworzyć dla nich metodę abstrakcyjną w nadklasie, którą będą pokrywać. Metody identyczne mogą zostać w całości przeniesione do nadklasy. W nadklasie można wówczas skonstruować szablon metody, składający się z wywołań metod identycznych lub abstrakcyjnych metod, które w podklasach zostały zaimplementowane w różny sposób.

Przykład

```
public class Taryfa1 {
   public double obliczKwote() {
      double abonament = 42;
      double impulsy = 0.35 * liczbaImpulsow;

      return abonament + liczbaImpulsow;
   }
}

public class Taryfa2 {
   public double obliczKwote() {
      double abonament = 65;
      double impulsy = 0.30 * liczbaImpulsow;

      return abonament + liczbaImpulsow;
   }
}
```


Przykładem takiego przekształcenia będą obiekty Taryfa1 i Taryfa2, reprezentujące plany abonamentowe operatora telekomunikacyjnego. W obu przypadkach kwota do zapłacenia składa się z abonamentu oraz opłaty za przeprowadzone rozmowy, zależnej od liczby impulsów.

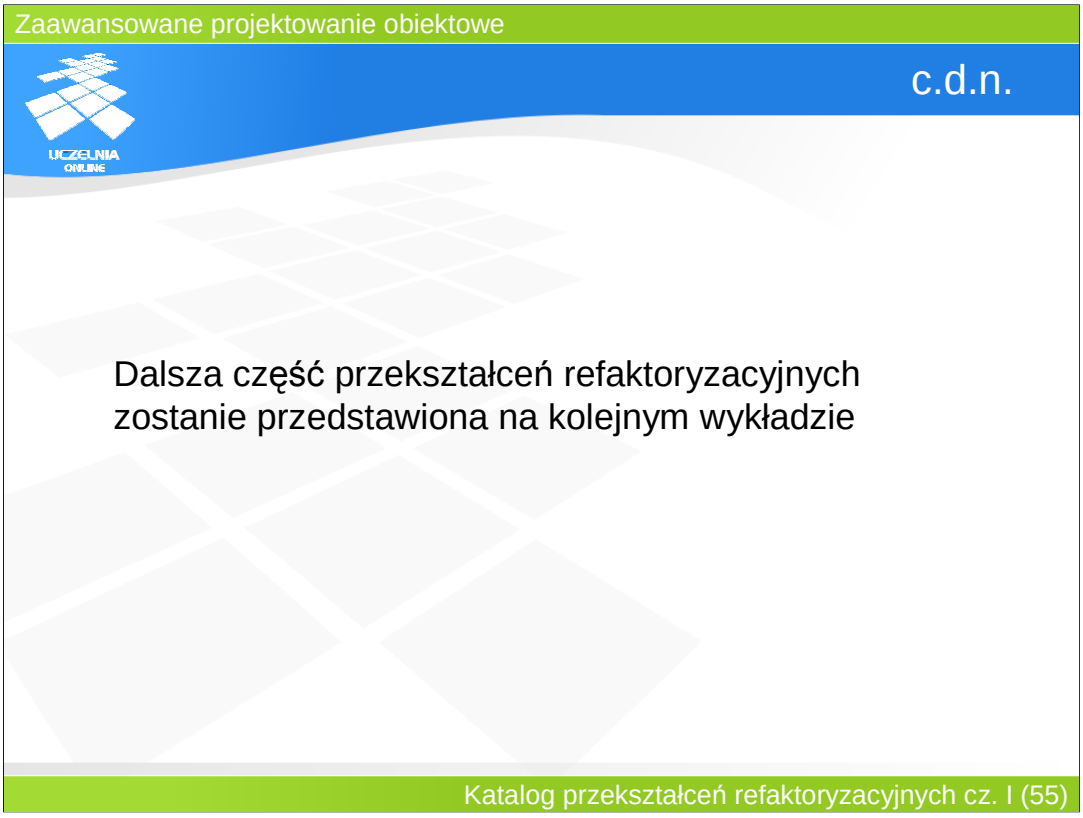
Przykład

```
abstract class Taryfa {
   public double obliczKwote(int liczbaImpulsow) {
      return abonament() + impulsy();
   }
   abstract double abonament();
   abstract double impulsy(int liczbaImpulsow);
}
public class Taryfa1 extends Taryfa {
   public double abonament() {
      return 42.0;
   }
   public double impulsy(int liczbaImpulsow) {
      return 0.35 * liczbaImpulsow;
   }
}
public class Taryfa2 extends Taryfa {
   public double abonament() {
      return 65.0;
   }
   public double impulsy(int liczbaImpulsow) {
      return 0.30 * liczbaImpulsow;
   }
}
```


W efekcie przekształcenia powstała nadklasa o nazwie Taryfa, która deklaruje abstrakcyjne metody *abonament()* oraz *implusy()*, a także posiada definicję metody-szablonu pod nazwą *obliczKwotę().* Metoda ta zwraca wynik będący sumą wyników metod *abonament()* oraz *implusy()*, pomimo faktu, że są one w tej klasie zadeklarowane jako abstrakcyjne.

Ich definicje są zawarte w podklasach Taryfa1 i Taryfa2. Utworzenie instancji jednej z tych podklas pozwala obliczyć wysokość kwoty do zapłaty za pomocą odziedziczonej metody *obliczKwotę()*, jednak wartość ta zależy od implementacji metod w podklasie.

W tym przypadku obie metody należały do kategorii metod różnych od siebie. W przypadku metody identycznej dla wszystkich podklas możliwe byłoby przeniesienie jej w całości do nadklasy.



Dalsza część katalogu zostanie przedstawiona podczas kolejnego wykładu.