Etap implementacji

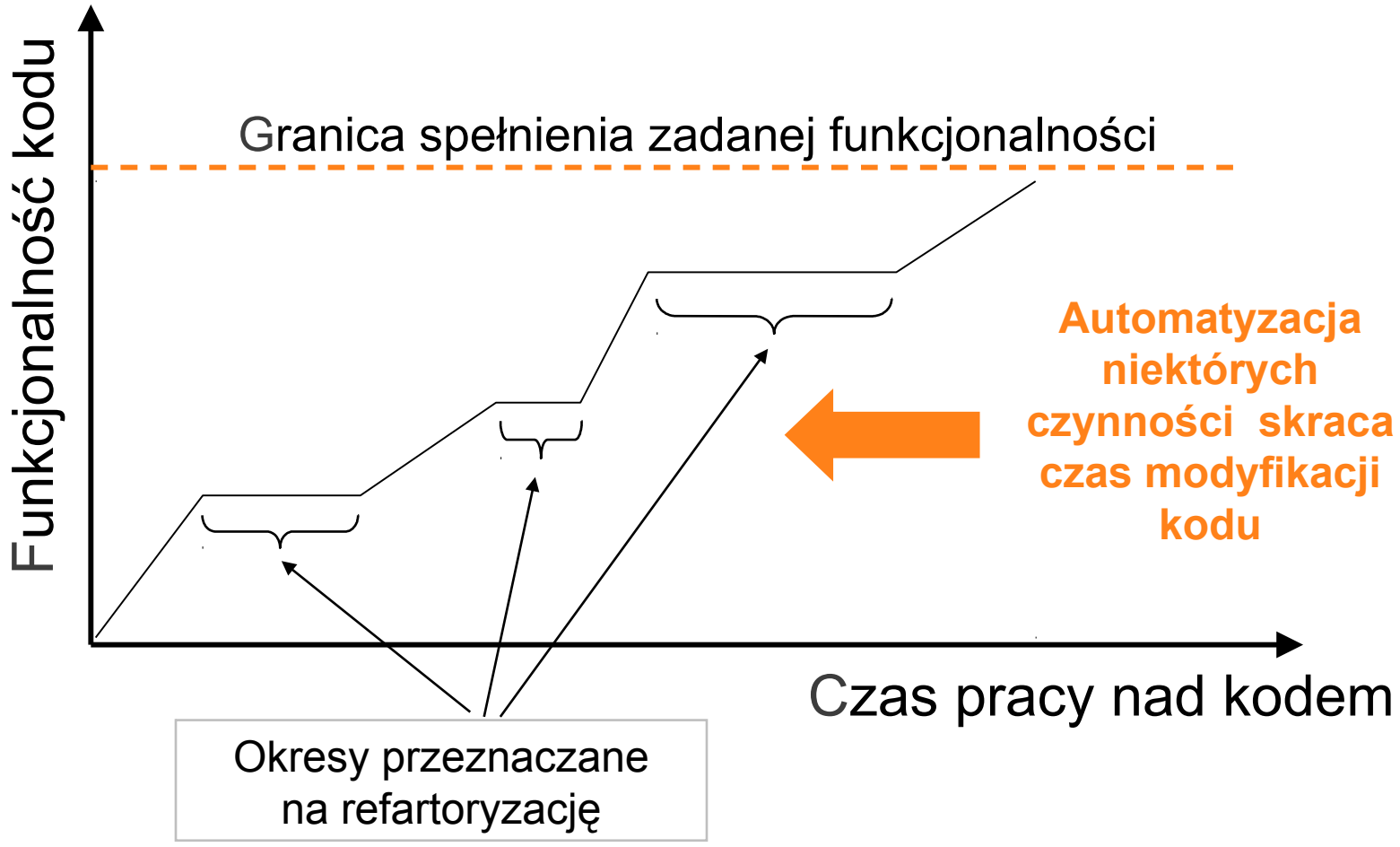
Refaktoryzacja

Koncepcja refaktoryzacji

- Termin refaktoryzacja (ang. Refactoring) definiuje się jako mechanizm zmiany struktury kodu bez zmiany jego zachowania (funkcjonalności).

- Celem refaktoryzacji jest więc nie wytwarzanie nowej funkcjonalności, ale utrzymywanie odpowiedniej, wysokiej jakości organizacji systemu.

Przebieg procesu projektowego



Korzyści z refaktoryzacji

- Ułatwienie pielęgnacji (modyfikacji) kodu.
- Zwiększenie integralności danych i zgodności ich reprezentacji z semantyką języka.
- Ułatwienie wykorzystania mechanizmów automatycznego generowania kodu.
- Ograniczenie redundancji (nadmiarowości) i narzucanie standardów budowy.
- Zachowanie silnej separacji warstw systemu (w przypadku systemów o architekturze wielowarstwowej).

Dwukierunkowe działania w ramach refaktoryzacji

- Modyfikowanie elementów systemu w celu wpasowania ich w przyjęte standardy i wzorce.
- Poszukiwanie nowych standardów i wzorców, które pojawiają się w systemie w trakcie jego rozwoju i ich precyzyjne definiowanie.

Refaktoryzacja a zarządzanie projektem informatycznym.

- Przy dużych projektach koszt refaktoryzacji jest rekompensowany obniżką kosztów wprowadzenia późniejszych globalnych zmian w projekcie.

- W przypadku projektów obarczonych dużym ryzkiem niepowodzenia (np. z powodu niestabilnych wymagań funkcjonalnych klienta) regularnie prowadzona refaktoryzacja wydaje się nieodzowną.

Istota refaktoryzacji

- Przeprowadzać zmiany w sposób bezpieczny, to znaczy utrzymujący dotychczasową funkcjonalność zmienianego fragmentu programu.

- Dwie podstawowe metody weryfikacji poprawności przekształceń:
  - analiza własności zmienianego kodu,
  - testowanie jednostkowe wprowadzonych zmian.

Analiza własności zmienianego kodu

- Sprowadza się do wyznaczenia dla każdego przekształcenia refaktoryzacyjnego warunków wstępnych oraz końcowych, które muszą być spełnione, aby zmiana była poprawna.

- Warunki te to najczęściej statyczne własności klasy lub metody (np. jej relacje dziedziczenia, powiązania z innymi elementami systemu), dlatego ich weryfikacji w większości przypadków może dokonać kompilator lub analizator składniowy.

Testowanie jednostkowe

- Jest implementowane przy użyciu bibliotek z serii xUnit, (np. JUnit).

- W istocie jest to regresyjna weryfikacja poprawności działania poszczególnych metod, czyli stwierdzenie, czy wprowadzane zmiany nie zmieniają zachowania klas.
- Testy stanowią niezmienniki przekształceń, bo muszą być spełnione zarówno przed, jak i po przeprowadzeniu zmian.

xUnit rodziną narzędzi do testów modułowych

Platforma	xUnit	URL
Java	JUnit	http://junit.org/
VBasic	VBUnit	http://www.vbunit.org
C++	CPPUnit	http://cppunit.sourceforge.net
C#	NUnit	http://www.nunit.org
Delphi	DUnit	http://dunit.sourceforge.net/
PHP	PHPUnit	http://www.phpunit.de/

Wolne narzędzia do refaktoryzacji

Platform	Tools	URL
General	X-develop	http://www.x-develop.com/
Delphi	Model Maker	http://modelmakertools.com/
Visual Basic	Refactor! for Visual Basic	http://msdn.microsoft.com/VBasic/Downloads/2005/Tools/Refactor/
C/C++	SlickEdit, Ref++, Xrefactory	http://www.slickedit.com/, http://www.ideat-solutions.com/refpp/, http://xref-tech.com/xrefactory/main.html
Java	IntelliJ Idea, Eclipse, JFactor	http://www.intellij.com/idea/, http://www.eclipse.org/, http://www.instantiations.com/jfactor

Refaktoryzacja a przykładowe IDE

- Typy refaktoryzacji dostępne w Eclipse:
  - Na poziomie struktury fizycznej projektu (Physical Structure).
  - Na poziomie struktury klas (Class Level Structure).
  - Na poziomie wewnętrznej budowy klasy (Structure inside a Class).

Metody „Physical Structure”

- Rename
- Move
- Change Method Signature
- Convert Anonymous Class to Nested
- Convert Nested Type to Top Level (Eclipse 2)
- Move Member Type to New File (Eclipse 3)

Metody „Class Level Structure”

- Push Down
- Pull Up
- Extract Interface
- Generalize Type (Eclipse 3)
- User Supertype Where Possible

Metody „Structure inside a Class”

- Inline
- Extract Method
- Extract Local Variable
- Extract Constant
- Introduce Parameter (Eclipse 3 only)
- Introduce Factory (Eclipse 3 only)
- Encapsulate Field

Ocena jakości kodu

- „Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

- “Przykry zapach” jest symptomem niskiej jakości kodu, która wskazuje na konieczność refaktoryzacji.

Martin Fowler
„Refactoring, Improving the design of existing code”, Addison-Wesley, 1999

Zduplikowany kod (Duplicated Code)

- Objawy: ten sam lub podobny kod pojawia się w różnych miejscach systemu.

- Rozwiązania:
  - Jedna klasa: wyłączenie wspólnych części do jednej metody (Extract Method).
  - Bliźniacze klasy: wyłączenie metody ze wspólną funkcjonalnością i następnie podniesienie (Pull-up the Method) do wspólnej nadklasy.
  - Niespokrewnione klasy: wyłączenie klasy (Extract Class) ze wspólnymi elementami i delegowanie do niej.

Duża metoda (Long Method)

- Objawy:
  - Metoda wykonuje wiele różnych czynności.
  - Brak wsparcia ze strony innych metod, przez co niektóre metody operują na niższym poziomie niż powinny.
  - Nadmiernie skomplikowana obsługa wyjątków.

- Rozwiązania:
  - Extract Method.
  - Wyłączenie tymczasowych zmiennych do zewnętrznych metod (Replace Temp with Query).
  - Zmniejszenie liczby parametrów (Introduce Parameter Object or Preserve Whole Object).
  - Stworzenie nowej klasy z metody i podział funkcionalności (Replace Method with Method Object).

Duża klasa (Large class)

- Objawy:
  - Klasa posiada zbyt duży zakres odpowiedzialności.
  - Występują liczne klasy wewnętrzne i instancje klasy.
  - Nadmierna liczba metod ułatwiających.

- Rozwiązania
  - Ekstrakcja klasy w celu podziału klasy przez referencje.
  - Ekstrakcja podklasy (Subclass) w celu podziału klasy przez dziedziczenie.
  - Ekstrakcja interfejsu w celu podziału klasy przez polimorfizm.
  - Przesunięcie części zadań do nadklasy (Superclass).

Leniwa klasa (Lazy Class)

- Objawy: klasa ma znikomą odpowiedzialność lub nie ma odpowiedzialności wcale.

- Rozwiązania:
  - Wchłonięcie przez nadklasę/podklasę (Collapse Hierarchy).
  - Wchłonięcie klasy przez referencje (Inline Class).

Klasa danych (Data Class)

- Objawy: klasa jest odpowiedzialna jedynie za przechowywanie danych i nie posiada istotnych metod (Transfer Object).

- Rozwiązania:
  - Wzbogacenie funkcjonalności (Extract Method i Move Method).
  - Usunięcie klasy (Inline Class).

Instrukcje przełączające (Switch Statements)

- Objawy: metoda posiada rozbudowany przepływ sterowania, oparty na złożonych wyrażeniach warunkowych.

- Rozwiązania:
  - Wyłączenie wspólnych części do osobnych metod w tej samej klasie (Extract Method).
  - Użycie polimorfizmu (Replace Conditional with Polymorphism/State).
  - Użycie dziedziczenia (Replace Conditional with Subclasses).

Odrzucony spadek (Refused Bequest)

- Objawy: podklasy nie potrzebują odziedziczonych metod i pól.

- Rozwiązania:
  - Utworzenie nowej klasy bliźniaczej (Push Down Method and Push Down Field).
  - Zastąpienie dziedziczenia delegacją (Replace Inheritance with Delegation).

Niewłaściwa hermetyzacja (Inappropriate Intimacy)

- Objawy: bezpośredni dostęp do wewnętrznych pól innej klasy.

- Rozwiązania:
  - Przesunięcie pól/metod do właściwej klasy (Move Method/ Move Field).
  - Ograniczenie dwukierunkowych relacji (Change Bidirectional Association to Unidirectional Association).
  - Wyłączenie elementów o wspólnym dostępie do osobnej klasy (Extract Class).
  - Zastąpienie dziedziczenia delegacją.

Zazdrość o funkcję (Feature Envy)

- Objawy:
  - Metoda jednej klasy intensywnie korzysta z funkcji innej klasy.
  - Niska spójność klasy.

- Rozwiązania
  - Przesunięcie metody do właściwej klasy (Move Method).
  - Użycie wzorca Visitor.

Łańcuchy wywołań metod (Message Chains)

- Objawy: długie łańcuchy wywołań getAnotherObject().

- Rozwiązania:
  - Ukrycie klas delegujących (Hide Delegate).
  - Przesunięcie niektórych metod w dół łańcucha (Extract Method and Move Method).

Chirurgia śrutówką (Shotgun Surgery)

- Objawy: wprowadzona zmiana wymaga modyfikacji wielu innych klas.

- Rozwiązania: przesunięcie wszystkich obszarów zmienności do jednej klasy (Move Method and Move Field).

Przekształcenia refaktoryzacyjne

Przykłady

Przykład: Rename Method

```
class ClassA
  void Name_1 ()
{
    // some code
  }
}
```

```
class ClassA
  void Name_2 ()
{
    // some code
  }
}
```

Warunki wstępne:

- w klasie *ClassA* nie istnieje metoda *void Name_2()*
- klasa *ClassA* nie dziedziczy metod *Name_1()* ani *Name_2()*

Warunki końcowe:

- klasa *ClassA* nie posiada metody *Name_1()*
- klasa *ClassA* posiada metodę *Name_2()*

Przykład: Extract Method

```
void scalarProduct(String[]
params) {
   int[] x = prepareX(params);
   int[] y = prepareY(params);
   int[] product =
computeXY(x, y);
   // ...
   for (i = 0; i < x.length; i++)
{
      out.println("X = " + x[i]);
      out.println("Y = " + y[i]);
      out.println("X * Y = " +
product[i]);
   }
}
```

```
void scalarProduct(String[]
params) {
   int[] x = prepareX(params);
   int[] y = prepareY(params);
   int[] product = computeXY(x, y);
   // ...
   printScalarProduct(x, y,
product);
}
```

```
void printScalarProduct(int[] x, int[]y,
int[] product) {
  for (i = 0; i < x.length; i++) {
      out.println("X = " + x[i]);
      out.println("Y = " + y[i]);
      out.println("X * Y = " + product[i]);
   }
}
```

Warunki wstępne:

- wskazany blok modyfikuje co najwyżej jedną zmienną
- w klasie nie istnieje dotychczas metoda *printScalarProduct()*
- klasa nie dziedziczy metod *scalarProduct()* ani *printScalarProduct()*

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.*

Przykład

```
public class Student {
    Collection wyklady;
    public Collection wyklady() {
        return wyklady;
    }
}
```

```
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);
    }
}
```

Wprowadź Null Object

M. Fowler, 1999

Problem

Referencje do klasy wymagają ciągłego porównywania z wartością *null*

Cel

Wprowadzenie obiektu reprezentującego wartość *null*

Mechanika

- utwórz podklasę reprezentującą wartość *null* o nazwie *NullKlasa,*
- utwórz w klasie i podklasie metodę *isNull(); w* klasie metoda zwraca *false*, w podklasie – *true,*
- skompiluj klasy,
- zastąp u klientów referencje *null* klasy instancjami podklasy,
- zastąp warunki sprawdzające referencję *null* wywołaniem *isNull(),*
- pokryj metody w podklasie zgodnie z semantyką obiektu reprezentującego *null*, usuń metody *isNull().*

Przykład

```
public class Student {
    Collection wyklady;
    public Collection wyklady() {
        return wyklady;
    }
}
```

```
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);
    }
}
```

Przykład

```
public class Student {
    Collection wyklady;
    public Collection wyklady() {
        return wyklady;
    }
}
```

```
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);
    }
}
```