Dr Katarzyna Grzesiak-Kopeć

Inżynieria oprogramowania

5. Refaktoryzacja

Plan wykładu

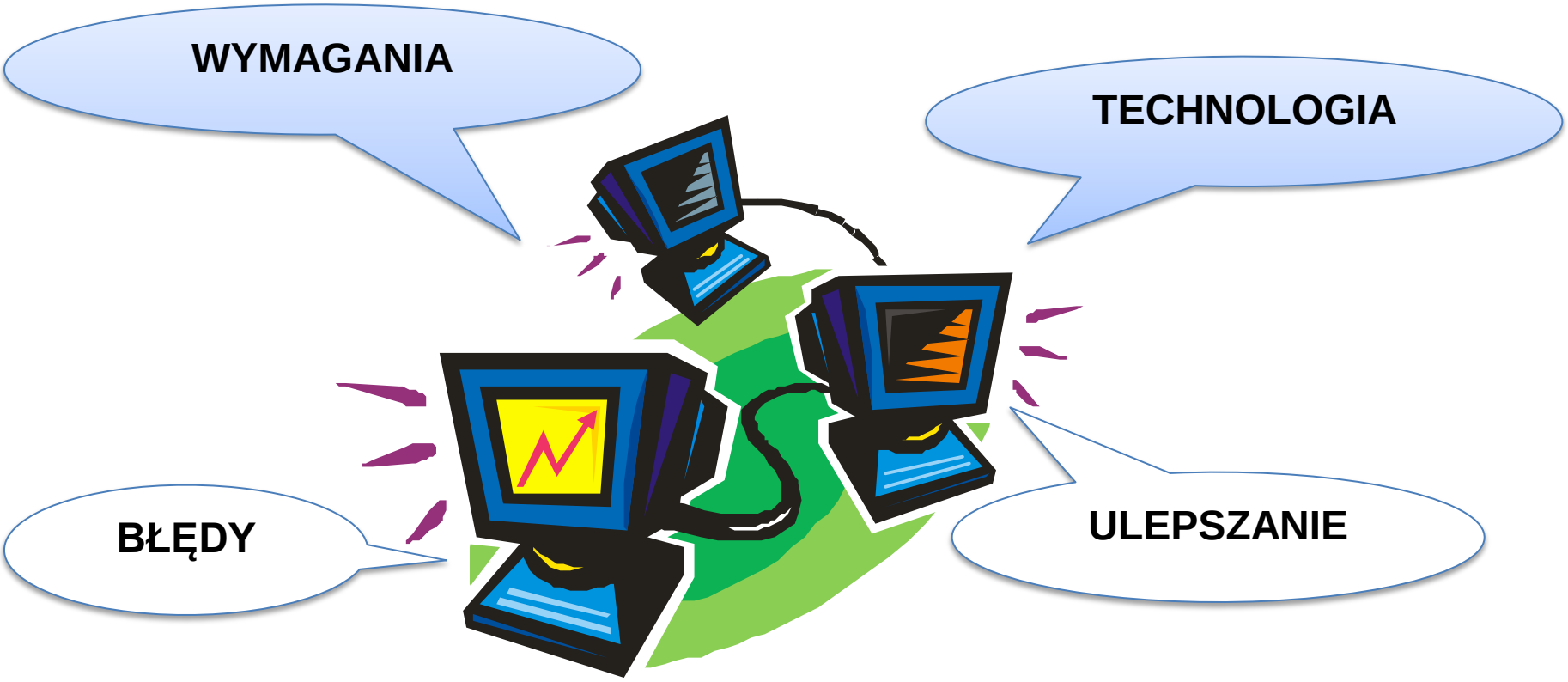
- Tworzenie oprogramowania
- Najlepsze praktyki IO
- Inżynieria wymagań
- Technologia obiektowa i język UML
- Techniki IO
- Metodyki zwinne
- Refaktoryzacja
- Mierzenie oprogramowania
- Jakość oprogramowania
- Programowanie strukturalne
- Modelowanie analityczne
- Wprowadzenie do testowania

Zmiany to norma!

B. Du Bois, Refactoring

- Zmiany zewnętrzne
  - Środowiska
  - Wymagań
  - Technologii
  - Zespół utrzymania
- Zmiany wewnętrzne
  - Rozmiar
  - Złożoność
  - Zależności

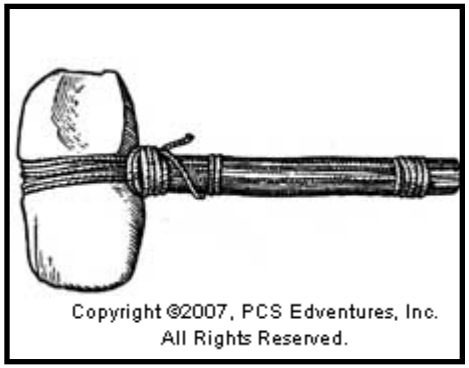
Ewolucja oprogramowania



B. Walter, Ewolucja oprogramowania i refaktoryzacja

Zmiana wymagań

- Oprócz błędów i niezrozumienia…
- Wymagania na dzisiaj, a nie na jutro



Zespół utrzymania

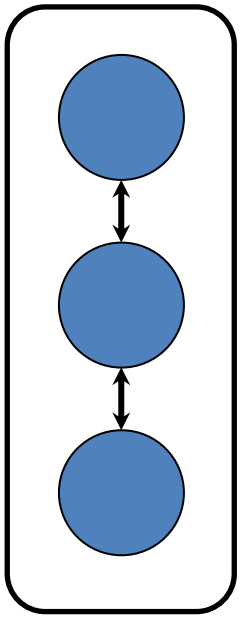
- Oprogramowanie działa przy licznych domyślnych założeniach
- Założenia te są w głowach deweloperów
  - Strata ludzi ⇨ strata wiedzy
  - Wiedza o procesie
  - Wiedza o produkcie

Zmiana technologii

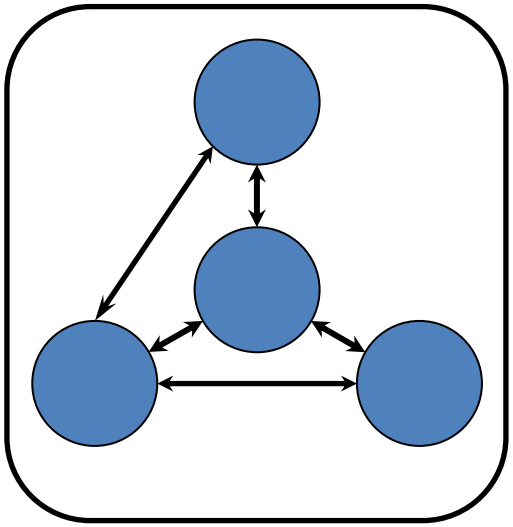
- Bazy danych
  - Płaskie pliki
  - Hierarchiczne bazy danych
  - Relacyjne bazy danych
  - …
- Paradygmat programowania
  - Proceduralne
  - Obiektowe
  - Z komponentów

Zmiany wewnętrzne

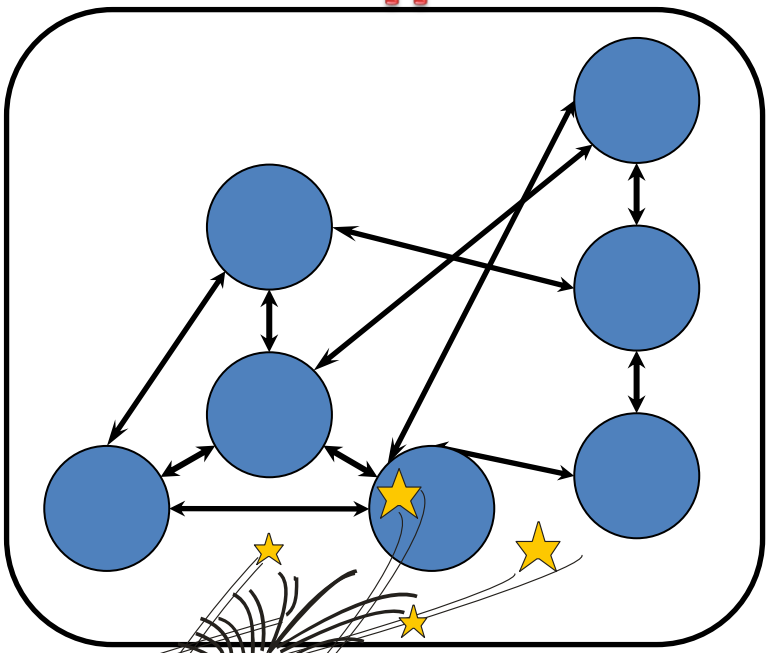
I

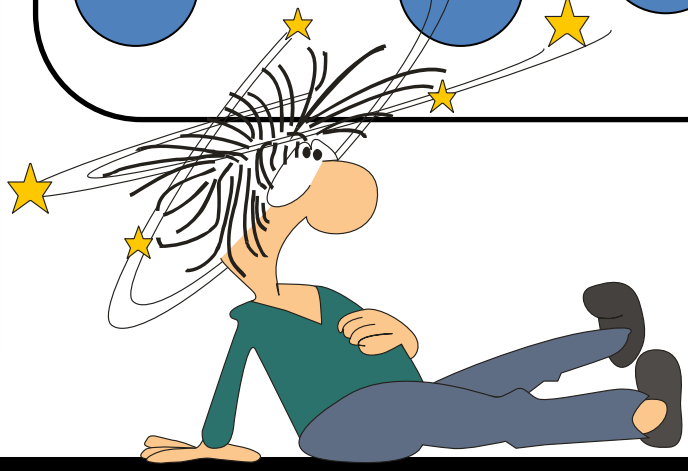


II



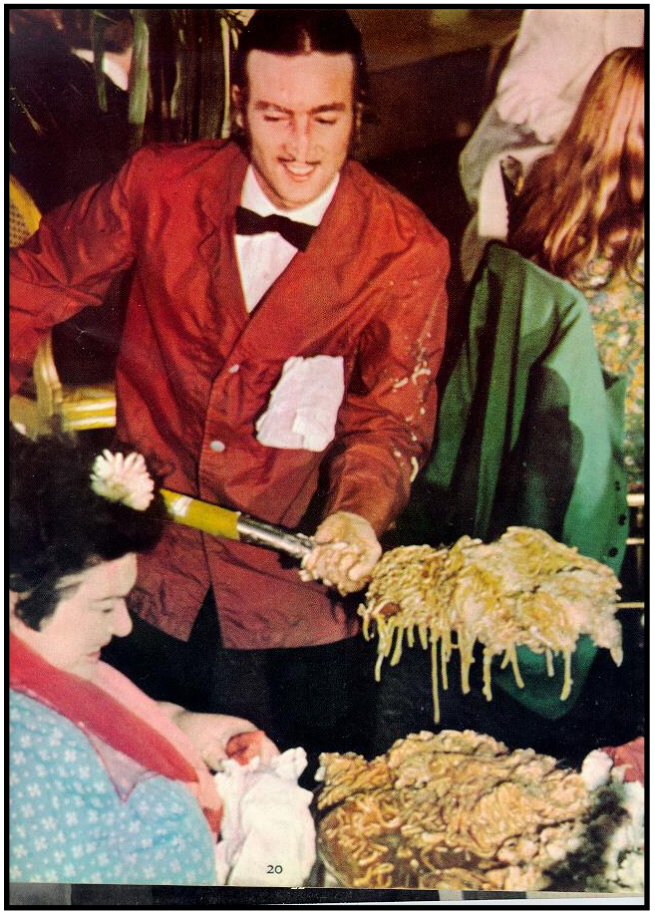
n





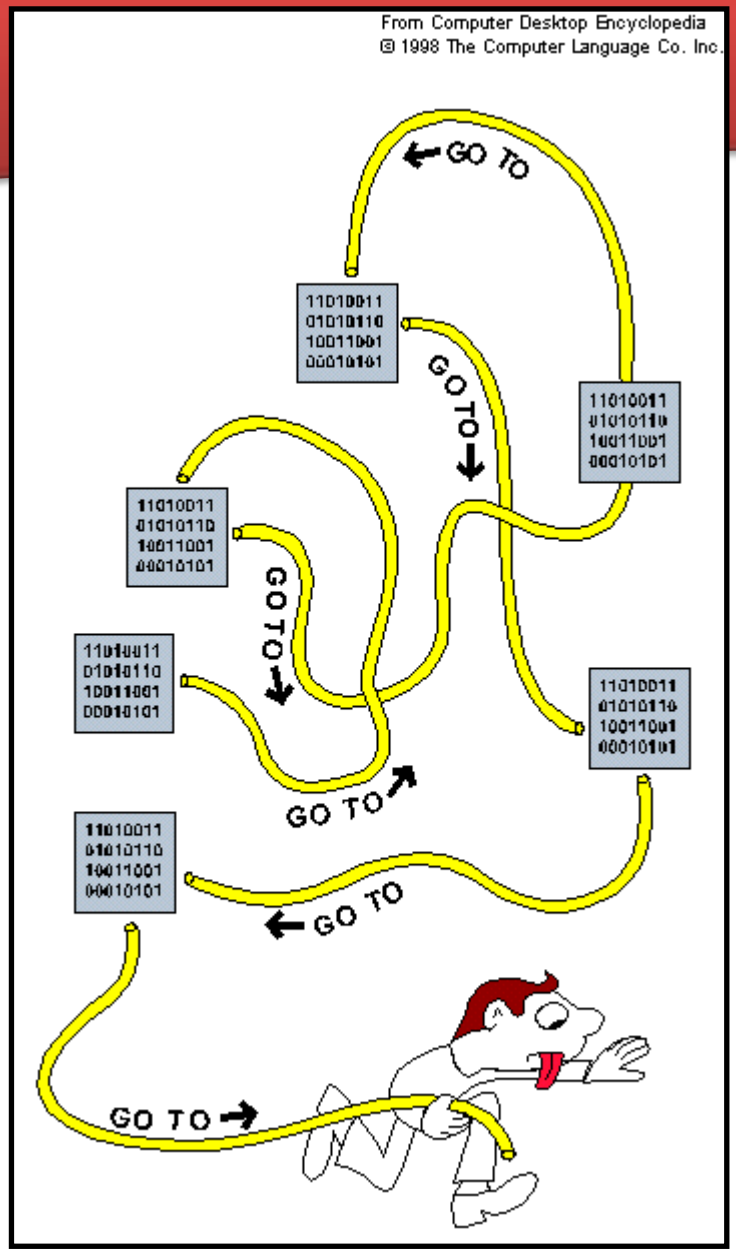
Spaghetti code

- Określenie pejoratywne
- Kod trudny do zrozumienia, utrzymania i poprawy
- Kod zawiera wiele instrukcji typu
  - GOTO,
  - wyjątki lub inne „nieustrukturyzowane” instrukcje rozgałęziające wątek kontroli



Spaghetti code

- the Pasta Theory of Programming
  - Lasagna code – programowanie strukturalne
  - Ravioli code – programowanie zorientowane obiektowo



Spaghetti code

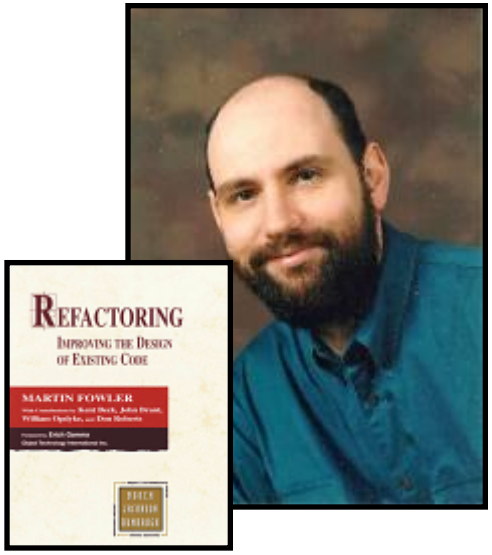

```
10 i = 0
20 i = i + 1
30 if i <= 10 then goto 80
40 if i > 10 then goto 60
50 goto 20
60 print "Program Completed."
70 end
80 print i; " squared = "; i * i
90 goto 20
```

Podsumowanie

- Liniowo rosnące zmiany zewnętrzne mogą spowodować eksponencjalny przyrost zmian wewnętrznych
- Jak sobie z tym radzić?
  - Zmiany zewnętrzne
  - Zmiany wewnętrzne
- Ucz się z nich – CODE SMELLS
- Znajdź je - REFACTORING

Refaktoryzacja - podstawa ☺

- William F. (Bill) Opdyke
- Martin Fowler
- Kent Beck



Refaktoryzacja

Zmiana *organizacji struktury* kodu programu nie zmieniająca jego działania .

$$x^2 - x - 2$$

$$(x + 1)(x - 2)$$

Refaktoryzacja

Refactoring by Martin Fowler, 1999

- Rzeczownik

> **A studied, described, and catalogued coding technique for changing a small part of the design of code without changing the external behavior of the code.**

- Czasownik

> **A strategy for code maintenance by means of applying refactorings freely whenever a need is discovered.**
> **A separate activity from adding new functionality.**

Po co refaktoryzacja?

- Ulepsza projekt oprogramowania
  - Często prowizorycznie zmieniamy kod
- Łatwiej zrozumieć oprogramowanie
  - Czytelniejszy kod
- Pomaga lokalizować błędy
  - Czyni strukturę czytelniejszą odkrywając błędy
- Pomaga szybciej programować
  - Co wynika z poprzednich punktów

Jakie języki?

- Wszystkie
- Obiektowe
  - Można refaktoryzować na dużo więcej sposobów

Kiedy refaktoryzować?

- Przed dodaniem funkcjonalności
- Kiedy trzeba poprawić błędy
- W trakcie inspekcji kodu
- Jeśli robisz to samo po raz trzeci
- Kiedy trafiasz na **bad smell**

Bad smells

- Powtarzający się kod
  - Stwórz metodę
- Długa lista parametrów
  - Wprowadź obiekt jako parametr
- Instrukcje **switch**
  - Zastosuj polimorfizm
- I mnóstwo innych
  - http://www.refactoring.com/catalog/index.html

Jak refaktoryzować?

- Rozpoczyna się od zaprojektowania dużej liczby testów dla kodu, który ma być poddany analizie
- Identyfikacja problemów – **bad smells**
- Refaktoryzacja i testowanie
- Program nie przejdzie testów ⇨ zaczynamy od nowa!

Kroki refaktoryzacji

- Małe
  - Projektuj mało, koduj mało, zmieniaj mało, testuj
  - Na tyle małe, by przewidzieć ich konsekwencje
- Odtwarzalne
  - By inni mogli je zrozumieć
- Ogólne
  - Reguły, które można stosować
- Spisane
  - Wiemy co, krok po kroku, i gdzie było zastosowane
- Zaczynamy od elementów najbardziej ryzykownych

Charakterystyka

M. Stal, Refactoring

- Nazwa
  - Extract Method
- Opis (sytuacji)
  - Fragment kodu, który może zostać wydzielony. Uczyń z tego kodu metodę i odpowiednio ją nazwij.
- Motywacja
  - zastosuj Extract Method kiedy istniejące metody są zbyt długie lub istnieją w kodzie powtórzenia

Charakterystyka

- Mechanizm
  - Utwórz nową metodę i odpowiednio ją nazwij
  - Skopiuj kod do tworzonej metody
  - Ustal zmienne lokalne
  - Jeśli zmieniasz jedną zmienną lokalną, to uczyń ją wartością zwracaną, jeśli więcej, to może konieczna jest dalsza refaktoryzacja
  - Zmienne lokalne, których nie zmieniasz, uczyń parametrami wejściowymi
  - Skompiluj
  - Zastąp w kodzie źródłowym skopiowany kod odpowiednim wywołaniem nowej metody
  - Skompiluj i przetestuj

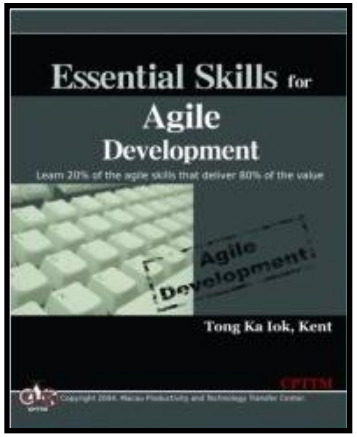
Charakterystyka

- Przykład

- Refaktoryzację można traktować jak wzorzec
- Większość jest odwracalna

Teraz przykłady

- Essential Skills for Agile Development
  - http://agileskills.org/
- Refactorings in Alphabetical Order
  - http://www.refactoring.com/catalog/index.html



Extract Method

REFACTORING

- Fragmenty kodu, które można połączyć w grupę
- Utwórz metodę, której nazwa wyjaśnia cel metody

```
void printOwing() {
        printBanner();

        //print details
        System.out.println ("name:       " + _name);
        System.out.println ("amount     " + getOutstanding());
}
```

```
void printOwing() {
        printBanner();
        printDetails(getOutstanding());
}

void printDetails (double outstanding) {
        System.out.println ("name:       " + _name);
        System.out.println ("amount     " + outstanding);
}
```

Rename Method

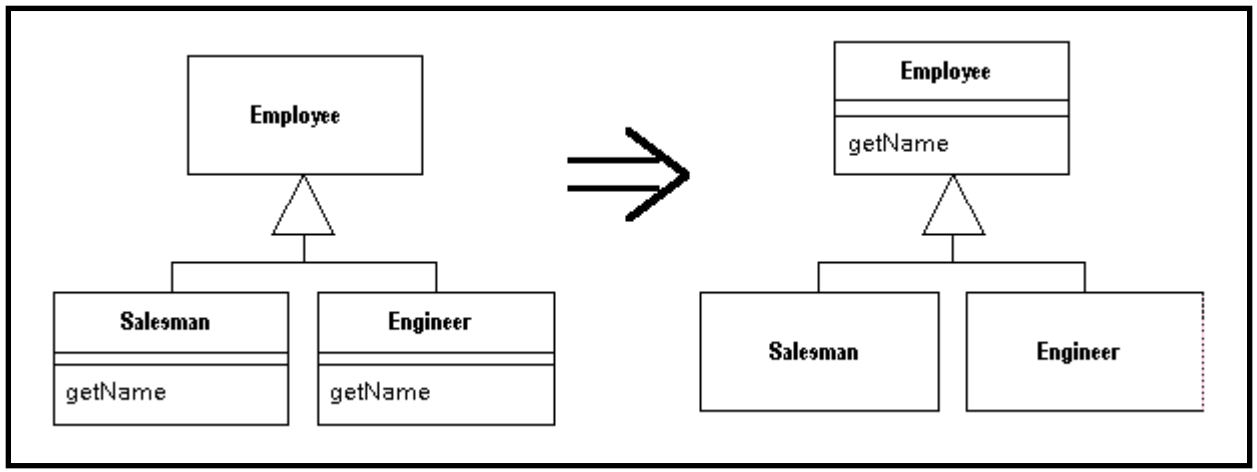
- Nazwa metody nic nie mówi o jej działaniu
- Zmień nazwę

```
//find the ID of the organization employing this participant
String oid = orgsInDB.getOrganization(participantId);
```

```
String oid = orgsInDB.findOrganizationEmploying(participantId);
```

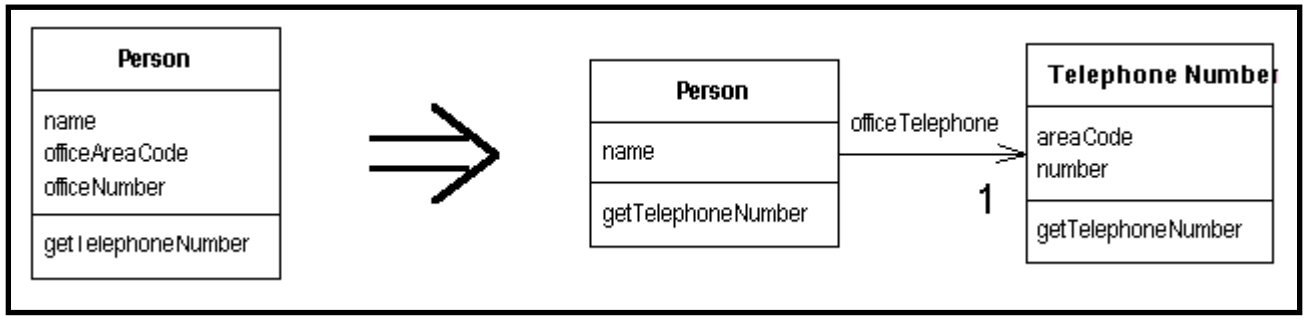
Pull Up Method

- Są metody robiące to samo w podklasach
- Przesuń je w górę hierarchii



Extract Class

- Jedna klasa „robi” to, co powinny dwie
- Stwórz nową klasę i przenieś do niej odpowiednie atrybuty oraz metody

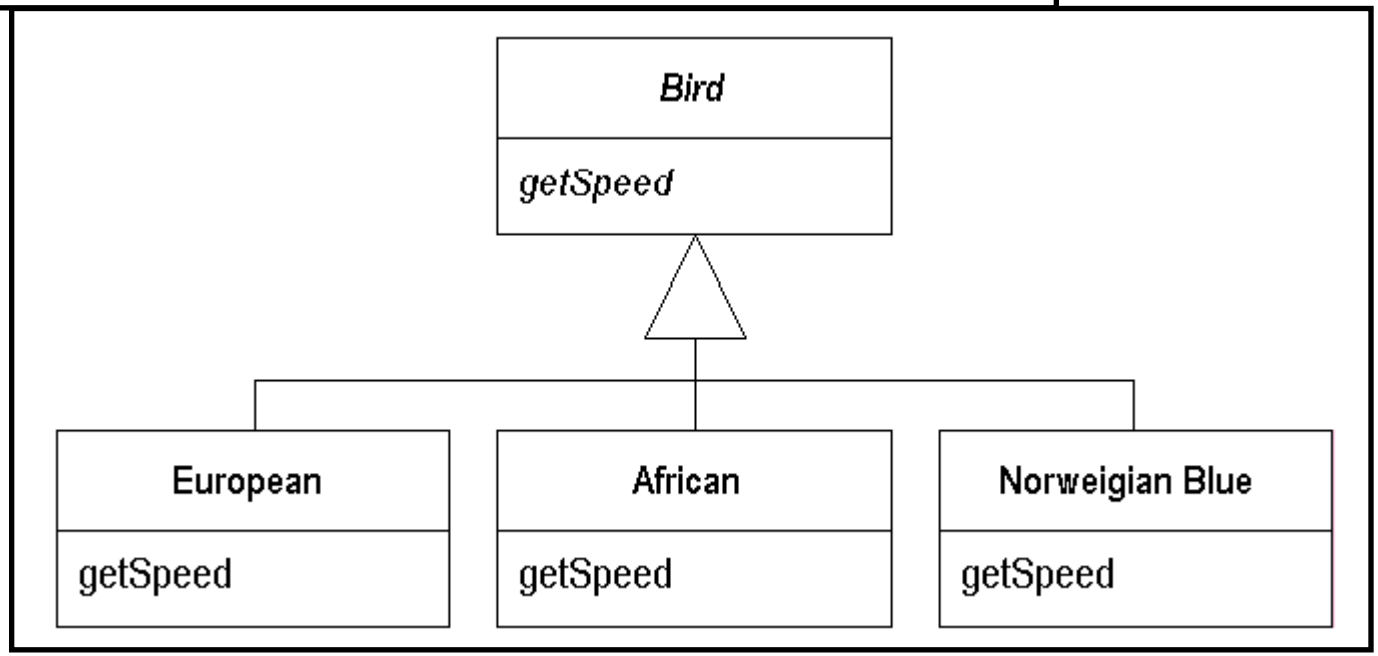


Replace Conditional with Polymorphism

- Tzw. **type-code**
- Przenieś każdy z warunków do *przeciążonej* metody w podklasie
- Z oryginalnej metody zrób *abstrakcyjną*

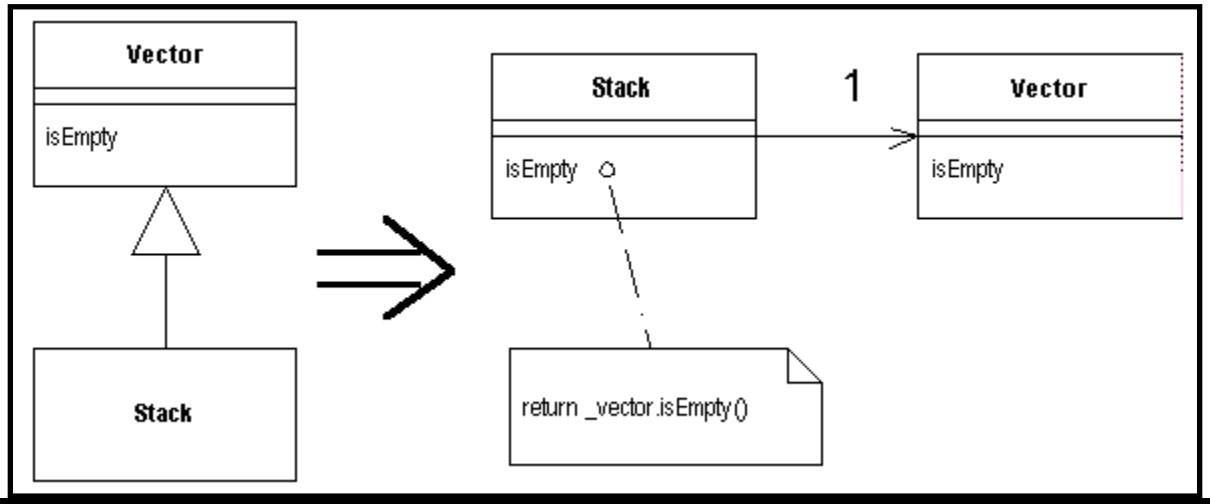
Conditional with Polymorphism

```
double getSpeed() {
        switch (_type) {
                case EUROPEAN:
                        return getBaseSpeed();
                case AFRICAN:
                        return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
                case NORWEIGIAN_BLUE:
                        return (_isNailed) ? 0 : getBaseSpeed(_voltage);
        }
        throw new RuntimeException ("Should be unreachable");
}
```

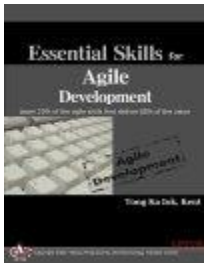


Replace Inheritance with Delegation

- Podklasa wykorzystuje tylko część interfejsu klasy nadrzędnej albo nie chce dziedziczyć danych
- Utwórz pole dla klasy nadrzędnej, dostosuj metody delegowane i usuń dziedziczenie



Code Smells




```
class Shape {
   final static int TYPELINE = 0;
   final static int TYPERECTANGLE = 1;
   final static int TYPECIRCLE = 2;
   final static int TYPETRIANGLE = 3;
   int shapeType;
   Point p1;
   Point p2;
   //third point of the triangle.
   Point p3;
   int radius;
}
class CADApp {
   void drawShapes(Graphics graphics, Shape shapes[]) {
      for (int i = 0; i < shapes.length; i++) {
         switch (shapes[i].getType()) {
            case Shape.TYPELINE:
               graphics.drawLine(shapes[i].getP1(), shapes[i].getP2());
               break;
            case Shape.TYPERECTANGLE:
               //draw the four edges.
               graphics.drawLine(...);
               graphics.drawLine(...);
               graphics.drawLine(...);
               graphics.drawLine(...);
               break;
            case Shape.TYPECIRCLE:
               graphics.drawCircle(shapes[i].getP1(), shapes[i].getRadius());
               break;
            case Shape.TYPETRIANGLE:
               graphics.drawLine(shapes[i].getP1(), shapes[i].getP2());
               graphics.drawLine(shapes[i].getP2(), shapes[i].getP3());
               graphics.drawLine(shapes[i].getP3(), shapes[i].getP1());
               break;
         }
      }
   }
}
```

Jak usunąć „type-code”?

```
class Shape {
}
class Line extends Shape {
   Point startPoint;
   Point endPoint;
}
class Rectangle extends Shape {
   Point lowerLeftCorner;
   Point upperRightCorner;
}
class Circle extends Shape {
   Point center;
   int radius;
}
```

```
class CADApp {
   void drawShapes(Graphics graphics, Shape shapes[]) {
      for (int i = 0; i < shapes.length; i++) {
         if (shapes[i] instanceof Line) {
            Line line = (Line)shapes[i];
            graphics.drawLine(line.getStartPoint(),line.getEndPoint());
         } else if (shapes[i] instanceof Rectangle) {
            Rectangle rect = (Rectangle)shapes[i];
            graphics.drawLine(...);
            graphics.drawLine(...);
            graphics.drawLine(...);
            graphics.drawLine(...);
         } else if (shapes[i] instanceof Circle) {
            Circle circle = (Circle)shapes[i];
```

Jak usunąć „if-then-else”?

```
class CADApp {
   void drawShapes(Graphics graphics, Shape shapes[]) {
      for (int i = 0; i < shapes.length; i++) {
         shapes[i].draw(graphics);
      }
   }
}
```

```
abstract class Shape {
   abstract void draw(Graphics graphics);
}
class Line extends Shape {
   Point startPoint;
   Point endPoint;
   void draw(Graphics graphics) {
      graphics.drawLine(getStartPoint(), getEndPoint());
   }
}
class Rectangle extends Shape {
   Point lowerLeftCorner;
   Point upperRightCorner;
   void draw(Graphics graphics) {
      graphics.drawLine(...);
      graphics.drawLine(...);
      graphics.drawLine(...);
      graphics.drawLine(...);
   }
}
class Circle extends Shape {
   Point center;
   int radius;
   void draw(Graphics graphics) {
      graphics.drawCircle(getCenter(), getRadius());
   }
}
```

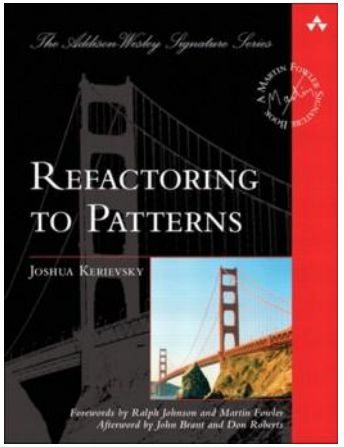
Ostatecznie…

```
interface Shape {
   void draw(Graphics graphics);
}
class Line implements Shape {
   ...
}
class Rectangle implements Shape {
   ...
}
class Circle implements Shape {
   ...
}
```

```
class CADApp {
   void drawShapes(Graphics graphics, Shape shapes[]) {
      for (int i = 0; i < shapes.length; i++) {
         shapes[i].draw(graphics);
      }
   }
}
```

Refaktoryzacja i wzorce

- Mariaż refaktoryzacji i wzorców projektowych
- Stosowanie wzorców
  - By ulepszyć projekt jest lepsze niż stosowanie ich w początkowej fazie projektowania
- Refaktoryzacja
  - Projekt może być ulepszony przez refaktoryzację wprowadzającą wzorzec projektowy



Problemy

- Czas!
- Czasem lepsza jest reinżynieria (przebudowa systemu)
- Trudno wybrać właściwy sposób rozwiązania
- Jak przewidzieć wpływ zmian lokalnych na całą architekturę?
- Stąd
  - Należy szkolić deweloperów
  - Doświadczenie pomaga
  - Kluczowe jest wsparcie narzędziowe

Koszty

- Język/środowisko
  - Zależy od narzędzi
  - Generalnie – znośny ☺
- Testowanie
  - Znowu – automatyczne testy!
- Dokumentacja
  - Może być dużo zmian!
- System
  - Zmiana interfejsów i odpowiedzialności – zmiana testów ⇨ Koszty

KONIEC