# Pielęgnacja kodu: refaktoryzacja

Jacek Starzyński, ZETiIS PW

# Plan wykładu

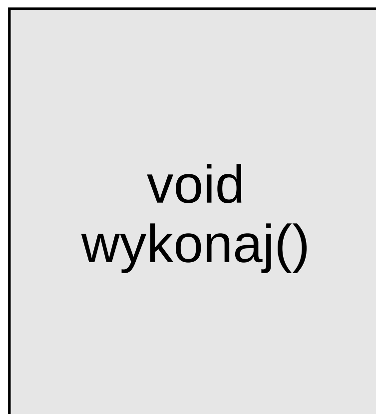
- Wprowadzenie
- Poprawność i jej weryfikacja
- Przykre zapachy w kodzie programu
- Refaktoryzacje

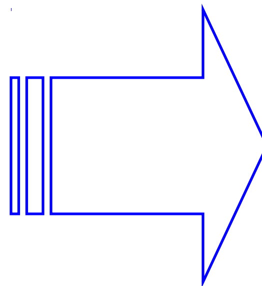
# Wprowadzenie: po co?

- Wysoki koszt pielęgnacji oprogramowania
  - Yourdon: do 80% kosztu użytkowania
  - Boehm: wytworzenie linii kodu: $30, pielęgnacja: $4000
- Naturalny wzrost złożoności i entropii oprogramowania
- Prawa Lehmana: konieczna ciągła restrukturyzacja, bez niej wzrasta złożoność, a zmniejsza się jakość oprogramowania

# Definicja

- **Refaktoryzacja to:**
- zmiana wewnętrznej struktury kodu programu,
- która zwiększa jego czytelność i obniża koszt pielęgnacji,
- ale nie zmienia jego obserwowalnego zachowania



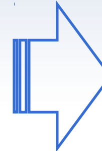



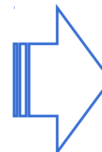
# Przykład: Extract Method

**Wyłączenie metody**

```
int dotProdFrmString(String[] parms) {
   int[] x = prepareX(parms);
   int[] y = prepareY(parms);

   int res= 0;
   for (int i = 0; i < x.length; i++) {
      res += x[i]*y[i];
   }
   return res;
}
```





```
int dotProdFromString(String[] parms) {
   int[] x = prepareX(params);
   int[] y = prepareY(params);
   return dotProd( x, y );
}
```

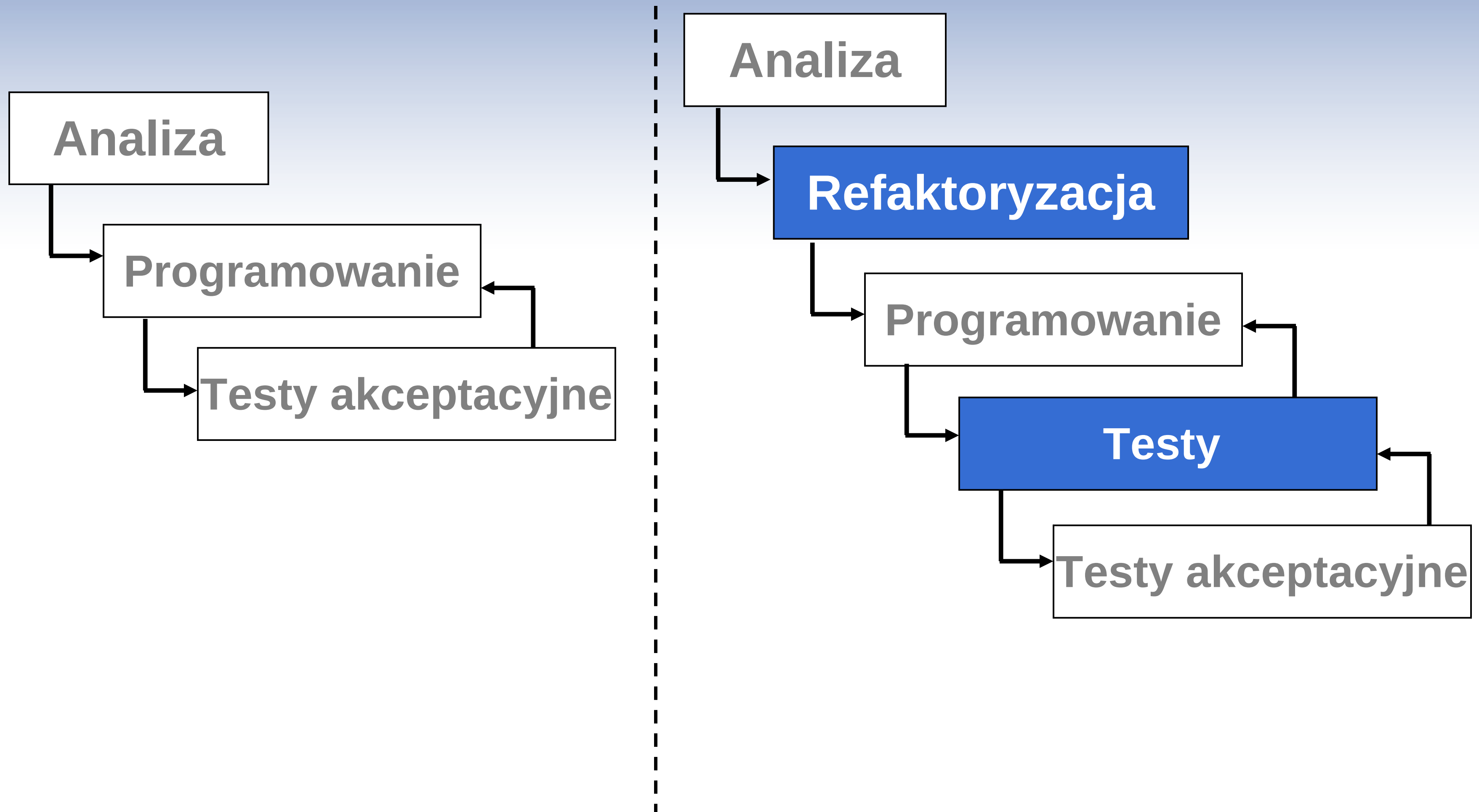
```
int dotProd(int[ ] x, int[ ] y) {
  int res= 0;
  for (int i = 0; i < x.length; i++) {
      res += x[i]*y[i];
   }
}
```

# Koszt refaktoryzacji

- **Refaktoryzacja nie zwiększa funkcjonalności programu, ale kosztuje, gdyż wymaga**
- identyfikacji problemu
- przekształcenia kodu
- weryfikacji poprawności

- **Koszt zależy od:**
  - własności języka programowania
  - wsparcia ze strony narzędzi CASE
  - natury wykonywanego przekształcenia
  - liczby i jakości posiadanych testów

# Porównanie cykli produkcyjnych



# Czy / kiedy refaktoryzacja?

## Za "tak"

- Jeśli były już porażki przy konserwacji systemu
- Przed implementacją nowej funkcjonalności
- Kiedy łatwo jest zidentyfikować obszar zmienności
- Przy regularnych przeglądach

## Za "raczej nie"

- ✗ Stały zakres prac
- ✗ Projekty jednorazowe
- ✗ Przedwcześnie zatwierdzone interfejsy
- ✗ Niestabilny kod
- ✗ Przy bliskich terminach

# Plan wykładu

Przykre zapachy w kodzie programów

# Duplicated Code

- **Nazwa**

  Zduplikowany kod
- **Objawy**

  Identyczny lub podobny kod znajduje się w wielu miejscach systemu
- **Rozwiązanie**

- w jednej klasie: wyłącz wspólne fragmenty do nowej metody (*Extract Method*)
- w klasach o wspólnej nadklasie: wyłącz wspólne części (*Extract Method*), a następnie przesuń ją do nadklasy (*Pull-up the Method*)
- w klasach niezwiązanych: wyłącz wspólne części do nowej klasy (*Extract Class*) i deleguj do nich wywołania

# Long Method

- **Nazwa**

  Długa metoda
- **Objawy**

  Metoda wykonuje w rzeczywistości wiele czynności

  Brak wsparcia ze strony metod niższego poziomu
- **Rozwiązanie**

• wyłącz fragmenty do nowych metod (*Extract Method*)

• wyłącz zmienne tymczasowe do zewnętrznych metod (*Replace Temp with Query)*

• wyłącz metodę do nowej klasy *(Replace Method with Method Object)*

• zmniejsz liczbę parametrów (*Introduce Parameter Object* lub *Preserve Whole Object)*

# Large Class

- **Nazwa**
  Nadmiernie rozbudowana klasa
- **Objawy**
  Klasa posiada zbyt wiele odpowiedzialności
  Liczne klasy wewnętrzne, pola i metody
  Duża liczba metod upraszczających
- **Rozwiązanie**
- wydziel nową klasę i odwołuj się do niej za pomocą referencji (*Extract Class*), dziedziczenia (*Extract Subclass* lub *Extract Superclass*) lub korzystając z polimorfizmu (*Extract Interface*)
- przesuń do niej pola i metody (*Pull up Member*, *Push Down Member*, *Move Member*)

# Switch Statements

- **Nazwa**
  [Skomplikowane] instrukcje warunkowe
- **Objawy**
  Metoda zawiera złożoną, wielopoziomową instrukcję *if* lub *switch*
- **Rozwiązanie**
- wyłącz gałęzie instrukcji warunkowej do polimorficznych klas (*Replace Conditional with Polimorphism/State*)
- wyłącz gałęzie instrukcji warunkowej do podklas (*Replace Conditional with Subclasses*)
- wyłącz wspólne fragmenty do nowej metody (*Extract Method*)

# Message Chains

- **Nazwa**
  Łańcuchy wywołań metod
- **Objawy**
  Łańcuch wywołań przez delegację
  Naruszenie prawa Demeter ("Only talk to your immediate friends.")
- **Rozwiązanie**
- usuń niepotrzebne wywołania i ukryj delegację (*Hide Delegate*)
- przesuń zbędne metody do klas w dalszej części łańcucha (*Extract Method* i *Move Method*)
-

# Data Class

- **Nazwa**

  Pojemnik na dane
- **Objawy**

  Klasa jedynie przechowuje dane i nie posiada metod oferujących użyteczne funkcje (*Data Transfer Object*)
- **Rozwiązanie**

• przesuń część kodu klientów do klasy-pojemnika danych (*Extract Method* i *Move Method*)

• podziel dane klasy pomiędzy klientów i usuń ją (*Inline Class*)

# Refused Bequest

- **Nazwa**
  Odrzucony spadek
- **Objawy**
  Podklasa nie wykorzystuje odziedziczonych metod i pól
- **Rozwiązanie**
- utwórz nową podklasę i przesuń do niej niechciane składowe z nadklasy (*Push Down Member*)
- jeżeli podklasa nie powinna posiadać interfejsu nadklasy, zastosuj delegację zamiast dziedziczenia (*Replace Inheritance with Delegation*)

# Inappropriate Intimacy

- **Nazwa**
  Niewłaściwa hermetyzacja
- **Objawy**
  Klasa bezpośrednio odwołuje się do składowych wewnętrznych innej klasy
- **Rozwiązanie**
- przesuń składową do właściwej klasy (*Move Member*)
- ogranicz powiązania do jednokierunkowych (*Change Bi-directional References to Unidirectional*)
- wyłącz nową klasę ze wspólnych elementów obu klas (*Extract Class*)
- w przypadku dziedziczenia odseparuj nadklasę od podklasy (*Replace Inheritance with Delegation*)

# Lazy Class

- **Nazwa**
  Bezużyteczna klasa
- **Objawy**
  Klasa nie posiada żadnej lub ograniczoną funkcjonalność
- **Rozwiązanie**
- przesuń wybrane funkcje z klas klienckich (*Move Member*)
- w przypadku podklas usuń klasę z hierarchii dziedziczenia (*Collapse Hierarchy*)
- podziel składowe pomiędzy jej klientów (*Move Member*) i usuń ją (*Inline Class*)

# Feature Envy

- **Nazwa**

  Zazdrość o funkcje
- **Objawy**

  Metoda w klasie częściej korzysta z metod
  w obcych klasach niż we własnej

  Niska spójność klasy
- **Rozwiązanie**

• przesuń metodę do właściwej klasy (*Move Method*)

• przekaż referencję *this* do metody w drugiej klasie
 (wzorzec projektowy *Visitor*)

# Shotgun Surgery

- **Nazwa**

  Rozproszona modyfikacja
- **Objawy**

  Zmiana w jednym miejscu powoduje konieczność modyfikacji w innych
- **Rozwiązanie**
- umieść zmienne elementy wewnątrz jednej klasy (*Extract Class*, *Move Method* i *Move Field*)
- usuń zbędne klasy (*Inline Class*)

# Plan wykładu

Wybrane refaktoryzacje

# Encapsulate Collection

**Problem**

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

**Cel**

Przeniesienie odpowiedzialności za kolekcję do jej właściciela

- **Sposób**

- 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;
    }
}
```

# 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);
    }
}
```

# Introduce Null Object

**Problem**

Referencje do klasy wymagają ciągłego porównywania z wartością *null*

**Cel**

Wprowadzenie obiektu reprezentującego wartość *null*

- **Sposób**
- 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()*

# Przykład

```
Ksiazka ksiazka = null
String autor = null;

// operacje na obiekcie ksiazka
if (ksiazka != null) {
   autor = ksiazka.autor();
} else {
   autor = "";
}

public class Ksiazka {
   public String autor() {
      return autor;
   }
}
```

# Przykład

```
public class PustaKsiazka extends Ksiazka {
   public czyPusta() {
      return true;
   }
}
Ksiazka ksiazka = new PustaKsiazka();
String autor;

if (ksiazka.czyPusta()) {
    autor = ksiazka.autor();
} else {
    autor = "";
}
```

# Przykład

```
public class PustaKsiazka extends Ksiazka {
    public String autor() {
       return "";
    }
}

Ksiazka ksiazka = new PustaKsiazka();
String autor = ksiazka.autor();
```

# Change Unidirectional Association with Bi

**Problem**

Asocjacja pomiędzy dwoma porównywalnymi klasami jest jednostronna

**Cel**

Symetryzacja relacji

- **Sposób**
- utwórz w klasie kontrolowanej referencję do klasy kontrolującej
- dodaj metodę uaktualniającą tę referencję
- popraw metodę w klasie kontrolującej, aby wywoływała metodę klasy kontrolowanej

# Przykład

```
public class Tom {
    private Ksiazka ksiazka;

    Ksiazka ksiazka() {
        return Ksiazka;
    }

    void przypiszKsiazke(Ksiazka ksiazka) {
        this.ksiazka = ksiazka;
    }
}

public class Ksiazka {
   //...
}
```

# Przykład

```
public class Tom {
    private Ksiazka ksiazka;

    Ksiazka ksiazka() {
        return Ksiazka;
    }

    void przypiszKsiazke(Ksiazka ksiazka) {
        this.ksiazka = ksiazka;
    }
}

public class Ksiazka {
    private Set tomy = new HashSet();

    public Set __tomy() {
       return tomy;
    }
}
```

# Przykład

```
public class Tom {
    private Ksiazka ksiazka;

    Ksiazka ksiazka() {
        return Ksiazka;
    }
    void przypiszKsiazke(Ksiazka ksiazka) {
        if (ksiazka != null)
            ksiazka.__tomy().remove(this);
        this.ksiazka = ksiazka;
        if (ksiazka != null)
            ksiazka.__tomy().add(this);
    }
}
public class Ksiazka {
    private Set tomy = new HashSet();

    public Set __tomy() {
       return tomy;
    }
}
```

# Replace Inheritance with Delegation

**Problem**

Podklasa niepotrzebnie dziedziczy pola i metody z nadklasy

**Cel**

Zastąpienie dziedziczenia jawną delegacją do dawnej nadklasy

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

# Przykład

```
public class KartaCzytelnicza {
   private Czytelnik czytelnik;

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

public class KartaCzytelniczaUlgowa extends KartaCzytelnicza{

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

# Przykład

```
public class KartaCzytelnicza {
   private Czytelnik czytalnik;

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

public class KartaCzytelniczaUlgowa extends KartaCzytelnicza{
   private KartaCzytelnicza karta = this;

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

# Przykład

```
public class KartaCzytelnicza {
    //...
}

public class KartaCzytelniczaUlgowa {
   private KartaCzytelnicza karta;
   public KartaCzytelniczaUlgowa(KartaCzytelnicza karta) {
      this.karta = karta;
   }

   public double naliczKare(int dni) {
      return 0.4 * karta.naliczKare(dni)
   }
   public Czytelnik czytelnik() {
      return karta.czytelnik();
   }
}
```

# Replace Conditional with Polymorphism

**Problem**

Wyrażenie warunkowe wykonuje różne akcje w zależności od stanu obiektu

**Cel**

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

- **Sposób**
- 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

# Przykład

```
public class KartaCzytelnicza {
   private TypKarty typKarty;

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

# 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;
      }
   }
}
```

# Przykład

```
abstract public class TypKarty {
   abstract public int typKarty();
   abstract double oplata(KartaBiblioteczna karta);
}

public class TypKartyJunior extends TypKarty {
   public double oplata(KartaBiblioteczna karta) {
       return 0;
   }
}

public class TypKartyStandard extends TypKarty {
   public double oplata(KartaBiblioteczna karta) {
       return 100 – (10 * karta.okresCzytelnictwa);
   }
}

public class TypKartySenior extends TypKarty {
   public double oplata(KartaBiblioteczna karta) {
       return 50;
   }
}
```

# Podsumowanie

- Refaktoryzacja jest techniką umożliwiającą łatwiejszą i tańszą pielęgnację oprogramowania
- Refaktoryzacja zachowuje funkcjonalność programu
- Analiza statyczna i testowanie są metodami weryfikacji poprawności refaktoryzacji
- Przykry zapach w kodzie jest określeniem złej struktury programu wymagającej poprawy
- Identyfikacja przykrych zapachów wymaga złożonego mechanizmu wspomagania decyzji

# Ćwiczenia

- Kod źródłowy: http://www.iem.pw.edu.pl/~jstar/dyd/java/ref/
- Opracować test dla metody Klient.faktura() (na podstawie klasy Main)
- Wykonać refaktoryzację *podział metody* (faktura)
- Wykonać refaktoryzację *przeniesienie metody* (faktura do klasy Zakup)
- Wykonać refaktoryzację *zastąpienie wyrażenia warunkowego przez polimorfizm*