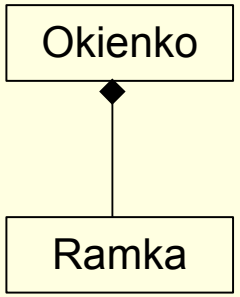
Java niezbędnik programisty
spotkanie nr 4

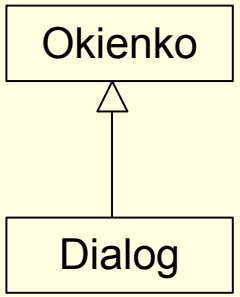
Dziedziczenie, wzorce projektowe

Wielokrotne użycie kodu

- agregacja całkowita (ang. *composition*) – związek całość część
  - czas życia część ograniczony przez czas życia całości
  - część należy tylko do jednej całości
  - klasa tworzy i używa obiekty innych klas



- uogólnienie (ang. *generalization*) – związek między nadklasą i podklasą
  - podklasa jest szczególnym przypadkiem nadklasy
  - podklasy można używać wszędzie tam gdzie nadklasy



Przykład

```
class A {
    void staraMetoda() {
         System.out.println("A.staraMetoda()");
    }
    void pisz() {
         System.out.println("A.pisz()");
    }
}

class B extends A {
    void pisz() {
         System.out.println("B.pisz()");
    }
    void nowaMetoda() {
         System.out.println("B.nowaMetoda()");
         pisz();
         super.pisz();
    }
}
```

```
B b = new B();
b.nowaMetoda();
```

B.nowaMetoda()

B.pisz()

A.pisz()

```
b.pisz();
```

B.pisz()

```
b.staraMetoda();
```

A.staraMetoda()

```
A a = new B();
a.pisz();
```

B.pisz()

```
b.staraMetoda();
```

A.staraMetoda()

Przykład

```
class A {
   void pisz() {
      System.out.println("A.pisz()");
   }

   void jakieWiązanie() {
      pisz();
   }
}

class B extends A {
   void pisz() {
      System.out.println("B.pisz()");
   }
}
```

```
B b = new B();
b.jakieWiązanie();
```

B.pisz()

```
A a = new B();
a.jakieWiązanie();
```

B.pisz()

Przykład

```
class A {
   int x = 1;

   void testZmiennej() {
      System.out.println(x);
   }
}

class B extends A {
   int x = 2;
   int y = 2;

   void testZmiennej() {
      System.out.println(x);
      System.out.println(super.x);
      super.testZmiennej();
   }
}
```

```
B b = new B();
A a = new B();
b.testZmiennej();
    2
    1
    1

a.testZmiennej();
    2
    1
    1
```

Przykład

```
class A {
   int x = 1;
}

class B extends A {
   int x = 2;
   int y = 2;
}

class C extends B {
   char x = 'a';

   void testZmiennej() {
      System.out.println(x);
      System.out.println(super.x);
      //System.out.println(super.super.x);
   }
}
```

Kolejność inicjalizacji

- statyczne składowe nadklasy
- statyczne składowe podklasy

(pełna inicjalizacja nadklasy)

- zwykłe składowe nadklasy
- konstruktor nadklasy (określony w pierwszej instrukcji konstruktora podklasy)
- uwaga nad dynamiczne wiązanie

(pełna inicjalizacja podklasy)

- zwykłe składowe podklasy
- konstruktor podklasy (trzeba zacząć od wywołania konstruktora nadklasy, domyślne konstruktory może wywołać kompilator)

Przykład

```
class Echo {
    static int getValue(String s) {
        System.out.println("getValue("+s+")");
        return 1;
    }
}
```

Przykład

```
class A {
   int x = Echo.getValue("A.x");
   void pisz() {
      System.out.println("A.pisz()");
   }

   A() {
      System.out.println("A()");
      pisz();
   }
}

class B extends A {
   int x = Echo.getValue("B.x");
   int y = Echo.getValue("B.y");
   void pisz() {
      System.out.println("B.pisz()");
   }

   B() {
      super();
      System.out.println("B()");
      pisz();
   }
}
```

```
B b = new B();
```

getValue(A.x)

A()

B.pisz()

getValue(B.x)

getValue(B.y)

B()

B.pisz()

Zastosowanie polimorfizmu

- Kontener używa serwletów do obsługi żądań (agregacja).
- Programista rozszerza `javax.servlet.http.HttpServlet`.
- Kontener używa rozszerzeń użytkownika nie wiedząc nic o dodatkowej funkcjonalności. To jest rzutowanie w górę – upcasting (por. strzałki na diagramach).
- Nową funkcjonalność wprowadzamy przesłaniając metody i polegając na dynamicznym wiązaniu.
- Jak nie trzeba rzutować w górę warto rozważyć agregację.

Serwlet API

javax.servlet.GenericServlet

```
  public void service(ServletRequest, ServletResponse);
```

javax.servlet.http.HttpServlet

```
 public void service(ServletRequest rq, ServletResponse rs) {
   service((HttpServletRequest) rq, (HttpServletResponse) rs);
 }
 protected void service(HttpServletRequest rq,
                        HttpServletResponse rs) {
   //sprawdź jaki rodzaj żądania
   //wywołaj doGet/doPost/doHead/doDelete/...
 }
 public void doXXX(HttpServletRequest rq,
                   HttpServletResponse rs) {}
```

Serwlet API

nasz.pakiet.PierwszySerwlet

```
public void doXXX(HttpServletRequest rq,
                  HttpServletResponse rs) {
  //obsługa żądania
}
```

- inny przykład: parsowanie dokumentów XML przy pomocy SAX
- dzięki dziedziczeniu i agregacji możliwe jest **tworzenie iteracyjne**

Dla programistów C++

```
class Nad {
   void pisz(int x) {
      System.out.println("Nad.pisz(int)");
   }
}

class Pod extends Nad{
   void pisz(char x) {
      System.out.println("Pod.pisz(char)");
   }
}

Pod p = new Pod();
p.pisz('a');
p.pisz(1);
//działa i nadpisywanie i podpisywanie
```

Finall

- dane
  - stałe kompilacji nie będą wplatane w wyrażenia
  - wartość typu podstawowego lub referencji nie może ulec zmianie
  - można użyć z argumentami metod
- metody
  - nie mogą być przesłaniane
  - nie ma dynamicznego wiązania, czasami kompilator może wstawiać kod metody w miejsce jej wywołania (*ang. inline*)
- obiekty
  - nie można rozszerzać

Atrybuty final trzeba zainicjalizować

```
class TestFinal {
   final int x;
   void pisz(final String s) {
      //s = "a";
      System.out.println("pisz("+s+")");
   }

   TestFinal() {
     x = 1; //inicjalizacja musi nastąpić
            //najpóźniej w konstruktorze
   }
}
```

Final a private

- metod private i tak nie można przesłonić, ale można zdefiniować metodę o takiej samej sygnaturze

```
class Zagadka {
   //final tu nic nie zmieni
   private void pisz() {
      System.out.println("Zagadka.pisz()");
   }

   Zagadka() {pisz();}
}

class PseudoPrzesloniecie extends Zagadka {
   private void pisz() {
      System.out.println("PseudoPrzesloniecie.pisz()");
   }

   PseudoPrzesloniecie() {pisz();}
}

PseudoPrzesloniecie p = new PseudoPrzesloniecie();
```

PseudoPrzesłonięcie

```
class Zagadka {
   private void pisz() {
      System.out.println("Zagadka.pisz()");
   }

   Zagadka() {pisz();}
}

class PseudoPrzesloniecie extends Zagadka {
   private void pisz() {
      System.out.println("PseudoPrzesloniecie.pisz()");
   }

   PseudoPrzesloniecie() {pisz();}
}

PseudoPrzesloniecie p = new PseudoPrzesloniecie();
```

Zagadka.pisz()
PseudoPrzesloniecie.pisz()

Hermetyzacja – get/set/is

```
class Pracownik {
   private double pensja;
   public void setPensja(double nowa) {
      pensja = nowa;
   }
   public void getPensja() {
      return pensja;
   }
}
```

Hermetyzacja – get/set/is

```
class Pracownik {
   private double pensja;
   private double[] historiaPodwyżek;
   private void dodajPodwyżkę(double oIle) {
      ...
   }

   public void setPensja(double nowa) {
      dodajPodwyżkę(pensja-nowa);
      pensja = nowa;
   }

   public void getPensja() {
      return pensja;
   }
}
```

Wielodziedziczenie

- Można rozszerzać tylko jedną nadklasę.
- Wielodziedziczenie w C++ sprawiało problemy.
- Java pozwala symulować wielodziedziczenie przy pomocy interfejsów.

Sprzężenie

Sprzężenie jest miarą jak bardzo jeden element jest połączony z, wie o lub polega na innych elementach. Element o luźnym (lub słabym) sprzężeniu nie jest zależny od zbyt wielu innych elementów; "zbyt wiele" zależy od sytuacji. Do tych elementów zalicza się klasy, podsystemy, systemy itd.

Klasa z silnym (lub mocnym) sprzężeniem polega na zbyt wielu innych klasach. Takie klasy mogą być niepożądane; niektóre borykają się z następującymi problemami:

- Zmiany w zależnych klasach pociągają zmiany lokalne.
- Trudno je zrozumieć w odosobnieniu.
- Trudniej je ponownie użyć, ponieważ wymaga to dodatkowej obecności klas, od których zależą.

Luźne sprzężenie

- **Problem** Jak utrzymać niski poziom zależności i mały wpływ zmian oraz zwiększać ponowne użycie?
- **Rozwiązanie** Odpowiedzialność przyporządkuj tak, aby pozostał niski poziom (niepotrzebnego) sprzężenia.

Ten wzorzec należy stosować jako **kryterium oceny**. Nie powinien być celem samym w sobie. Czasami inne wzorce mogą sugerować rozwiązania obniżające spójność.

Silne sprzężenie zarówno ze stabilnymi, jak i wszechobecnymi elementami rzadko stanowi problem. Aplikacja J2EE może być na przykład bezpiecznie sprzężona z bibliotekami Javy i J2EE (java.util, itd.), ponieważ są stabilne i powszechne.

Spójność

Spójność (lub bardziej ściśle spójność funkcjonalna) jest miarą jak bardzo powiązane i ukierunkowane są odpowiedzialności wyznaczone elementowi.

Klasa o niskiej spójności wykonuje wiele niezwiązanych czynności lub wykonuje zbyt dużą pracę. Takie klasy są niepożądane. Borykają się z następującymi problemami:

- są trudne do zrozumienia
- są trudne w ponownym użyciu
- są trudne w konserwacji
- są delikatne; nieustannie dotykane przez zmiany

Wysoka spójność

- **Problem** Jak zachować kontrolę nad złożonością?
- **Rozwiązanie** Odpowiedzialność przydziel tak, by spójność pozostała wysoka.

Wzorzec Wysoka spójność – jak wiele innych rzeczy w technice obiektowej – ma odpowiednik w świecie rzeczywistym. Często zdarza się, że gdy osoba podejmie się zbyt wielu, niezależnych odpowiedzialności – szczególnie takich, które właściwie powinna zlecić innym - wtedy nie jest wydajna.

Wzorce projektowe

- Pewne wypróbowane i poprawne rozwiązania problemów projektowych mogą być (i były) wyrażane jako dobre praktyki, heurystyki lub **wzorce** – nazwane przepisy zadanie-rozwiązanie. (na podstawie "*Applying UML And Patterns"* Craiga Larmana).
- Sam termin „wzorzec” ma sugerować coś powtarzalnego. To nie są nowe pomysły!
- Dwa istotne zbiory wzorców to wzorce GOF i GRASP.
- GRASP to metodyczne podejście do uczenia się podstaw projektowania obiektowego.
- GRASP: Rzeczoznawca, Twórca, **Wysoka spójność**, **Luźne sprzężenie**, Zarządca, Polimorfizm, Pośrednictwo, Czysty wymysł, Kapsułkowanie zmienności
- GOF (23 ogólnie uznane wzorce): Fabryka, Singleton, Adapter, Strategia, Kompozyt, Fasada, Obserwator (Delegowanie, Wydawca-Prenumerator)

Adapter

- ***Kontekst/Problem****:* Jak poradzić sobie z niekompatybilnymi interfejsami lub dostarczyć stabilny interfejs dla podobnych komponentów posiadających różniące się interfejsy?
- ***Rozwiązanie****:* Używając pośredniego obiektu adaptera przekształć oryginalny interfejs komponentu na inny.

Dodajemy **poziom pośredni** składający się z obiektów, które przystosowują różnorodne zewnętrzne interfejsy do spójnego interfejsu używanego w aplikacji!

Konwencja nazewnicza: nazwę wzorca wbudowujemy w nazwę typu

Adaptery są Fasadami (opakowują dostęp do podsystemu lub systemu w pojedynczym obiekcie).

Fabryka

- Kto powinien tworzyć obiekty, np. obiekty reprezentujące (adaptery) zewnętrzne usługi (mogące mieć różne interfejsy, np. AdapterKartyVisa).
- Chcemy utrzymać **rozdzielenie zagadnień** – obiekty dziedzinowe odpowiadają jedynie za logikę aplikacji (chcemy zachować **wysoką spójność**). (Nie zawsze jest to możliwe – **programowanie aspektowe**).

```
public NaszaFabryka {
    ...
    public AdapterKarty getAdapterKarty() {
         //odczytaj z pliku konfiguracyjnego jakiego adaptera użyć

         //utwórz odpowiedni i zwróć go
    }
}
```

Singleton

- Jak uzyskać dostęp do fabryki z poprzedniego przykładu?
- Kto ją tworzy?
- Zazwyczaj potrzeba tylko jednej fabryki?
- Możemy przekazywać fabrykę jako parametr, ale możemy użyć wzroca Singleton

```
public static synchronized NaszaFabryka getEgzemplarz()
{
   if ( egzemplarz == null )
   {
      // sekcja krytyczna, jeżeli aplikacja jest wielowątkowa
      egzemplarz = new FabrykaUsług();
   }
   return egzemplarz;
}
//notacja getXxx(), setXxx(), isXxx()
```

Dostęp globalny w Javie

```
public class Kasa
{
   public void inicjuj()
   {
      ... jakąś pracę ...
      //uzyskujemy dostęp do będącej singletonem Fabryki przez
      //wywołanie getEgzemplarz
      adapterKarty =
               NaszaFabryka.getEgzemplarz().getAdapterKarty();

      ... jakąś pracę ...
   }
   // pozostałe metody...
}
```