Programowanie obiektowe

Programowanie obiektowe (ang. object-oriented programming) jest obecnie najpopularniejszą techniką tworzenia programów komputerowych. Program komputerowy wyraża się jako zbiór obiektów będących bytami łączącymi stan (czyli dane) i zachowanie (czyli metody, które są procedurami operującymi na danych obiektu). W celu realizacji zadania obiekty wywołują nawzajem swoje metody, zlecając w ten sposób innym obiektom odpowiedzialność za pewne czynności.

Programowanie obiektowe umożliwia programowi dostosowanie się do specyficznego języka danego problemu poprzez dodanie nowych typów obiektów, dzięki czemu przy pisaniu kodu opisującego rozwiązanie natrafiamy na słowa wyrażające sam problem. Abstrakcja taka jest potężniejsza i bardziej elastyczna od tych, którymi dysponowaliśmy wcześniej (np. programowanie proceduralne). Pozwala ono wręcz na opisanie problemu we właściwych mu kategoriach.

W porównaniu z tradycyjnym programowaniem proceduralnym, w którym dane i procedury nie są ze sobą powiązane, programowanie obiektowe ułatwia tworzenie dużych systemów, współpracę wielu programistów i ponowne wykorzystywanie istniejącego kodu.

OBIEKT

Obiekt wg Booch'a to: Obiekt ma stan, zachowanie i tożsamość

Oznacza to, że może:

posiadać dane wewnętrzne (określające jego stan)
metody (stanowiące zachowanie obiektu)
obiekt można jednoznacznie odróżnić od innych obiektów (tożsamość)

Przez obiekt rozumiemy dynamicznie stworzony egzemplarz jakiejś klasy, obiekt ten jest nazwany instancją tej klasy. Przykładowy proces wykonania mnożenia używając programowania obiektowego:

1	int result = 0;	inicjalizacja zmiennej
2	arithemitc_Obj1 = new Arithmetic_Class();	Tworzenie nowej instancji klasy Arithmetic_Class
3	result = arithithemitic_Obj1.multiply(78,69);	78*69

KLASA

Klasa - opisuje zbiór obiektów o tej samej charakterystyce (danych składowych) oraz tych samych możliwych zachowaniach* jak np. liczba zmiennoprzecinkowa, której dane charakterystyczne to wartości i zestaw dozwolonych na niej operacji. Różnica polega na tym, że raczej definiujemy klasy odpowiadające rozwiązywane problemowi.

A jakby ktoś na komisji **kciał** coś bardziej łopatologicznego: **klasa** jest to taki plan wg jakiego zostanie stworzony obiekt. Opisuje on dane jakie będzie posiadał i jak będzie się zachowywał czyli metody; obiekt za to jest to tak jakby pochodna klasy, coś z niej stworzonego i ten dany obiekt będzie posiadał wszystkie właściwości klasy, z której został stworzony. Może być wiele obiektów z tej samej klasy.

*metoda - w językach obiektowych funkcje zdefiniowane w klasach nazywamy metodami.

W związku z klasą chciałbym wnieść zmiankę o konstruktorach, a mianowicie jest to taka metoda, o identycznej nazwie z klasą, w której się ona znajduje. Podczas tworzenia nowego obiektu jest ona automatycznie wywoływana.

Modyfikatory dostępu do składowych klas - deklarując składowe klasy (przez składową rozumiemy tu atrybut, metodę, konstruktor, klasę lub interfejs) można poprzedzać je jednym z modyfikatorów dostępu. Modyfikator określa, gdzie będzie można używać danej składowej klasy. Dostęp do składowej jest oczywiście możliwy tylko wtedy, gdy dostępny jest typ, do którego składowej chcemy się dostać. Ponadto składowa musi mieć odpowiednie modyfikatory dostępu. Oto lista modyfikatorów występujących w Javie:

- public (dostęp publiczny) umożliwia dostęp bez ograniczeń.
- protected (dostęp chroniony) umożliwia dostęp z pakietu, w którym jest zdefiniowana klasa zawierająca składową i z jej podklas, nawet jeśli są zdefiniowane w innym pakiecie.
- private (dostęp prywatny) umożliwia dostęp z klasy zadeklarowanej na najwyższym poziomie struktury programu, w której jest zawarta (nie koniecznie bezpośrednio) deklaracja opatrzona tym modyfikatorem dostępu.
- brak modyfikatora (dostęp domyślny) umożliwia dostęp z pakietu, w którym jest zawarta rozważana deklaracja.

Oczywiście dla jednej deklaracji można podać tylko jeden modyfikator dostępu.

Ogólnie zasady stosowania modyfikatorów dostępu są dość intuicyjne ale zwróćmy uwagę na pewne niuanse:

- dostęp chroniony zezwala na dostęp nie tylko w podklasach, ale też w całym pakiecie,
- dostęp prywatny umożliwia dostęp w całej klasie zadeklarowanej na najwyższym poziomie, zatem w poniższej sytuacji:

```
class Zewnętrzna{
 class Lokalna1{
  private int i1;
 }
 class Lokalna2{
  int met(){return new Lokalna1().i1;}
 }
}
```

metoda z klasy Lokalna2 ma prawo odwołać się do prywatnej składowej obiektu z klasy Lokalna1, co jest mało intuicyjne.
prawa dostępu dotyczą klas, a nie obiektów.

Niezwykle istotnym jest, by definiując klasę starannie przemyśleć, co obiekty klasy mają chronić, a co udostępniać na zewnątrz. Jeśli deklarujemy atrybuty, to praktycznie zawsze powinniśmy zadeklarować je jako prywatne lub chronione. Metody zwykle deklarujemy jako publiczne, ale zdarzają się też metody, które powinny być używane tylko w obrębie danej klasy lub hierarchii, wtedy definiujemy je jako prywatne lub chronione.

Przykłady

Java

```
class Osoba{
	String imię;
	String nazwisko;
}
```

```
Osoba o = new Osoba();
```

```
class Osoba{
	String imię;
	String nazwisko;

	Osoba(String imię, String nazwisko){
		this.imię = imię;
		this.nazwisko = nazwisko;
	}

	String imię(){
		return imię;
	}

	String nazwisko(){
		return nazwisko;
	}
}
```

```
Osoba o2 = new Osoba(„Jan”, „Kowalski”);

Osoba o3 = new Osoba(„John”, „Blacksmith”);
```

```
class Zewnętrzna{

	class Lokalna1{
		private int i1;
	}

	class Lokalna2{
		int met(){
			return new Lokalna1().i1;}
		}
}
```

C++

```
#include <string>
using std::string;
…
Class Osoba{
public:
	string imię;
	string nazwisko;
};
```

```
Osoba o;
```

```
#include <string>
using std::string;
…
Class Osoba{
public:
	string imię;
	string nazwisko;

	Osoba(string imię, string nazwisko){
		this->imię=imię;
		this->nazwisko=nazwisko;
	}

string imie(){
	return imię;
}

string nazwisko(){
	return nazwisko;
}

~Osoba();
};
```

```
Osoba o2(„Jan”, „Kowalski”);

Osoba o3(„John”, „Blacksmith”);
```

```
Class Zewnętrzna{
public:
        class Lokalna1{
                public: Lokalna1();
                private: int i1;
        };

        class Lokalna2{
                public: int met(){
                return Lokalna1().i1;                    // new ?
        };
};
```

Czym jest dziedziczenie i hierarchia?

Bardzo często podczas takiego modelowania zauważamy, że pewne pojęcia są mocno ze sobą związane. Podejście obiektowe dostarcza mechanizmu umożliwiającego bardzo łatwe wyrażanie związków pojęć polegających na tym, że jedno pojęcie jest uszczegółowieniem (lub uogólnieniem) drugiego. Ten mechanizm nazywa się dziedziczeniem. Stosujemy go zawsze tam, gdzie chcemy wskazać, że dwa pojęcia są do siebie podobne.

Zwróćmy uwagę, że zastosowanie dziedziczenia jest związane ze znaczeniem klas powiązanych tym związkiem, a nie z ich implementacją.

Dziedziczenie jest jedną z fundamentalnych cech podejścia obiektowego. Pozwala kojarzyć klasy obiektów w hierarchie klas. Te hierarchie, w zależności od użytego języka programowania, mogą przyjmować postać drzew, lasów drzew, bądź skierowanych grafów acyklicznych.

Realizacja dziedziczenia polega na tym, że klasa dziedzicząca dziedziczy po swojej nadklasie wszystkie jej atrybuty i metody (i nie ma znaczenia, czy te atrybuty i metody były zadeklarowane bezpośrednio w tej nadklasie, czy ona też odziedziczyła je po swojej z kolei nadklasie).

W różnych językach programowania można natknąć się na różną terminologię. Klasę po której inna klasa dziedziczy nazywa się nadklasą lub klasą bazową, zaś klasę dziedziczącą podklasą lub klasą pochodną.

Oto typowe zastosowania dziedziczenia:

- opisanie specjalizacji jakiegoś pojęcia (np. klasa Prostokąt dziedzicząca po klasie Czworokąt),
- specyfikowanie pożądanego interfejsu (klasy abstrakcyjne, interfejsy),
- opisywanie rozszerzania pojęć (np. klasa KoloroweOkno dziedzicząca po klasie CzarnoBiałeOkno).

Często podklasy różnią się pod względem realizacji zobowiązań określonych w nadklasach (na przykład zobowiązanie do śpiewania przyjęte w klasie Ptak będzie inaczej realizowana w podklasie Słowik, a inaczej w podklasie Wrona). Czasami chcemy wyrazić w hierarchii klas wyjątki (np. Pingwin choć jest niezwykle sympatyczny to jak na przedstawiciela klasy Ptak słabo lata). Takie sytuacje możemy wyrazić dzięki przedefiniowywaniu metod (ang. method overriding).

Przykłady

Java

```
class Pracownik extends Osoba
      // dziedziczenie po klasie

 class Samochód implements Pojazd, Towar
      // dziedziczenie po kilku interfesjach

 class Chomik extends Ssak implements Puchate, DoGłaskania
      // dziedziczenie po klasie i kilku interfejsach
```

```
class A{
  int iA=1;
  void infoA(){
    System.out.println(
      "Jestem infoA() z klasy A\n"+
      "  wywołano mnie w obiekcie klasy " + this.getClass().getSimpleName() + "\n" +
      "  iA="+iA);
  }
 }

class B extends A{
  int iB=2;
  void infoB(){
    infoA();
    System.out.println(
      "Jestem infoB() z klasy B\n"+
      "  wywołano mnie w obiekcie klasy " + this.getClass().getSimpleName() + "\n" +
      "  iA="+iA + ", iB=" + iB);
  }
 }

class C extends B{
  int iC=3;
  void infoC(){
    infoA();
    infoB();
    System.out.println(
      "Jestem infoC() z klasy C\n"+
      "  wywołano mnie w obiekcie klasy " + this.getClass().getSimpleName()+ "\n" +
      "  iA="+iA + ", iB=" + iB + ", iC=" + iC);
  }
 }

 C c = new C();
 c.infoC();
```

~ //

Jestem infoA() z klasy A
 wywołano mnie w obiekcie klasy C
 iA=1
Jestem infoA() z klasy A
 wywołano mnie w obiekcie klasy C
 iA=1
Jestem infoB() z klasy B
 wywołano mnie w obiekcie klasy C
 iA=1, iB=2
Jestem infoC() z klasy C
 wywołano mnie w obiekcie klasy C
 iA=1, iB=2, iC=3

C++

Class Pracownik: [public] Osoba

Class samochód: [public] Pojazd, Towar

// C++ pozwala na wielodziedziczenie po kilku klasach, właściwie jest ono równoważne intefejsom, ale bardziej niebezpieczne

---

```
Class A{
public:
        int iA;
        void infoA(){
        };
        int A(){
                int initial_iA};
        ~A();
};

A::A(int initial_iA){
        iA=initial_iA;
}

A::~A(){
}

void A::infoA(){
        cout << " Jestem infoA() z klasy A, wywolano mnie w obiekcie klasy C i" << iA;
}
```

```
Class B: public A{
public:
	int iB;
	void infoB(){
	};
	int B(){
		int initial_iB};
	~B();
};

B::B(int initial_iB){
	iB=initial_iB;
}

B::~B(){
}

void B::infoB(){
	infoA();
	cout << " Jestem infoB() z klasy A, wywolano mnie w obiekcie klasy C i" << iA << iB;
}

Class C: public B{
public:
	int iC;
	void infoC(){};
	int C(){int initial_iC};
	~C();
};

C::C(int initial_iC){
	iC=initial_iC;
}

C::~C(){
}

void C::infoC(){
	infoA();
	infoB();
	cout << " Jestem infoC() z klasy C, wywolano mnie w obiekcie klasy C i" << iA << iB <<
	iC;
}

int main(){
	A a(1);
	B b(2);
	C c(3);
	c.infoC();
}
// Wywołanie tej tego programu w c++ powinno oznaczać to samo co odpowiednika w Javie.
```

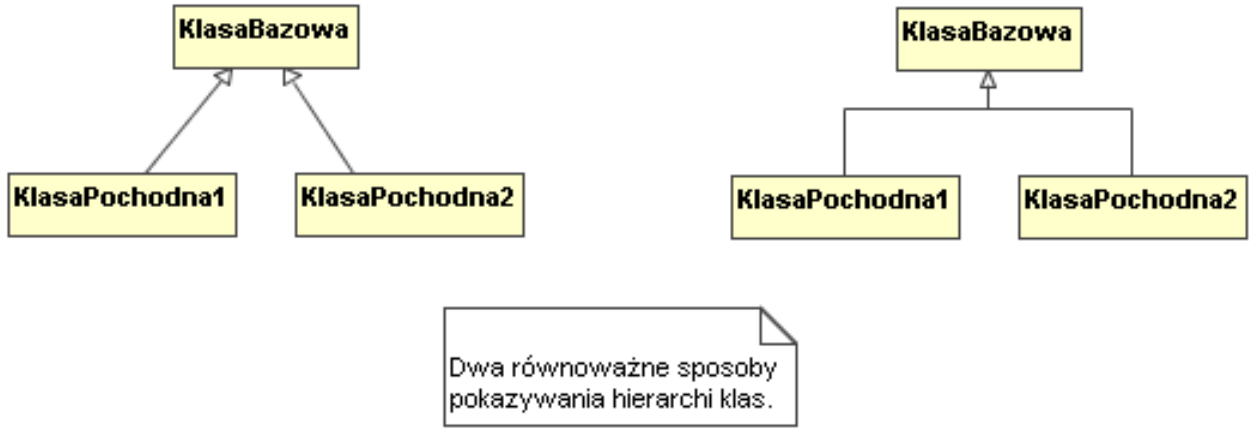
Przykład Hierarchii klas na podstawie gry Monopol:

Tak jakby ktoś dostał pytanie to może ładnie im pościemniać ;)

Implementując grę w Monopol trzeba np. zdefiniować klasy odpowiadające różnym rodzajom pól. Wszystkie rodzaje pól będą mieć nazwę i opis oraz będą posiadały metodę wykonującą czynności związane z odwiedzeniem pola przez któregoś z graczy. Treść tej metody zależy od rodzaju odwiedzanego pola i będzie inna dla pola, które można nabyć, pola szansy czy pola poboru opłaty podatkowej. Poza nazwą, opisem i tą metodą poszczególne rodzaje pól mogą zawierać inne składowe. Większość obiektowych języków pozwala w tym celu zdefiniować hierarchię klas. Klasa bazowa (ang. base class) bądź inaczej nadklasa (ang. superclass) definiuje składowe wspólne dla wszystkich wariantów. Klasy pochodne (ang. derived class) bądź inaczej podklasy (ang. subclass) definiują pozostałe składowe, które występują tylko w poszczególnych wariantach. Związek między nadklasą i podklasami nazywany jest uogólnieniem (ang. generalization), bądź związkiem uogólnienie-uszczegółowienie. Na diagramach uogólnienie pokazywane jest przy pomocy ciągłej linii łączącej klasę bazową i klasę pochodną, która od strony klasy bazowej zakończona jest niewypełnioną trójkątną strzałką.

Podstawowym kryterium pozwalającym określić kiedy stosować uogólnienie jest odpowiedź na pytanie:
Czy podklasa jest szczególnym przypadkiem nadklasy?

Jeżeli tak, to znaczy, że obiekty podklasy są również egzemplarzami nadklasy i powinny dawać się używać w tych samych kontekstach.
Dziedziczenie



Aby nie powielać kodu, klasa pochodna dziedziczy (ang. inherits) wszystkie składowe klasy bazowej. Jeżeli zachowanie niektórych metod powinno być zmienione, klasa pochodna może je przedefiniować (ang. override) podając nowe definicje. Ponieważ klasa pochodna może również zawierać nowe składowe, często mówi się również, że podklasa rozszerza (ang. extend) nadklasę.

Za wykorzystaniem:

B. Eckel Thinking in Java
mimuw.edu.pl
en.wikipedia.org
pl.wikipedia.org

ps. mam nadzieję, że się nie obrażą ;)