Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

I. Programowanie obiektowe - wprowadzenie

1. Geneza programowania zorientowanego obiektowo

1.1. W obrębie (klasycznego) stylu programowania imperatywnego, wypracowano koncepcję tzw. **programowania strukturalnego**. Zakłada ona grupowanie fragmentów kodu w podprogramy (procedury i funkcje) i definiowanie zasad komunikowania się tych podprogramów między sobą.

1.2. Istotne zalety programowania strukturalnego to:
- poprawa czytelności programu,
- możliwość tworzenia i użytkowania (uniwersalnych) bibliotek podprogramów,
- możliwość równoległego opracowywania kodu programu przez grupę programistów.
- poprawa jakości programów (??).

1.3. W istocie, programowanie strukturalne zakładało strukturalizację samych instrukcji – nie wymuszało strukturalizacji danych. Ujemnymi tego konsekwencjami były:
- dostęp wszystkich podprogramów do "globalnego banku danych" (przypadkowa, niepożądana modyfikacja danej rzutowała na dalszy, nieprawidłowy przebieg obliczeń programu),
- konieczność (zwykle istotnej) modyfikacji kodu źródłowego programu przy dostosowywaniu opracowanego oprogramowania do nowych, zmieniających się warunków zewnętrznych,
- trudności w czytelnym odzwierciedlaniu rzeczywistości na etapie tworzenia kodu programu (rzeczywistość postrzeganą jako zbiór obiektów i zachodzących między nimi relacji, chętnie odwzorowywano w tej naturalnej formie w poszczególnych fazach procesu projektowania).

1.4. Naturalny sposób modelowania rzeczywistości przeniknął w końcu także i do programowania i stał się zalążkiem nowego, współcześnie rozpowszechnionego **stylu programowania zorientowanego obiektowo**. Na jego gruncie wyrosła szeroka klasa

języków obiektowych: tak specjalizowanych, np. O++ (oparty na C++ język do programowania baz danych), jak i ogólnego przeznaczenia, np. C++, C#, czy Java.

1.5. Modelowanie obiektowe w ścisłym tego słowa znaczeniu rozwijało się równolegle z językami programowania obiektowego. W latach 90. ubiegłego wieku, Booch, Jacobson i Rumbaugh postanowili, na bazie znanych metod modelowania (np. metoda Boocha, Object Modelling Technique), opracować zunifikowany język modelowania. W 1996 roku, gdy powstała dokumentacja tzw. Unified Method, kilka wielkich firm komputerowych utworzyło konsorcjum **Unified Modelling Language (UML)**, które po roku działań wypracowało wersję 1.0 języka UML. Rozwojem tego oprogramowania zajmuje się od tamtej pory Object Management Group (wersja UML 2.4.1 w kwietniu 2012).

1.6. Modelowanie systemu informatycznego w UML polega na tworzeniu odpowiednich diagramów. Można je podzielić na dwie kategorie: diagramy struktury (np. klas, obiektów, pakietów, komponentów) oraz diagramy zachowań (np. przypadków użycia, przepływu informacji, aktywności, stanów).

Na diagramy UML można nakładać rozmaite ograniczenia. Do ich definiowania służy język formalny **Object Constraint Language (OCL)**.

Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

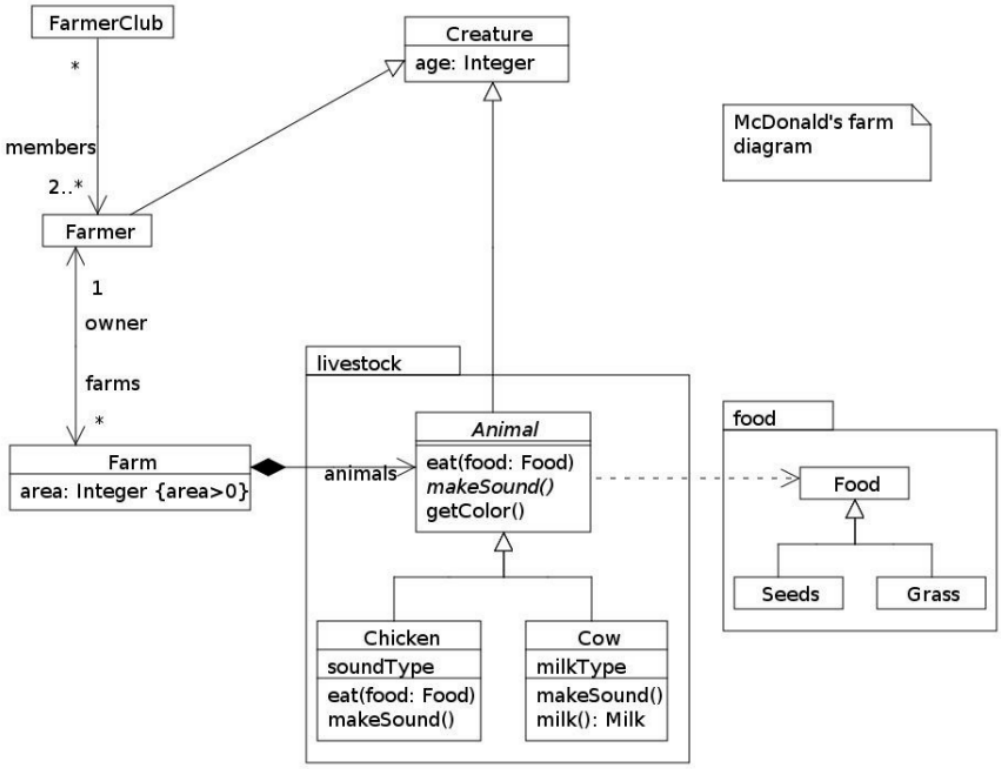

Diagram klas, McDonald's farm, źródło:
http://www.mini.pw.edu.pl/~mkobos/wp-content/uploads/2011/03/uml-diagramy_klas.pdf
(zależność, asocjacja, agregacja całkowita, dziedziczenie)

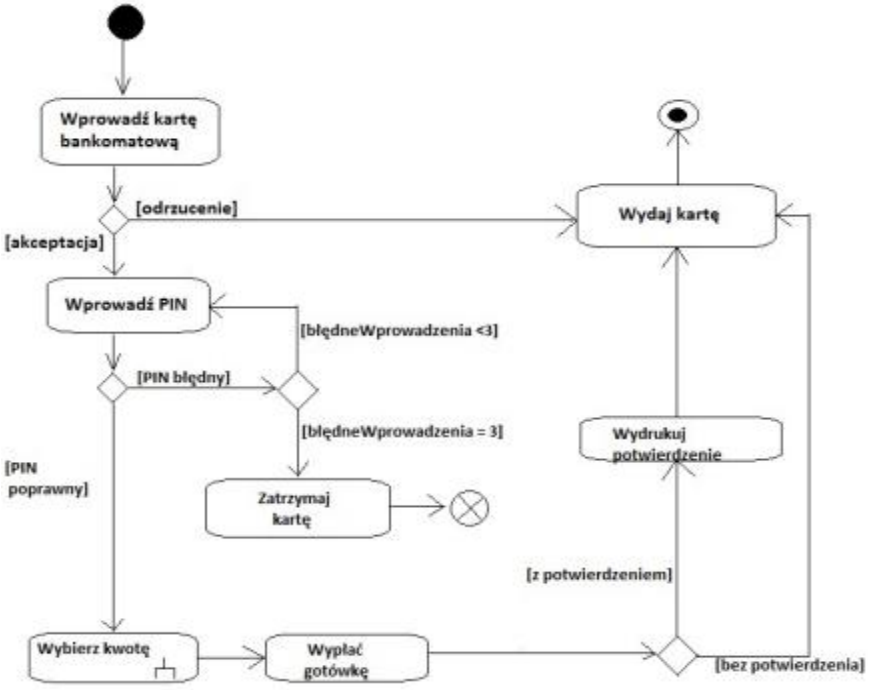

Diagram czynności: funkcjonowanie bankomatu, źródło:
http://math.uni.lodz.pl/~robpleb/wyklad4_6.pdf
(stan początkowy, czynność, rozgałęzienie, przepływ sterowania)

Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

2. Paradygmaty programowania obiektowego

2.1. Podstawowe paradygmaty programowania obiektowego to: hermetyzacja, dziedziczenie i polimorfizm.

Paradygmat rozumie się jako:
- zbiór poglądów teoretycznych i metodologicznych, umożliwiających uzgodnienie działań w pewnej dyscyplinie;
- zbiór pojęć i reguł tworzących podwaliny dla rozwoju dyscypliny (nauki).

2.2. Hermetyzacja (ang. *encapsulation*) oznacza połączenie danych i instrukcji programu w jednostkę programową, zwaną **obiektem**.

2.2.1. Podstawową zaletą hermetyzacji jest możliwość zabezpieczenia danych przed nieograniczonym dostępem ze strony różnych fragmentów kodu programowego. W tym celu wszystkie dane (**pola** w obiekcie) i zapisy instrukcji (**metody** w obiekcie) dzieli się na ogólnodostępne (interfejs obiektowy) i wewnętrzne (implementacja obiektu). Dostęp do pól i metod wewnętrznych jest możliwy tylko za pośrednictwem "łącza obiektowego" – pól i metod ogólnodostępnych. Wybrane pola i metody można więc ukryć przed określonymi (także wszystkimi) obiektami zewnętrznymi.

2.2.2. Hermetyzacja ma ogromne znaczenie dla przenośności programów i optymalizowania nakładów potrzebnych na ich modyfikacje. Wpływa także dodatnio na osiąganie niezawodności w projektach programistycznych.

2.3. Oto przykład definicji pewnego obiektu (klasy!) w języku C++ [*Jakubczyk K., Turbo Pascal i Borland C++*]:

Plik nagłówkowy `Figury.h`:

```
#ifndef H_FIGURY
#define H_FIGURY

class Figura
{ private:
    char nazwa[10];
  protected:
    int x, y, czy_rysunek;
  public:
    Figura(int xPoczatek, int yPoczatek);
    virtual ~Figura();
    virtual void pokaz();
    virtual void ukryj();
    virtual void rysuj();
    int podaj_x()    {return x;}  ;
    int podaj_y()    {return y;}  ;
    int widoczny()   {return czy_rysunek;};
    virtual void przesun_do(int xKoniec,int yKoniec);
    virtual void przesun_o(int xWektor,int yWektor);
};

#endif
```

Plik z kodem źródłowym `Figury.cpp`:

```
#include <graphics.h>
#include "Figury.h"

Figura::Figura(int xPoczatek, int yPoczatek)
{ x = xPoczatek;
  y = yPoczatek;
  czy_rysunek = 0;
}
Figura::~Figura()
{ if (czy_rysunek) ukryj();
}
```

```
void Figura::pokaz()
{ czy_rysunek = 1;
  rysuj();
}

void Figura::ukryj()
{ int color = getcolor();
  setcolor(getbkcolor());
  czy_rysunek = 0;
  rysuj();
  setcolor(color);
}

void Figura::rysuj()
{ putpixel(x, y, getcolor());
}

void Figura::przesun_do(int xKoniec, int yKoniec)
{ int byl_widoczny = czy_rysunek;
  if (byl_widoczny) ukryj();
  x = xKoniec;
  y = yKoniec;
  if (byl_widoczny) pokaz();
}

void Figura::przesun_o(int xWektor, int yWektor)
{ przesun_do(x+xWektor, y+yWektor);
}

int main(void)
{ Figura p1 = Figura(3, 4);
  Figura *p2 = new Figura(25, 30);
  p1.pokaz();
  p1.przesun_o(-2, -3);
  p2->pokaz();
  return 0;
}
```

2.4. Mechanizm **dziedziczenia** (ang. *inheritance*) służy w językach obiektowych do odwzorowania powiązań typu generalizacja – specjalizacja, które często występują w naturze. Umożliwia programiście definiowanie potomków istniejących obiektów. Każdy potomek dziedziczy przy tym pola i metody obiektu bazowego, a dodatkowo uzyskuje pewne pola i metody unikatowe, nadające mu nowy charakter. Typ takiego obiektu potomnego może stać się z kolei typem bazowym do zdefiniowania kolejnego typu potomnego.

2.4.1. Dziedziczenie może mieć charakter jednokrotny lub wielokrotny (wielodziedziczenie). W tym drugim wypadku dopuszcza się możliwość definiowania typów potomnych dziedziczących równocześnie po wielu typach bazowych.

2.4.2. Mechanizm dziedziczenia obiektów ma znaczenie dla optymalizowania nakładów pracy potrzebnych na powstanie programu (optymalizacja kodu, możliwość zrównoleglenia prac nad fragmentami kodu) i jego późniejsze modyfikacje (do przeprowadzenia zmian w grupie powiązanych klas/obiektów wystarczy często "przeprogramowanie" samej klasy/obiektu bazowego).

2.5. Przedstawiony wyżej bazowy typ `Figura` może posłużyć do specjalizacji mającej na celu wydefiniowanie odcinka, okręgu, czy innych figur geometrycznych. Oto jego zastosowanie w definicji klasy reprezentującej okrąg.

Plik nagłówkowy `Okregi.h`:

```
#ifndef H_OKREGI
#define H_OKREGI

#include "Figury.h"

class Okrag : public Figura
{ int r;
  public:
    Okrag(int xSrodek, int ySrodek, int rDlugosc);
    int podaj_r()    {return r;};
    virtual void zmien_r(int rDlugosc);
    virtual void zmien_r_o(int rZmiana);
    virtual void rysuj();
};

#endif
```

Plik z kodem źródłowym `Okregi.cpp`:

```
#include <graphics.h>
#include "Okregi.h"

Okrag::Okrag(int xSrodek, int ySrodek, int rDlugosc) :
                          Figura(xSrodek, ySrodek)
{ r = rDlugosc;
}

void Okrag::zmien_r(int rDlugosc)
{ int byl_widoczny = widoczny();
  if (byl_widoczny) ukryj();
  r = rDlugosc;
  if (byl_widoczny) pokaz();
}
```

```
void Okrag::zmien_r_o(int rZmiana)
{ zmien_r(r + rZmiana);
}

void Okrag::rysuj()
{ circle(x, y, r);
}

int main(void)
{ Okrag o1 = Okrag(0, 2, 3);
  o1.pokaz();
  o1.zmien_r_o(2);
  printf("wsp.x: %d\n wsp.y: %d\n",
        o1.podaj_x(), o1.podaj_y());
  Figura * f3 = new Okrag(0,0,5); // poprawne!
  f3->pokaz();}
```

2.6. Polimorfizm, trzeci paradygmat programowania obiektowego umożliwia tworzenie w typach potomnych tzw. **metod wirtualnych**, nazywających się identycznie jak w typach bazowych, lecz różniących się od swoich odpowiedników znaczeniem.

2.6.1. Polimorfizm, stanowiący uzupełnienie dziedziczenia sprawia, że możliwe jest tworzenie kodu, który będzie wykorzystywany (w przyszłości) w warunkach nie dających się przewidzieć na etapie jego wstępnego projektowania (w teraźniejszości). Mechanizm polimorfizmu wykorzystuje się też do realizacji pewnych metod w trybie nakazowym, abstrahującym od szczegółowego typu obiektu.

2.6.2. Każdy obiektowy język programowania definiuje pewne ograniczenia składniowe związane z używaniem metod wirtualnych. We wszystkich wypadkach dotyczą one sygnatury metody (identyczna liczba parametrów formalnych, zgodność typów odpowiadających sobie parametrów), a często także typu wyniku funkcji. Na ogół nie wymaga się, by metoda wirtualna była przesłonięta we wszystkich klasach na ścieżce prowadzącej od typu bazowego do typu potomnego w hierarchii dziedziczenia.

2.7. W klasie `Figura`, zdefiniowanej w pliku nagłówkowym `Figury.h`, znajduje się bezparametrowa funkcja wirtualna `rysuj`, o typie pustym `void`, przeznaczona do podświetlania punktu. Ma następującą postać:

```
void Figura::rysuj()
{ putpixel(x,y,getcolor());
}
```

W klasie `Okrag`, zdefiniowanej w pliku nagłówkowym `Okregi.h` jako typ potomny klasy `Figura`, funkcję `rysuj` zredefiniowano w postaci:

```
void Okrag::rysuj()
{ circle(podaj_x(), podaj_y(),r);
}
```

umożliwiającej rysowanie na ekranie okręgu o wskazanych współrzędnych środka i wskazanym promieniu.

2.8. Z bazowego typu klasowego `Figura` można wywieść także klasę potomną `Lamana`, przeznaczoną do reprezentacji linii łamanej.

Oto stosowny plik nagłówkowy `Lamane.h`:

```
#ifndef H_LAMANE
#define H_LAMANE

#include "Figury.h"

struct Segment
{ int dx, dy;
  Segment *nast;
};

class Lamana : public Figura
{ Segment *lista_segmentow;
  public:
    Lamana(int xPoczatek, int yPoczatek,
           Segment *sLista);
    virtual ~Lamana();
    Segment *podaj_liste() {return lista_segmentow;};
    virtual void rysuj();
    virtual void nowa_lista(Segment *sLista);
};

Segment *nowy_segment(int dx, int dy, Segment *sLista);

#endif
```

Plik `Lamane.cpp` z kodem źródłowym funkcji klasy `Lamana` i funkcji pomocniczej jest w postaci:

```
#include <graphics.h>
#include <stdlib.h>
#include "Lamane.h"

Lamana::Lamana(int xPoczatek, int yPoczatek,
               Segment *sLista) : Figura(xPoczatek,
                yPoczatek), lista_segmentow(sLista)
{}

Lamana::~Lamana()

{ nowa_lista(nullptr);
}

void Lamana::rysuj()

{ moveto(podaj_x(), podaj_y());
  for (Segment *p = lista_segmentow; p; p = p→nast)
       linerel(p→dx, p→dy);
}

void Lamana::nowa_lista(Segment *sLista)

{ int byla_widoczna = widoczny();
  Segment *p;
  if (byla_widoczna) ukryj();
  while(lista_segmentow)
      { lista_segmentow = (p = lista_segmentow)→nast;
        delete p;
      }
  lista_segmentow = sLista;
  if (byla_widoczna != 0 && lista != nullptr)
      pokaz();
}

Segment *nowy_segment(int dx, int dy,
                              Segment *sLista)

{ Segment *p = new Segment;
  p→dx = dx;
  p→dy = dy;
  p→nast = sLista;
  return p;
}
```

Zadanie:

Zmodyfikuj definicję klasy `Lamana`, przez:
- zmianę trybu dostępu do pola `lista_segmentow` z `private` na `protected`;
- wprowadzenie do niej dodatkowego konstruktora, o nagłówku:

```
Lamana::Lamana(int xPoczatek, int yPoczatek)
```

Zdefiniuj klasę `Kwadrat`, dziedziczącą po klasie `Lamana`, z dodatkowymi polami `xPrzeciwny` i `yPrzeciwny`, przechowującymi współrzędne tego wierzchołka kwadratu, który jest położony naprzeciwko punktu wyjściowego (`x` i `y`). Konstruktor klasy `Kwadrat` ma odpowiadać za inicjalizację tych czterech współrzędnych.
Uzupełnij definicję klasy `Kwadrat` o niezbędne pola i metody. W szczególności:
- dodaj do klasy prywatną metodę `wyznacz_l_segmentow`, odpowiedzialną za umieszczenie właściwej wartości (listy segmentów) w polu `lista_segmentow`,
- przesłoń w niej metodę wirtualną `rysuj` z klasy `Lamana` – nową metodą `rysuj` bazującą na wykorzystaniu dwóch przeciwległych wierzchołków kwadratu i metody prywatnej `wyznacz_l_segmentow`.