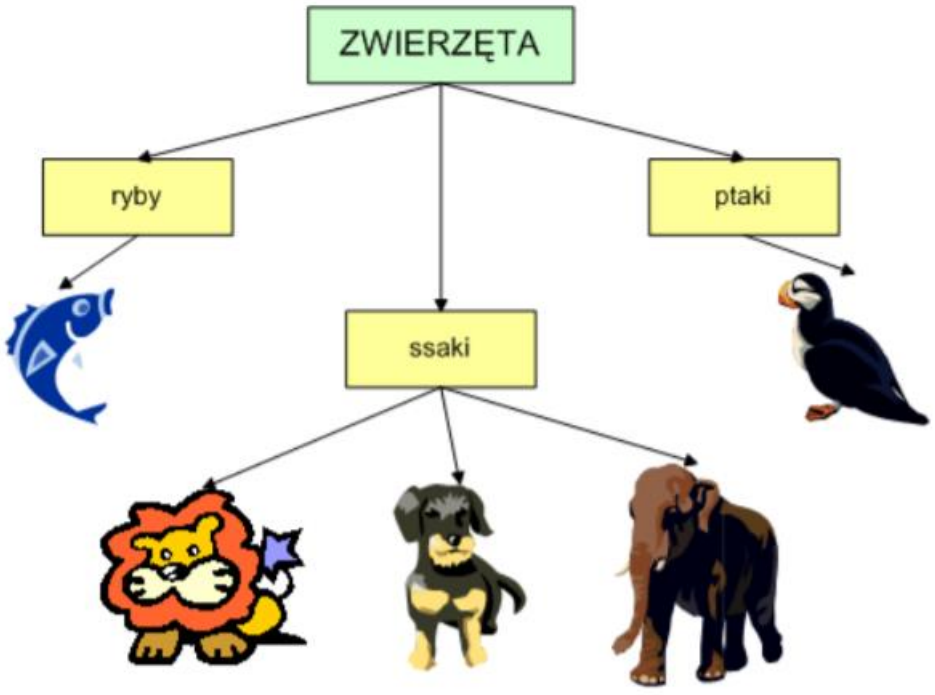
Laboratorium Podstaw Informatyki

1. Przebieg ćwiczenia laboratoryjnego

Dziedziczenie

Kontynuując zagadnienie obiektowości z poprzedniego laboratorium należy zauważyć, że jednym z powodów, dla którego techniki obiektowe zyskały taką popularność, jest znaczący postęp w kwestii ponownego wykorzystywania raz napisanego kodu oraz rozszerzania i dostosowania go do własnych potrzeb. Kod oparty na programowaniu obiektowym łatwiej poddaje się modyfikacji przygotowującej go do ponownego użycia. Jednym z tych narzędzi jest tytułowy mechanizm dziedziczenia. W połączeniu z technologią funkcji wirtualnych oraz polimorfizmu daje on niezwykle szerokie możliwości, o które zostaną przedstawione na kolejnych laboratoriach.

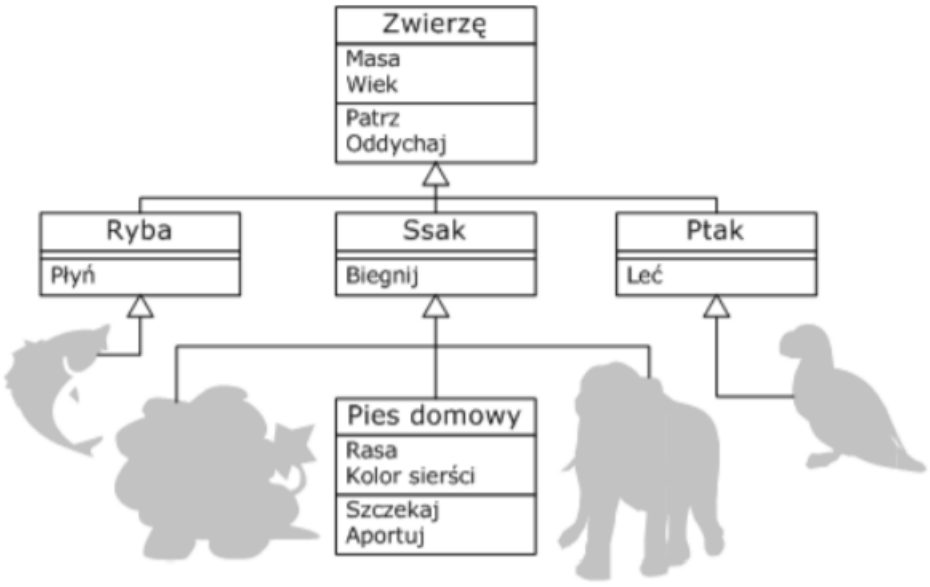
Mechanizm dziedziczenia najłatwiej przejawia się w klasyfikacji biologicznej. Widząc na przykład psa wiemy, że nie tylko należy on do gatunku zwanego psem domowym, lecz także do gromady znanej jako ssaki (wraz z końmi, słoniami, lwami, małpami, ludźmi itd.). Te z kolei, razem z gadami, ptakami czy rybami należą do kolejnej, znacznie większej grupy organizmów zwanych po prostu zwierzętami. Nasz pies jest zatem jednocześnie: psem domowym, ssakiem i zwierzęciem.



Gdyby był obiektem w programie, wtedy musiałby należeć aż do trzech klas naraz. Byłoby to oczywiście niemożliwe, jeżeli wszystkie miałyby być wobec siebie równorzędne. Tutaj jednak tak nie jest: występuje między nimi hierarchia, jedna klasa pochodzi od drugiej. Zjawisko to nazywamy dziedziczeniem. Dziedziczenie (ang. inheritance) to tworzenie nowej klasy na podstawie jednej lub kilku istniejących wcześniej klas bazowych.

Wszystkie klasy, które powstają w ten sposób (nazywamy je pochodnymi), posiadają pewne elementy wspólne. Części te są dziedziczone z klas bazowych, gdyż tam właśnie zostały zdefiniowane. Ich zbiór może jednak zostać poszerzony o pola i metody specyficzne dla klas pochodnych. Będą one wtedy współistnieć z "dorobkiem" pochodzącym od klas bazowych, ale mogą oferować dodatkową funkcjonalność. Tak w teorii wygląda system dziedziczenia w programowaniu obiektowym. Najlepiej będzie, jeżeli teraz przyjrzymy się, jak w praktyce może wyglądać jego zastosowanie. Wracając do przykładu ze zwierzętami. Chcąc stworzyć programowy odpowiednik zaproponowanej hierarchii, musielibyśmy zdefiniować najpierw odpowiednie klasy bazowe. Następnie odziedziczylibyśmy ich pola i metody w klasach pochodnych i dodali nowe, właściwe tylko im. Powstałe klasy same mogłyby być potem bazami dla kolejnych, jeszcze bardziej wyspecjalizowanych typów. Idąc dalej tą drogą dotarlibyśmy wreszcie do takich klas, z których sensowne byłoby już tworzenie normalnych obiektów. Pojęcie klas bazowych i klas pochodnych jest zatem względne : dana klasa może wprawdzie pochodzić od innych, ale jednocześnie być bazą dla kolejnych klas. W ten sposób ustala się wielopoziomowa hierarchia, podobna zwykle do drzewka.

Ilustracją tego procesu może być poniższy diagram:



Wszystkie przedstawione na nim klasy wywodzą się z jednej, nadrzędnej wobec wszystkich: jest nią naturalnie klasa Zwierzę . Dziedziczy z niej każda z pozostałych klas - bezpośrednio , jak Ryba , Ssak oraz Ptak , lub pośrednio - jak Pies domowy. Tak oto tworzy się kilkupoziomowa klasyfikacja oparta na mechanizmie dziedziczenia.

Odziedziczanie składowych na przykładzie prostej hierarchii klas zwierząt.

U jej podstawy leży "najbardziej bazowa" klasa Zwierzę. Zawiera ona dwa pola, określające masę i wiek zwierzęcia, oraz metody odpowiadające za takie czynności jak widzenie i oddychanie. Składowe te mogły zostać umieszczone tutaj, gdyż dotyczą one wszystkich interesujących nas zwierząt i będą miały sens w każdej z klas pochodnych.
Tymi klasami, bezpośrednio dziedziczącymi od klasy Zwierzę , są Ryba , Ssak oraz Ptak . Każda z nich niejako "z miejsca" otrzymuje zestaw pól i metod, którymi legitymowało się bazowe Zwierzę. Klasy te wprowadzają jednak także dodatkowe, własne metody: i tak Ryba może pływać, Ssak biegać, zaś Ptak latać. Wreszcie, z klasy Ssak dziedziczy najbardziej interesującą nas klasę, czyli Pies domowy. Przejmuje ona wszystkie pola i metody z klasy Ssak, a więc pośrednio także z klasy Zwierzę. Uzupełnia je przy tym o kolejne składowe, właściwe tylko sobie. Ostatecznie więc klasa Pies domowy zawiera znacznie więcej pól i metod niż mogłoby się z początku wydawać:

Wykazuje poza tym pewną budowę wewnętrzną: niektóre jej pola i metody możemy bowiem określić jako własne i unikalne, zaś inne są odziedziczone po klasie bazowej i mogą być wspólne dla wielu klas. Nie sprawia to jednak żadnej różnicy w korzystaniu z nich: funkcjonują one identycznie, jakby były zawarte bezpośrednio wewnątrz klasy.

Dziedziczenie w C++

Pozyskawszy ogólne informacje o dziedziczeniu jako takim, można zobaczyć, jak idea ta została przełożona na język C++. Przedstawione zostanie sposób definiowania nowej klasy w oparciu o już istniejące oraz jakie dodatkowe efekty są z tym związane. Mechanizm dziedziczenia jest w C++ bardzo rozbudowany, o wiele bardziej niż w większości pozostałych języków zorientowanych obiektowo. Udostępnia on kilka szczególnych możliwości, które być może nie są zawsze niezbędne, ale pozwalają na dużą swobodę w definiowaniu hierarchii klas.
Poznanie ich wszystkich nie jest konieczne, aby sprawnie korzystać z dobrodziejstw programowania obiektowego. Definicja klasy składa się przede wszystkim z listy deklaracji

jej pól oraz metod, podzielonych na kilka części wedle specyfikatorów praw dostępu. Najczęściej każdy z tych specyfikatorów występuje co najwyżej w jednym egzemplarzu, przez co składnia definicji klasy wygląda następująco:

```
class nazwa_klasy
{
private :
deklaracje_prywatne
protected :
deklaracje_chronione
public :
deklaracje_publiczne
};
```

Nieprzypadkowo pojawił się tu nowy specyfikator protected. Jego wprowadzenie związane jest ściśle z pojęciem dziedziczenia. Pojęcie to wpływa na dwa pozostałe rodzaje praw dostępu do składowych klasy. Zbierzmy więc je wszystkie w jednym miejscu, wyjaśniając definitywnie znaczenie każdej z etykiet:

- **private** : poprzedza deklaracje składowych, które mają być dostępne jedynie dla metod definiowanej klasy. Oznacza to, iż nie można się do nich dostać, używając obiektu lub wskaźnika na niego oraz operatorów wyłuskania . lub -> . Ta wyłączność znaczy również, że prywatne składowe nie są dziedziczone i nie ma do nich dostępu w klasach pochodnych, gdyż nie wchodzą w ich skład.
- specyfikator **protected** ("chronione") także nie pozwala, by użytkownicy obiektów naszej klasy poznawali wartości w opatrzonych nimi polach i metodach. Jak sama nazwa wskazuje, są one chronione przed takim dostępem z zewnątrz. Jednak w przeciwieństwie do deklaracji private , składowe zaznaczone przez protected są dziedziczone i występują w klasach pochodnych, będąc dostępnymi dla ich własnych metod. Pamiętajmy zatem, że zarówno private , jak i protected nie pozwala , aby oznaczone nimi składowe klasy były dostępne na zewnątrz. Ten drugi specyfikator zezwala jednak na dziedziczenie pól i metod.
- public jest najbardziej liberalnym specyfikatorem. Nie tylko pozwala na odziedziczanie swych składowych, ale także na udostępnianie ich szerokiej rzeszy obiektów poprzez operatory wyłuskania.

Powyższe opisy przedstawia przykład, który będzie bardziej przemawiał do wyobraźni. Dana jest klasa prostokąta:

```
class CRectangle
{
private :
// wymiary prostokąta
float m_fSzerokosc, m_fWysokosc;
protected :
// pozycja na ekranie
float m_fX, m_fY;
public :
// konstruktor
CRectangle() { m_fX = m_fY = 0.0 ;
m_fSzerokosc = m_fWysokosc = 10.0 ; }
//
// metody
float Pole() const { return m_fSzerokosc * m_fWysokosc; }
float Obwod() const { return 2 * (m_fSzerokosc+m_fWysokosc); }
};
```

Opisują go cztery liczby, wyznaczające jego pozycję oraz wymiary. Współrzędne X oraz Y są polami chronionymi, zaś szerokość oraz wysokość - prywatnymi. Dlaczego właśnie tak? Otóż powyższa klasa będzie również bazą dla następnej. Pamiętamy z geometrii, że szczególnym rodzajem prostokąta jest kwadrat. Ma on wszystkie boki o tej samej długości, zatem nielogiczne jest stosowanie do nich pojęcia szerokości i wysokości. Wielkość kwadratu określa bowiem tylko jedna liczba, więc definicja odpowiadającej mu klasy może wyglądać następująco:

```
class CSquare : public CRectangle // dziedziczenie z CRectangle
{
private :
// zamiast szerokości i wysokości mamy tylko długość boku
float m_fDlugoscBoku;
// pola m_fX i m_fY są dziedziczone z klasy bazowej, więc nie ma
// potrzeby ich powtórnego deklarowania
public :
// konstruktor
CSquare { m_fDlugoscBoku = 10.0 ; }
//
// nowe metody
float Pole() const { return m_fDlugoscBoku * m_fDlugoscBoku; }
float Obwod() const { return 4 * m_fDlugoscBoku; }
};
```

Dziedziczy ona z CRectangle, skoncentrujmy się na konsekwencjach owego dziedziczenia. Pola m_fSzerokosc oraz m_fWysokosc były w klasie bazowej oznaczone jako prywatne, zatem ich zasięg ogranicza się jedynie do tej klasy. W pochodnej CSquare nie ma już po nich śladu, zamiast tego pojawia się bardziej naturalne pole m_fDlugoscBoku z sensowną dla kwadratu wielkością.

Związane są z nią także dwie nowe-stare metody, zastępujące te z CRectangle . Do obliczania pola i obwodu wykorzystujemy bowiem samą długość boku kwadratu, nie zaś "jego" szerokośc i wysokość, których w klasie w ogóle nie ma.
W definicji CSquare nie ma także deklaracji m_fX oraz m_f Y. Nie znaczy to jednak, że klasa tych pól nie posiada, gdyż zostały one po odziedziczone z bazowej CRectangle . Stało się tak oczywiście za sprawą specyfikatora protected. Należy używać specyfikatora protected , kiedy chcemy uchronić składowe przed dostępem z zewnątrz, ale jednocześnie mieć je do dyspozycji w klasach pochodnych.

Zadanie do realizacji:

1. Utwórz klasę Zwierzak zawierającą właściwości: Masa, Wiek oraz metody Patrz(), Oddychaj()
2. Utwórz klasy: Ryby, Ssaki, Ptaki dziedziczące po klasie Zwierzak, posiadające metody:
   - Ssak - Biegaj()
   - Ptak - Lataj()
   - Ryba - Plywaj()
3. Utwórz klasy Welonek, Nemo, Karp dziedziczące po klasie Ryby
4. Utwórz klasy Lew, Pies, Slon dziedziczące po klasie Ssaki
5. Utwórz klasy Papuga, Kanarek, Golab dziedziczące po klasie Ptaki
6. W klasie Zwierzak dodaj metodę Zyj() znajdująca się w sekcji publicznej, ustaw operatory dziedziczenia tak aby każda kolejna klasa dziedziczyła tą metodę (Metoda ma za zadanie wyświetlenie na ekranie tekstu „Żyję i mam się dobrze”).
7. W pętli głównej programu utwórz kilka obiektów różnych klas i odnotuj sekwencję wykonywania konstruktorów. Wywołaj metodę Zyj() a następnie dodaj taką samą metodę Zyj() w wybranej klasie potomka. Ponownie wywołaj metodę Zyj() na obiekcie tej zmodyfikowanej klasy.