Funkcje wirtualne, dziedziczenie wielokrotne

Wykład 7

Konwersje standardowe przy dziedziczeniu

- Wskaźnik lub referencja do obiektu klasy pochodnej może być niejawnie przekształcony na wskaźnik (referencję) dostępnej jednoznacznie klasy podstawowej (tylko przy dziedziczeniu publicznym)
- Konwersje zachodzą
  - Przy przesyłaniu argumentów do funkcji – funkcja która powinna otrzymać wskaźnik (referencję) do obiektu klasy podstawowej może otrzymać wskaźnik (referencję) do obiektu klasy pochodnej
  - Przy zwracaniu przez funkcję rezultatu - funkcja, która powinna zwrócić wskaźnik (referencję) do obiektu klasy podstawowej może zwrócić wskaźnik (referencję) do klasy pochodnej

Konwersje standardowe przy dziedziczeniu...

- Konwersje zachodzą...
  - Przy przeładowanych operatorach – analogicznie jak w przypadku funkcji
  - W wyrażeniach inicjalizujących – do konstruktora kopiującego spodziewającego się referencji do obiektu klasy podstawowej można wysłać referencję do obiektu klasy pochodnej
- Obiekty klas pochodnych mogą być traktowane jak obiekty swych klas podstawowych tylko wtedy, gdy pracujemy na ich adresach (referencjach)
- UWAGA: Do funkcji spodziewającej się adresu tablicy obiektów klasy podstawowej nie można wysłać adresu tablicy obiektów klasy pochodnej
- Przykład cpp_7.1

Funkcje wirtualne

- Dopiero przy funkcjach wirtualnych pojawia się możliwość faktycznego programowania (z)orientowanego obiektowo
- Realizacja klas podstawowych i pochodnych w zasadzie jest taka sama ale
  - Pojawia się słowo `virtual` przy nazwach funkcji składowych (niekoniecznie wszystkich), które umożliwia wykonywanie różnych funkcji w zależności od typu obiektu na rzecz, którego chcemy taką funkcję wywołać
  - Wykorzystanie funkcji wirtualnych może w znakomity sposób ułatwić i uprościć nam pracę na projektem

Posługiwanie się funkcjami wirtualnymi

- Możliwe jest ustawienie wskaźnika (referencji) typu klasy podstawowej tak, żeby pokazywał na obiekt klasy pochodnej
  - Wynika to z uogólnienia tzn. klasa pochodna jest szczególnym (bardziej wyspecjalizowanym) typem klasy podstawowej
    - Np. mając wskaźnik do pojazdów możemy nim pokazywać na samochód lub nawet na „malucha" i nie jest to niezgodne z naszym wyobrażeniem o rzeczywistości (samochód jest rodzajem pojazdu)
- Sytuacja odwrotna nie jest już prawdziwa
  - Np. mając wskaźnik do samochodu nie możemy nim (w ogólności) pokazywać na pojazd, gdyż pojazdem może np. być rower

Różnica między funkcją wirtualną, a zwykłą

- Mając wskaźnik klasy podstawowej pokazujący na obiekt klasy pochodnej wywołanie metody
  - Niewirtualnej (zwykłej) spowoduje wywołanie odpowiedniej funkcji składowej w klasie podstawowej – zupełnie normalna sytuacja
    - Przykład cpp_7.2
  - Wirtualnej spowoduje wywołanie odpowiedniej funkcji składowej uzależnionej od typu obiektu, na który w danym momencie pokazuje wskaźnik
    - Przykład cpp_7.3
- Dodanie przymiotnika **`virtual`** przy funkcji składowej w klasie podstawowej mówi, że od tego momentu wszystkie dalsze pokolenia będą tą funkcję mieć wirtualną
  - Tylko wtedy kiedy taka funkcja jest identyczna tzn. posiada taką samą nazwę, przyjmuje takie same parametry oraz zwraca taki sam typ

Polimorfizm

- Polimorfizm – wielość form
  - Jest to wykazywanie przez metodę różnych form działania w zależności od tego jaki typ obiektu aktualnie jest wskazywany przez wskaźnik lub referencję
  - `Shape->Rys();` oznacza w zależności od kontekstu
    - `Shape->Shape::Rys();`
    - `Shape->Circ::Rys();`
    - `Shape->Rec::Rys();`
  - Sama funkcja wirtualna polimorfizmu nie wykazuje
  - Funkcja nie jest polimorficzna, ale tylko jej wywołanie jest

Jaki jest pożytek z polimorfizmu

- Program jest rozszerzalny o nowe obiekty (typy), a ich dodanie nie wymaga zmian w już istniejącym kodzie
  - W szczególności w miejscach, gdzie decyduje się jakiej klasy jest obiekt pokazywany przez wskaźnik lub nazywany referencją
  - Nie wymaga instrukcji wyboru
- Jednak są również wady
  - „Inteligentne" zachowanie kompilatora w stosunku do funkcji wirtualnych okupione jest dłuższym czasem ich wywoływania
  - Oraz tym, iż obiekty danej klasy są większe od odpowiadających im obiektów klasy, która nie zawiera funkcji wirtualnych
  - Dlatego funkcje nie są z założenia wirtualne (w C++)
- Przykład cpp_7.4

Wczesne i późne wiązanie

- Wczesne wiązanie następuje w sytuacji kiedy wywoływane są zwykłe funkcje i na etapie kompilacji wywołania funkcji powiązane zostają z adresami, pod którymi te funkcje się znajdują
  - Inaczej wiązanie w czasie kompilacji
- Późne wiązanie występuje w sytuacji kiedy posługujemy się funkcjami wirtualnymi. Kiedy kompilator widzi funkcję wirtualną to nie podstawia określonego adresu, ale generuje odpowiedni kod pozwalający na wybór określonej wersji funkcji na etapie wykonania programu
- W wywołaniu funkcji wirtualnych może wystąpić wczesne wiązanie jeżeli już na etapie kompilacji wiadomo dokładnie, która wersja funkcji ma zostać wywołana
  - Wywołanie na rzecz obiektu (`obiekt.funkcja()`)
  - Jawne użycie kwalifikatora zakresu (`wskaźnik->klasa::funkcja()`)
    - Należy stosować jeżeli sięgamy do składników klasy podstawowej z funkcji składowych klasy pochodnej
  - Wywołanie funkcji z konstruktora lub destruktora klasy podstawowej

Klasy abstrakcyjne

- Klasa abstrakcyjna to taka klasa, która nie reprezentuje żadnego konkretnego obiektu
  - Np. pojazd, figura geometryczna itp...
- Takie klasy tworzy się, aby po nich dziedziczyć
- W pewnym sensie są to niedokończone klasy
- Tworzymy funkcje wirtualne, których będziemy używać, ale implementacje tych funkcji pozostawiamy klasom pochodnym
  - Np. w klasie figura, powinna znaleźć się funkcja rysuj, mimo iż jeszcze nie wiadomo jak taką figurę narysować

Funkcje czysto wirtualne

- Funkcje czysto wirtualne mają ścisły związek z klasami abstrakcyjnymi
- Skoro nie ma sensu tworzyć obiektów klasy abstrakcyjnej to dla przykładowej funkcji rysuj nie jest potrzebna implementacja w klasie podstawowej
- Deklaracja funkcji czysto wirtualnej
  - `virtual void rysuj() = 0;`
- Dopóki klasa ma chociaż jedną funkcje czysto wirtualną to NIE MOŻNA stworzyć żadnego obiektu takiej klasy
- Przykład cpp_7.5

Funkcje czysto wirtualne...

- Brak możliwości stworzenie obiektu klasy abstrakcyjnej z funkcją czysto wirtualną odnosi się do wszystkich sytuacji
  - Oczywiste: stworzenie obiektu np. `figura a;`
  - Wywołanie funkcji przez wartość `void fun(figura a);`
    - Bo tworzony jest obiekt automatyczny
  - Zwracanie przez funkcję obiektu klasy figura
  - Nie można wywołać konwersji do typu figura
- Natomiast możemy posługiwać się wskaźnikami i referencjami
  - Tutaj uwidaczniają się wielkie możliwości programowania zorientowanego obiektowo

Aspekty tworzenie funkcji wirtualnych

- Jeżeli funkcja ma być wirtualna to
  - Możemy zadeklarować ją jako zwykłą funkcję wirtualną (nie czysto wirtualna)
    - Musimy dostarczyć implementację
    - Jeżeli w klasie pochodnej nie będzie implementacji to zostanie wywołana funkcja z klasy podstawowej
  - Możemy zadeklarować ją jako funkcję czysto wirtualną bez implementacji
    - Jeżeli klasa pochodna nie dostarczy implementacji to staje się automatycznie klasą abstrakcyjną
  - Możemy zadeklarować ją jako funkcję czysto wirtualną, ale pomimo to dostarczyć jej implementację
    - Ale tylko poza ciałem klasy, standard nie dopuszcza definicji funkcji czysto wirtualnej bez osobnej deklaracji
    - Po co skoro i tak się nigdy nie wykona???
- Przykład cpp_7.6

Wirtualny destruktor

- Wirtualny destruktor może istnieć mimo, iż jego nazwa w klasie pochodnej jest inna niż w klasie podstawowej
  - Jest to jedyny wyjątek kiedy funkcje wirtualne mogą mieć różne nazwy
- Używając wskaźnika lub referencji klasy podstawowej pokazującego na obiekt klasy pochodnej możemy chcieć zniszczyć właśnie obiekt klasy pochodnej
  - Wtedy oczywiście powinien ruszczyć do pracy destruktor również klasy pochodnej, a nie tylko podstawowej
- Jeżeli klasa ma być klasą bazową to zawsze powinna deklarować destruktor jako wirtualny
- Przykład cpp_7.7

C++11 override i final

- W C++11 dodano nowe możliwości kontroli dziedziczenia i przesłaniania metod
- Ważne jest to w aspekcie dobrego designu i możliwości kontroli czy nie popełniamy logicznego błędu
- Mając funkcję wirtualną w klasie pochodnej można dodać (powtórzyć) do jej deklaracji słowo **`virtual`**
  - Nie jest to zalecane
- Chcąc wymusić sprawdzenie czy funkcja, którą deklarujemy przesłania metodę wirtualną z klasy bazowej należy użyć słowa **`override`**
  - Przykład cpp_7.7a
- Z drugiej strony czasami istnieje potrzeba aby zagwarantować aby wirtualna metoda nie była przesłaniana w klasach pochodnych
  - Wtedy używa się słowa **`final`**
- Podobnie można sobie wyobrazić że klasa ma sam nie pozawalać aby po niej dziedziczyć
  - Wtedy do definicji klasy dodajemy słowo **`final`**
  - Przykład cpp_7.7b

Konwersje jawne

- Podczas wykorzystywania polimorfizmu możemy natrafić na pewien problem
  - Posługując się obiektem za pomocą uogólnienia nie znamy jego rzeczywistego typu
  - W niektórych sytuacjach potrzebna nam jest dokładana informacja o faktycznym typie obiektu
    - Kiedy?
- W C++ możemy jednak odzyskać informację o typie obiektu za pomocą mechanizmu *RTTI* (*run-time type information*)
  - Spróbować przekształcić obiekt na jego rzeczywisty typ korzystając z rzutowania
  - Ustalić klasę obiektu

Rzutowanie `dynamic_cast`

- Posługując się operatorem rzutowania `dynamic_cast` możemy przywrócić obiektowi jego rzeczywisty typ
  - `dynamic_cast<Klasa&>(obiekt);`
  - Typ do którego przekształcamy obiekt musi być referencją lub wskaźnikiem
- Przy rzutowaniu możemy bezpośrednio odwołać się do składnika klasy do której rzutujemy
  - `dynamic_cast<Klasa&>(obiekt).fun();`
  - Jeżeli rzutowanie nie będzie możliwe to zostanie wyrzucony wyjątek (`std::bad_cast`)
- Z rzutowaniem `dynamic_cast` zwiazany jest dodatkowy narzut czasowy w stosunku do `static_cast`
- Przykład cpp_7.8

Zapytanie o typ

- Za pomocą operatora **typeid** możemy ustalić typ obiektu podczas wykonania programu
  - Musimy dołączyć nagłówek **typeinfo**
  - **cout << typeid(obj).name();**
  - **if(typeid(obj) == typeid(Klasa))**
- Operator **typeid** zwraca obiekt **std::type_info**, dla którego zdefiniowane są między innymi
  - Funkcja **name()** - zwraca nazwę klasy (dokładna postać zależy od implementacji)
  - Operatory **==** i **!=** pozwalające na porównanie dwóch typów
- Przykład cpp_7.9

Mechanizm *RTTI* i projektowanie klas

- Mechanizm *RTTI* należy używać z rozwagą (w szczególności **`typeid`**)
  - Na pewno nie należy używać instrukcji warunkowych, w których **`typeid`** służy do określania jaką funkcję wywołać
- Używanie tego mechanizmu sprzyja tworzeniu nieprawidłowo zaprojektowanego kodu, dlatego należy go unikać
- Jeżeli tylko jest możliwe to należy używać funkcji wirtualnych i projektować klasy tak, żeby możliwe było wywołanie odpowiedniej funkcji składowej
- W szczególnym przypadku możemy posłużyć się rzutowaniem **`dynamic_cast`**
- Przykład cpp_7.10 (podobnie jak cpp_7.8)

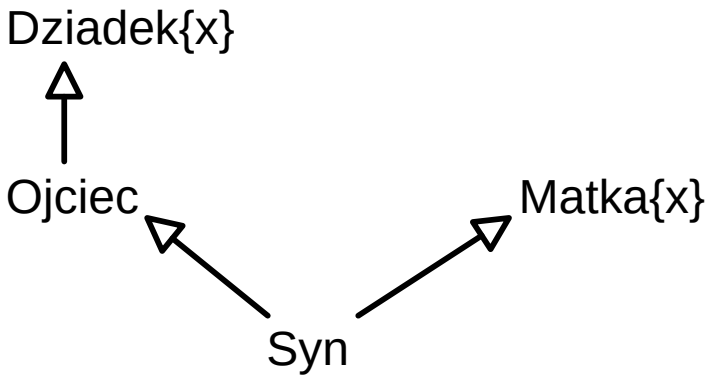
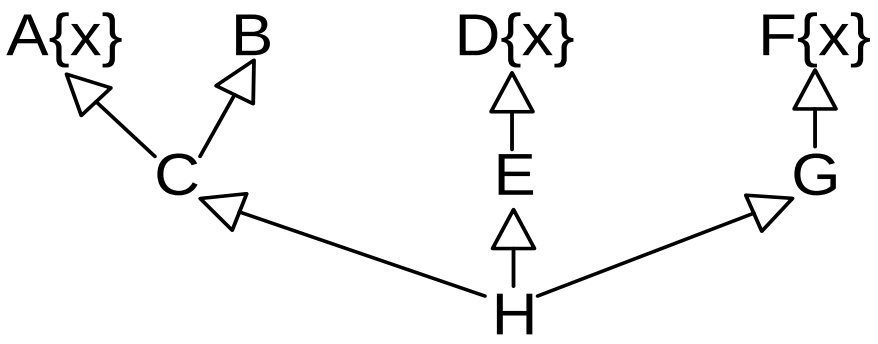
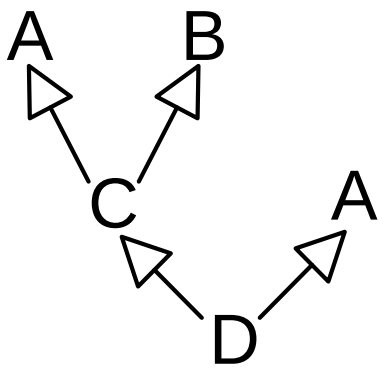
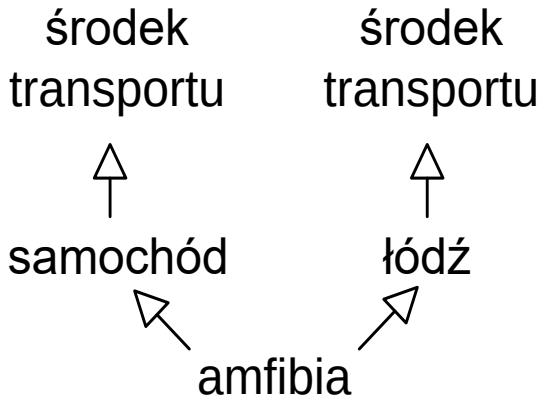
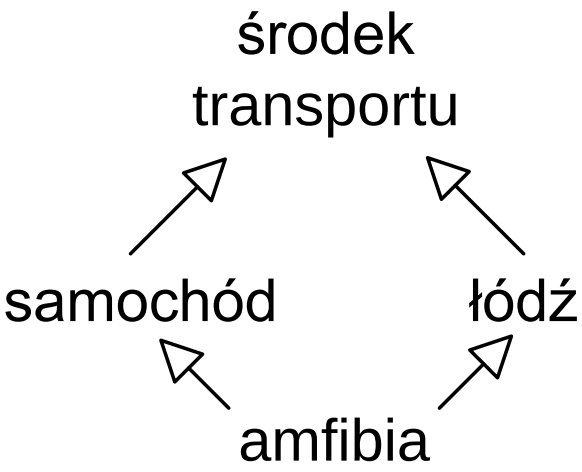
Dziedziczenie wielokrotne

- Klasa może dziedziczyć po więcej niż jednej klasie bazowej, wtedy mówimy o dziedziczeniu wielokrotnym
  - Nie jest to cecha dostępna we wszystkich językach programowania zorientowanego obiektowo
- Tego typu dziedziczenie jest rzadziej stosowane
  - Zaletą jego jest możliwość połączenia ze sobą zupełnie niezależnych od siebie klas
  - `class C : public A, public B {...}`
- Przykład cpp_7.11

Aspekty dziedziczenia wielokrotnego

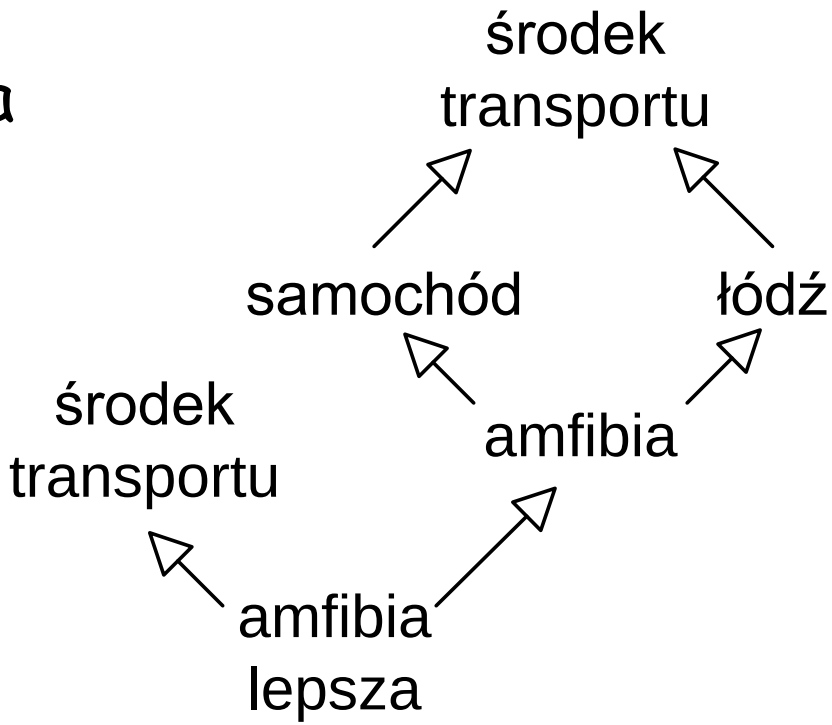
- Dana klasa podstawowa może znaleźć się na liście dziedziczenia tylko raz
- Definicja klasy umieszczona na liście pochodzenia musi być wcześniej znana, nie wystarcza deklaracja zapowiadająca typu `class A;`
- Na liście pochodzenia przed nazwami klas pojawiają się określenia sposobu dziedziczenia
  - Domniemanie jest identyczne jak w dziedziczeniu jednokrotnym, czyli `private`

Konstruktory przy dziedziczeniu wielokrotnym

- Konstruktor klasy pochodnej na liście inicjalizacyjnej może zawierać wywołania konstruktorów swoich bezpośrednich klas podstawowych
  - ```
    class C : public A, public B {
              C() : B(), A() {...}
    }
    ```
- Kolejność wywoływania konstruktorów uzależniona jest od kolejności umieszczenia nazw klas podstawowych na liście pochodzenia
- Przykład cpp_7.12
    ```

Wieloznaczność przy dziedziczeniu wielokrotnym

- Z wieloznacznością mamy do czynienia w sytuacji gdy wyrażenie odnoszące się do składnika klasy podstawowej równie dobrze może odnosić się do innego składnika drugiej klasy podstawowej
- `A{x,...}, B{x,...}`
  `class C : public A, public B`
- Aby odniesienie do składnika było możliwe musimy użyć operatora zakresu `::`
  - `A::x;` lub `B::x;`
- Należy pamiętać, że najpierw sprawdzana jest jednoznaczność, a dopiero później dostęp!!!
  - Czyli jeśli nawet składnik jednej klasy jest prywatny to mimo to kompilator zaprotestuje
- Przykład cpp_7.13

Wieloznaczność, a pokrewieństwo

- Bliższe pokrewieństwo usuwa wieloznaczność
- Wieloznaczności nie ma bo droga do nazwy **x** w klasie podstawowej **Matka** jest znacząco krótsza niż do identycznego składnika w klasie **Dziadek**



Wieloznaczność, a pokrewieństwo...

- Jeżeli mimo wszystko musimy posłużyć się kwalifikatorem zakresu to nie musimy podawać dokładnego określenia, w której klasie interesujący nas obiekt się znajduje
- Wystarczy podać zakres od którego zaczną się poszukiwania
  - `A::x == C::x`
  - `D::x == E::x`
  - `F::x == G::x`



Wirtualne klasy bazowe

- Na liście bezpośrednich przodków dana klasa może pojawić się tylko i wyłącznie jeden raz
- Ale nic nie stoi na przeszkodzie, żeby klasa znalazła się wielokrotnie na wyższym poziomie dziedziczenia
  - W przypadku dziedziczenia zwykłego w klasie pochodnej dostaniemy zwielokrotnioną tą samą informację
- W przypadku drugiego grafu dostęp do składników nie jest jednoznaczny





Wirtualne klasy bazowe...

- Istnieje jednak stosunkowo proste rozwiązanie na duplikowanie informacji w klasie pochodnej
  - Podstawowa klasa wirtualna
- Słowo **`virtual`** pojawia się na liście dziedziczenia przed nazwą klasy
- Przy takim dziedziczeniu graf wygląda następująco



Wirtualne klasy bazowe...

- Deklarowanie dziedziczenia wirtualnego
  - ```
    class samochod: virtual public srodek_trans
    {...};
    class lodz: virtual public srodek_trans
    {...};
    class amfibia: public samochod, public lodz
    {...};
    ```
- Otrzymujemy zmniejszoną klasę amfibia bez duplikatów
- Nie ma ryzyka niejednoznaczności, mimo iż do składników możemy dostać się na dwa sposoby
- Przykład cpp_7.14
    ```

Dziedziczenie wirtualne i niewirtualne jednocześnie

- Ta sama klasa może być odziedziczona wirtualnie i niewirtualnie
- W pokazanym przykładzie **`amfibia_lepsza`** posiada
  - Wspólny zestaw składników dla wszystkich wirtualnych dziedziczeń
  - Oraz osobny zestaw odziedziczony w zwykły sposób



Klasa finalna

- Standard C++ nie zawiera słowa kluczowego **`final`**, które w łatwy sposób pozwala na stworzenie klasy po której nie będzie możliwe dalsze dziedziczenie
- Z pomocą wirtualnego dziedziczenia możliwe jest stworzenie klasy finalnej
- Należy wykorzystać to iż konstruktor klasy podstawowej wirtualnej wywoływany jest z klasy najbardziej pochodnej
- Co jeszcze jest niezbędne?

Konstrukcja i inicjalizacja w klasach wirtualnych

- Za konstrukcję wirtualnego dziedzictwa odpowiada klasa najbardziej pochodna
  - Klasa najbardziej pochodna to taka, która tworzy obiekt nie będący już pod-obiektem innego obiektu, czyli jest najniżej w hierarchii dziedziczenia
  - Stoi to poniekąd w sprzeczności z tym, że na liście inicjalizacyjnej stoi wywołanie konstruktora klasy bezpośrednio nadrzędnej
    - W takim przypadku kompilator i tak uruchomi konstrukcję obiektu odziedziczonego wirtualnie z klasy najbardziej pochodnej
- Przykład cpp_7.15

Kiedy dziedziczyć, a kiedy osadzać składniki (klasy)

- Dziedziczenie wybieramy w sytuacji kiedy dany obiekt jest rodzajem innego
  - Np. kwadrat jest rodzajem figury geometrycznej, samochód jest rodzajem pojazdu
- Zawieranie obiektów składowych używamy w sytuacji, gdy jeden obiekt składa się z innych obiektów
  - Np. samochód składa się (miedzy innymi) z czterech kół, radio składa się z tranzystorów
- Nie zawsze jest oczywiste, czy lepsze jest dziedziczenie, czy też może zawieranie

Aspekty dziedziczenia

- Klasa pochodna powinna przesłaniać tylko te funkcje, które zostały zadeklarowane jako wirtualne w klasie podstawowej
- Jeżeli klasa pochodna ma zostać klasą bazową to powinna także wszystkie funkcje, które mogą być przysłonięte deklarować jako wirtualne
- Przesłaniane funkcje powinny mieć te same domyślne wartości parametrów w klasie pochodnej, co w klasie podstawowej
- Klasa bazowa oczywiście powinna posiadać wirtualny destruktor
- Jedynie publiczne dziedziczenie określa relacje generalizacji. Pozostałe przypadki dziedziczenia umożliwiają tylko wykorzystanie już istniejącego kodu

Pułapki projektowania z użyciem dziedziczenia

- Struktura hierarchii klas stanowi jedną z fundamentalnych decyzji podejmowanych na etapie projektowania. Dlatego bardzo ważne jest popełnienie możliwe jak najmniejszej liczby błędów (a najlepiej w ogóle)
- Ograniczanie dziedziczenia
  - Jaki jest związek między klasą kwadrat i prostokąt?
    - W dziedziczeniu kwadrat nie jest prostokątem, ponieważ nie wszystkie operacje dostępne dla prostokąta można wykonać dla kwadratu (dodatkowy warunek na długość boków)
  - Rozwiązanie
    - Zignorowanie konsekwencji – odpowiedzialność za poprawność kodu spada na programistę
    - Eliminacja konsekwencji – np. wyświetlenie błędu, wyrzucenie wyjątku itp.
    - Eliminacja przyczyn błędu – inny projekt klas np. klasa bazowa opisująca czworokąty

Błędy projektowania

- Dziedziczenie zmieniające wartość
  - Np. ułamek zwykły (6/5) oraz ułamek zwykły z liczbą całkowitą (1 i 1/5)
    - Nowy ułamek wprowadza nową składową, ale zmienia senes odziedziczonej składowej
  - Stan składowych klasy reprezentujących pewną wartość nie może zmieniać się w klasie pochodnej
  - W tym przypadku powinnyśmy mówić o zawieraniu, a nie o dziedziczeniu
  - Ewentualnie odziedziczyć tak, żeby wartości nie uległy zmianie

Błędy projektowania...

- Dziedziczenie zmieniające interpretację wartości
  - Np. mamy klasę ułamek reprezentująca tylko ułamki dodatnie
  - Dziedziczymy tą klasę, aby stworzyć ułamki ze znakiem
  - W takiej sytuacji dodana nowa składowa zmienia interpretację wartości odziedziczonych składowych
  - Semantyka składowych klasy podstawowej nie może zmieniać się w klasie pochodnej

Dziedziczenie uwagi

- Nie każde określenie „bycia czymś" wyrażone zdaniem określa relacje dziedziczenia
- Dziedziczenia należy unikać gdy
  - Nie wszystkie dziedziczone operacje są przydatne
  - Zmienia się znaczenie dziedziczonych operacji
  - Zmienia się znaczenie dziedzicznych składowych
  - Właściwości składowych klasy bazowej zostają zawężone w klasie pochodnej
- Przypisanie dowolnego obiektu klasy pochodnej obiektowi klasy podstawowej zawsze musi mieć sens (opuszczenie dodatkowych składowych zawartych w klasie pochodnej)